

Verifying Non-blocking Data Structures with Manual Memory Management

Sebastian Wolff

2021

Von der Carl-Friedrich-Gauß-Fakultät der Technischen Universität Carolo-Wilhelmina zu Braunschweig zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Dissertation von Sebastian Wolff. Eingereicht am 03. März 2021. Disputation am 25. Juni 2021.

Referenten: Prof. Roland Meyer, Prof. Rupak Majumdar, Prof. Constantin Enea.

PhD Thesis
Online Version

“ *In practice, who is going to make the one thing that does everything when you can make a hundred things that do each thing perfectly.* ” — Neil deGrasse Tyson [2018[🔗](#)]

Abstract

Verification of concurrent data structures is one of the most challenging tasks in software verification. The topic has received considerable attention over the course of the last decade. Nevertheless, human-driven techniques remain cumbersome and notoriously difficult while automated approaches suffer from limited applicability. This is particularly true in the absence of garbage collection. The intricacy of non-blocking manual memory management (manual memory reclamation) paired with the complexity of concurrent data structures has so far made automated verification prohibitive.

We tackle the challenge of automated verification of non-blocking data structures which manually manage their memory. To that end, we contribute several insights that greatly simplify the verification task. The guiding theme of those simplifications are *semantic reductions*. We show that the verification of a data structure's complicated target semantics can be conducted in a simpler and smaller semantics which is more amenable to automatic techniques. Some of our reductions rely on *good conduct* properties of the data structure. The properties we use are derived from practice, for instance, by exploiting common programming patterns. Furthermore, we also show how to automatically check for those properties under the smaller semantics.

The main contributions are: (i) A compositional verification approach that verifies the memory management and the data structure separately. The approach crucially relies on a novel specification formalism for memory management implementations that over-approximates the reclamation behavior. (ii) A notion of weak ownership that applies when memory is reclaimed and reused. Weak ownership bridges the gap between techniques for garbage collection, which can assume exclusive access to owned memory, and manual memory management, where dangling pointers break such exclusivity guarantees. (iii) A notion of pointer races and harmful ABAs the absence of which ensures that the memory management does not influence the operations of the data structure, i.e., it behaves as if executed under garbage collection. Notably, we show that a check for pointer races and harmful ABAs only needs to consider executions where at most a single address is reused. (iv) A notion of strong pointer races the absence of which entails the absence of ordinary pointer races and harmful ABAs. We devise a highly-efficient type check for strong pointer races. This results in a light-weight analysis that first type checks a data structure and then performs the actual verification under garbage collection using an off-the-shelf verifier. (v) Experimental evaluations that substantiate the usefulness of the aforementioned contributions. To the best of our knowledge, we are the first to fully automatically verify practical non-blocking data structures with manual memory management.

Contents

Preface	7
1 Introduction	8
Contribution 1: SMR Specifications and Compositional Verification	10
Contribution 2: Ownership for Manual Memory Reclamation	10
Contribution 3: Avoiding Reallocations	11
Contribution 4: Verification under Garbage Collection	12
Outlook	13
I Preliminaries	14
2 Non-blocking Data Structures	15
2.1 Linearizability	15
2.2 Fine-grained Synchronization	16
2.3 Manual Memory Reclamation	17
2.3.1 Free Lists	19
2.3.2 Epoch-Based Reclamation	21
2.3.3 Hazard Pointers	23
2.4 Data Structure Implementations	25
2.4.1 Stacks	26
2.4.2 Queues	28
2.4.3 Sets	30
3 Model of Computation	37
3.1 Memory, or Heaps and Stacks	37
3.2 Syntax of Programs	38
3.3 Semantics of Commands	38
3.4 Semantics of Programs	39
4 Thread-Modular Analysis	42

II	Contributions	43
5	Compositional Verification	44
5.1	SMR Automata	45
5.2	SMR Specifications	46
5.3	Verification Relative to SMR Automata	48
6	Ownership and Reclamation	51
6.1	Reclamation breaks Ownership	51
6.2	Regaining Ownership	55
6.3	Evaluation	57
6.3.1	Integrating Safe Memory Reclamation	57
6.3.2	Linearizability Experiments	57
7	Pointer Races	59
7.1	Similarity of Computations	59
7.2	Preserving Similarity	63
7.3	Detecting ABAs	66
7.4	Reduction Result	69
7.5	Evaluation	70
7.5.1	Soundness checks	70
7.5.2	Linearizability Experiments	71
7.5.3	Verifying SMR Implementations	73
8	Strong Pointer Races	75
8.1	Annotations	76
8.2	Avoiding All Reallocations	78
8.3	A Type System to Prove Strong Pointer Race Freedom	79
8.3.1	Guarantees	80
8.3.2	Types	80
8.3.3	Type Rules	83
8.3.4	Soundness	85
8.4	Example	87
8.4.1	Type Transformer Relation	88
8.4.2	Angels	88
8.4.3	Typing	89
8.4.4	Annotations	90
8.4.5	Hazard Pointers	90
8.5	Invariant Checking	92
8.6	Type Inference	94
8.7	Avoiding Strong Pointer Races	96

8.8	Evaluation	99
III	Discussion	101
9	Related Work	102
9.1	Data Structures	102
9.2	Memory Reclamation	103
9.3	Reasoning and Verification	103
9.3.1	Memory Safety	103
9.3.2	Typestate	104
9.3.3	Program Logics	105
9.3.4	Linearizability	105
9.3.5	Moverness	106
10	Future Work	108
11	Conclusion	111
	Bibliography	112
	Acknowledgements	128
	Appendices	129
A	Additional Material	130
A.1	Compositionality	130
A.2	Hazard Pointer Specification	132
A.3	Relaxation of Strong Pointer Races	134
B	Meta Theory	136
B.1	Formal Definitions	136
B.2	Compositionality	142
B.3	Ownership	143
B.4	Reductions	143
B.5	Type System	148
C	Proof of Meta Theory	150
C.1	Compositionality	150
C.2	Ownership	159
C.3	Reductions	160
C.4	Type System	211

Preface

Parts of this thesis have already appeared in one of the following peer-reviewed publications:

- [1] Frédéric Haziza, Lukás Holík, Roland Meyer, and **Sebastian Wolff**. 2016. *Pointer Race Freedom*. In: *VMCAI*, LNCS vol. 9583. Springer. [✂ DOI:10.1007/978-3-662-49122-5_19](https://doi.org/10.1007/978-3-662-49122-5_19)
Relevant for: Chapter 6.
- [2] Roland Meyer and **Sebastian Wolff**. 2019. *Decoupling lock-free data structures from memory reclamation for static analysis*. In: *PACMPL 3 (POPL)*. [✂ DOI:10.1145/3290371](https://doi.org/10.1145/3290371)
Relevant for: Chapters 1 to 5 and 7.
- [3] Roland Meyer and **Sebastian Wolff**. 2020. *Pointer life cycle types for lock-free data structures with memory reclamation*. In: *PACMPL 4 (POPL)*. [✂ DOI:10.1145/3371136](https://doi.org/10.1145/3371136)
Relevant for: Chapters 1, 3, 8 and 9.

Further publications related to this thesis:

- [4] Lukás Holík, Roland Meyer, and Tomáš Vojnar, and **Sebastian Wolff**. 2017. *Effect Summaries for Thread-Modular Analysis - Sound Analysis Despite an Unsound Heuristic*. In: *SAS*, LNCS vol. 10422. Springer. [✂ DOI:10.1007/978-3-319-66706-5_9](https://doi.org/10.1007/978-3-319-66706-5_9)
- [5] Roland Meyer and **Sebastian Wolff**. 2018. *Reasoning About Weak Semantics via Strong Semantics*. In: *Principled Software Development*, Springer. [✂ DOI:10.1007/978-3-319-98047-8_18](https://doi.org/10.1007/978-3-319-98047-8_18)

Technical reports of [1-4] are available as:

- [6] Frédéric Haziza, Lukás Holík, Roland Meyer, and **Sebastian Wolff**. 2015. *Pointer Race Freedom*. In: *CoRR* abs/1511.00184. [✂ arxiv.org/abs/1511.00184](https://arxiv.org/abs/1511.00184)
- [7] Lukás Holík, Roland Meyer, and Tomáš Vojnar, and **Sebastian Wolff**. 2017. *Effect Summaries for Thread-Modular Analysis*. In: *CoRR* abs/1705.03701. [✂ arxiv.org/abs/1705.03701](https://arxiv.org/abs/1705.03701)
- [8] Roland Meyer and **Sebastian Wolff**. 2018. *Decoupling lock-free data structures from memory reclamation for static analysis*. In: *CoRR* abs/1810.10807. [✂ arxiv.org/abs/1810.10807](https://arxiv.org/abs/1810.10807)
Relevant for: Appendices B and C.
- [9] Roland Meyer and **Sebastian Wolff**. 2019. *Pointer life cycle types for lock-free data structures with memory reclamation*. In: *CoRR* abs/1910.11714. [✂ arxiv.org/abs/1910.11714](https://arxiv.org/abs/1910.11714)
Relevant for: Appendices A to C.

A web page accompanying this thesis is available at: [✂ https://wolff09.github.io/phd/](https://wolff09.github.io/phd/)

Introduction

Software is ubiquitous. Today, it is the driving force behind controlling and managing all sorts of systems ranging from microwave ovens to critical infrastructure. While one may survive unscathed a cold meal as the result of defective oven software, quite the opposite is true for defects in medical equipment and transportation. Famously, and even more so tragically, a computer-aided radiation therapy device from the early 1980s, the Therac-25, suffered from a software defect [Leveson and Turner 1993]. The result: massive radiation overdoses which resulted in at least six patients dying. Fast forward several decades and software is much more widely spread in safety-critical systems. Yet, defects still endanger and claim the lives of people. In the 2000s, Toyota replaced with software the physical connection between the acceleration pedal and engine in some of their cars. The software malfunctioned [Barr 2013; CBS News 2010; Yoshida 2013a,b]. The result: around ninety passengers were killed in car accidents as the car would accelerate uncontrollably. In 2019, a software defect in e-scooters was reported, locking the wheels at potentially high velocities [Carson 2019]. The result: several injured riders. The list of software defects causing economic loss and human damage goes on [Charette 2005].

The above brief history of software defects calls for thorough software verification. It needs to be checked that software is *correct*, that is, behaves as intended. A basic building block of software are data structures. They are the backbone of virtually all programs across all areas of application [Mehta and Sahni 2004]. Their importance in programming is best summarized by Wirth [1978]:

“Algorithms + Data Structures = Programs.”

The question of how to store and access data is fundamentally mission-critical, so efficient and correct data structure implementations are imperative. In times of highly concurrent computing being available even on commodity hardware, concurrent implementations are needed. In practice, the class of non-blocking data structures has been shown to be particularly efficient [Harris 2001; Henzinger et al. 2013a; Ladan-Mozes and Shavit 2004; Michael 2002a; Wu et al. 2016]. Using fine-grained synchronization and avoiding such synchronization whenever possible results in unrivaled performance and scalability. Unfortunately, this use of fine-grained synchronization is what makes non-blocking data structures also unrivaled in terms of complexity. Indeed, bugs have been discovered in published non-blocking data structures [Doherty et al. 2004a; Michael and Scott 1995]. This confirms the need for verification. More specifically, this confirms the need for formal proofs of correctness: the inherent non-determinism of concurrency renders testing techniques unable to make defects acceptably improbable [Clarke 2008].

Data structure verification has received considerable attention over the past decade (Chapter 9 gives a detailed overview). Doherty et al. [2004b], for example, give a manual (mechanized) proof of a non-blocking queue. Such

proofs require a tremendous effort and a deep understanding of the data structure and the verification technique. Or, as Clarke and Emerson [1981] put it:

*“The task of [manual] proof construction can be quite tedious,
and a good deal of ingenuity may be required.”*

Automated approaches remove this burden. Vafeiadis [2010a,b], for instance, verifies singly-linked data structures fully automatically.

Surprisingly, many proofs presented in the literature, whether manual or automatic, are unfit for practice. The reason for this is that most techniques are restricted to implementations that rely on a garbage collector (GC) [Abdulla et al. 2016; Cao et al. 2017; Krebbers et al. 2018]. 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 requires manual memory management (MM). That is, it is the programmer’s obligation to avoid memory leaks by reclaiming memory that is no longer in use (using `free` or `delete`). Hence, manual memory management is also referred to as manual memory reclamation. In non-blocking data structures, this task is much harder than it may seem at first glance. The root of the problem is that threads typically traverse the data structure without synchronization. This leads to threads holding pointers to objects that have already been removed from the structure. If objects are reclaimed immediately after the removal, those threads are in danger of accessing deleted memory. Such accesses are considered unsafe (undefined behavior in C/C++ [ISO 2011]) and are a common cause for system crashes due to a `segfault`. The solution to this problem are so-called *safe memory reclamation* (SMR) algorithms [Michael 2002b]. Their task is to provide non-blocking means for deferring the reclamation/deletion until all unsynchronized threads have finished their accesses. This is done by replacing explicit deletions with calls to a function `retire` provided by the SMR algorithm which defers the deletion. To defer the deletion sufficiently long, the SMR algorithm relies on feedback from the data structure. To that end, threads issue *protections* of the memory that they are going to access. A protection requests the SMR algorithm to defer the deletion of the protected memory until the protection is revoked. The exact form of protections depends on the SMR algorithm. Coming up with efficient and practical SMR implementations is difficult [Brown 2015; Cohen 2018; Nikolaev and Ravindran 2020] and an active field of research (cf. Chapter 9).

The use of SMR algorithms to manage manually the memory of non-blocking 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 non-blocking in order not to spoil the non-blocking guarantee of the data structure using it. In fact, SMR algorithms are quite similar to non-blocking data structures implementation-wise. So far, this added complexity could not be tamed in a principled way by automatic verifiers.

The present thesis tackles the challenge of automatically verifying non-blocking data structures which use SMR. To make the verification tractable, we contribute several insights that greatly simplify the verification task. The guiding theme of those simplifications are *semantic reductions*. We show that the verification of a program’s complicated target semantics can be done in a simpler and smaller semantics which is more amenable to automatic techniques. For instance, we show that verifiers can ignore manual memory management altogether and instead assume a garbage collector (cf. Contribution 4 below). Our reductions typically rely on *good conduct* properties of the program.

The properties we rely on are derived from practice and exploit common programming patterns, like avoiding dereferences of dangling pointers. Besides practically motivated properties, we also show how to automatically check for those properties under the smaller semantics. We summarize our contributions.

Contribution 1: SMR Specifications and Compositional Verification

We propose a compositional verification technique [de Roever et al. 2001]. We split up the single, monolithic task of verifying a non-blocking data structure together with its SMR implementation into two separate tasks: verifying the SMR implementation and verifying the data structure implementation without the SMR implementation. At the heart of our approach is a specification of the SMR behavior. Crucially, this specification has to capture the influence of the SMR implementation on the data structure. Our main observation is that there is no influence. More precisely, there is no *direct* influence. The SMR algorithm influences the data structure only *indirectly*: the data structure retires to-be-reclaimed memory, the SMR algorithm eventually reclaims the memory, and then the data structure can reuse the reclaimed memory.

In order to come up with an SMR specification, we exploit the above observation as follows. We let the specification define when reclaiming retired memory is allowed. Then, the SMR implementation is correct if the reclamations it performs are a subset of the reclamations allowed by the specification. For verifying the data structure, we use the SMR specification to over-approximate the reclamations of the SMR implementation. This way we over-approximate the influence the SMR implementation has on the data structure, provided the SMR implementation is correct. Hence, our approach is sound for solving the original verification task.

Towards lightweight SMR specifications, we rely on the insight that SMR implementations, despite their complexity, implement rather simple temporal properties [Gotsman et al. 2013]. These temporal properties are incognizant of the actual SMR implementation. Instead, they reason about those points in time when a call of an SMR API function is invoked or returns. We exploit this by having SMR specifications judge when reclamation is allowed based on the *history* of SMR function invocations and returns. Technically, we introduce SMR automata to specify SMR implementations. SMR automata are similar to ordinary finite-state automata plus more powerful acceptance criteria.

With SMR automata at hand, we are ready for compositional verification. Given an SMR automaton, we first check that the SMR implementation is correct wrt. that automaton. Second, we verify the data structure. To that end, we strip away the SMR implementation and let the SMR automaton execute the reclamation. More precisely, we non-deterministically delete those parts of the memory which are allowed to be reclaimed according to the SMR automaton. The verification result is sound since the SMR automaton over-approximates the influence the SMR implementation can have on the data structure.

Contribution 2: Ownership for Manual Memory Reclamation

Data structures are typically implemented as part of concurrency libraries. Hence, we aim to verify them for all possible future use cases. In particular, this means to verify them for an arbitrary number of concurrent client

threads, rather than a fixed number of clients. To do so, thread-modular reasoning is employed [Berdine et al. 2008; Flanagan and Qadeer 2003b; Jones 1983; Owicki and Gries 1976]: threads are verified individually, abstracting away from the relation between threads. Intuitively, the technique splits up system states into partial states that reflect a single thread’s perception of the overall state. To account for the interaction among threads, the updates of each thread are recorded in a so-called interference set. Partial thread states are then subject to spontaneous updates from that set. Applying an interference update, however, suffers from imprecision. For example, parts of a thread’s partial state may be modified despite being inaccessible to other threads in the original system state. Such spurious updates arise since the relation between threads got lost due to the abstraction. The imprecision leads to false alarms in practice.

To rule out false alarms, spurious interference updates need to be identified and discarded. Ownership reasoning is a well-known and widely applied technique for that purpose [Castegren and Wrigstad 2017; Dietl and Müller 2013; Gotsman et al. 2007; O’Hearn 2004; Vafeiadis and Parkinson 2007]. Under garbage collection, ownership refers to the fact that a thread has exclusive access to parts of the memory. Here, exclusivity means that other threads can neither write nor read the owned memory. Hence, ownership entails a strict separation of owned memory when applying interference updates. The separation makes thread-modularity precise enough for verification to be practical under GC.

When memory is managed manually, however, the strong exclusivity guarantees of the above notion of ownership do not apply. The reason for this are dangling pointers. They can observe another thread’s reallocation of previously reclaimed memory and subsequently access the now owned memory. Altogether, this means that ownership reasoning as applied under GC is unsound under MM. This inapplicability of well-performing GC techniques makes MM verifiers imprecise and scale poorly [Abdulla et al. 2013; Vafeiadis 2010a,b].

We overcome the issue of lacking ownership that makes automated techniques under MM imprecise. We reintroduce ownership in a weakened form: ownership may be broken by dangling pointers but retains the strong exclusivity guarantees for non-dangling pointers. We substantiate the claims of improved precision with experimental evidence. Interestingly, our experiments reveal that it is less relevant whether or not dangling pointers challenge the exclusivity, that is, read or write owned memory. It is the exclusivity wrt. non-dangling pointers that improves existing analyses, both in terms of precision and scalability.

Contribution 3: Avoiding Reallocations

Although our compositional approach localizes the verification effort, it leaves the verification tool with a hard task: verifying shared-memory programs with memory reuse. Even with ownership reasoning, the task remains too hard for automated verification to be practical for complex data structures or complex SMR algorithms. To overcome this problem, we suggest verification under a simpler semantics, a semantics that tames the complexity of reasoning about memory reuse. More specifically, we prove sound that it suffices to consider reusing a single memory location only. The rationale behind this result is the following. From the literature we know that avoiding memory reuse altogether is not sound for verification [Michael and Scott 1996]. Put differently, correctness under garbage

collection does not imply correctness under manual memory management via SMR. The discrepancy becomes evident in the ABA problem. An ABA is a scenario where a pointer to address a is changed to point to address b and back to a again. Under MM, a thread might erroneously conclude that the pointer has never changed if the intermediate value was not seen due to a certain interleaving. Typically, the root of the problem is that address a is removed from the data structure, reclaimed, reallocated, and reenters the data structure. Under GC, the exact same code does not suffer from this problem. A pointer to address a prevents it from being reused.

From ABAs we learn that avoiding memory reuse does not allow for a sound analysis. Surprisingly, it turns out that any discrepancy between GC and MM manifests as an ABA. So our goal is to check with little overhead to a GC analysis whether or not the program under scrutiny suffers from the ABA problem. If not, correctness under GC implies correctness under MM. Otherwise, we reject the program and verification fails.

We propose a lightweight ABA check that requires reallocations of a single address only. Note that a program is free from ABAs if it is free from *first* ABAs. Fixing the problematic address a of such a first ABA allows us to avoid reuse of any address except a while retaining the ability to detect the ABA. Intuitively, this is the case because the first ABA is the first time the program reacts differently on a reused address than on a fresh address. Hence, replacing reallocations with allocations of fresh addresses before the first ABA retains the behavior of the program.

We implemented the ABA check and a GC analysis in a tool to verify data structures and SMR implementations. Our experiments confirm the usefulness of the reduction. To the best of our knowledge, our tool is the first to automatically verify non-blocking data structures which use intricate SMR algorithms.

Contribution 4: Verification under Garbage Collection

The above result comes with a promising generalization that we already hinted at: the actual verification task can be conducted under garbage collection. This suggests the use off-the-shelf GC verifiers. Soundness, however, requires the program to be free from ABAs. To check this requires us to inspect memory deletions and reallocations of at least a single address. Deletions and reallocations, in turn, prohibit the use of GC verifiers. Even worse: we need custom verifiers with techniques tailored towards manual memory management, techniques that are still inefficient and imprecise despite the effort that the research community puts forward [Abdulla et al. 2013; Holík et al. 2017].

We seek to overcome the limited applicability of MM verifiers and their customization in order to establish ABA freedom. To that end, we present a type system a successful type check of which guarantees the absence of ABAs. The key insight behind the type system is that in every ABA at least one dangling pointer participates. Indeed, for a pointer to observe that an address is retired, reclaimed, and reused, the pointer has to continuously reference that address—the pointer is dangling. If a dangling pointer is used, we let the type check fail. As a result, a successful type check entails ABA freedom. In fact, a successful type check also guarantees memory safety in the sense that all dereferences are safe.

The main challenge for the type system is to syntactically detect the semantic property of whether or not a pointer is dangling. Due to the lack of synchronization in non-blocking data structures, a pointer may become dangling without a thread noticing. Programmers are aware of the problem. They use the protection mechanism of the SMR algorithm in such a way that the deletion of retired objects is guaranteed to be deferred, effectively preventing pointers from becoming dangling. To cope with this, our types integrate knowledge about the SMR algorithm. More specifically, a pointer's type at some program location over-approximates the reclamation behavior of the SMR algorithm for the address held by the pointer, for all executions reaching the program location. Consequently, types allow us to detect when a pointer may become dangling. Technically, we assume we are given an SMR automaton specifying the SMR algorithm in use and let types denote sets of states of the SMR automaton. A core aspect of our development is that the actual SMR automaton is an input to our type system—it is not tailored towards a specific SMR automaton.

In practice, a pure syntactic approach as the one described above lacks precision. To guide the type check's detection of dangling pointers, we exploit shape invariants [Jones and Muchnick 1979], i.e., invariants capturing the correlation of pointers and objects in memory at runtime. Type systems, however, typically cannot detect such invariants. We embrace this weakness. A design decision of our type system is that it does not track shape information nor alias information. Instead, we rely on light-weight annotations to mark pointers referencing non-retired objects. To relieve the programmer from arguing about annotations, we automatically prove their correctness and place them in a guess-and-check manner [Flanagan and Leino 2001]. Surprisingly, we can refute incorrect annotations under GC with off-the-shelf verifiers.

We implemented a tool that performs a type check, checks annotations for correctness, and invokes an existing GC verifier for the actual analysis. Our experiments confirm that the type check is highly efficient. Furthermore, we confirm the practicality of discharging annotations with an off-the-shelf verifier. To the best of our knowledge, our tool is the first to automatically verify non-blocking set data structures which use SMR algorithms.

Outlook

The remainder of the thesis is structured in three parts.

Preliminaries are discussed in Part I. Chapter 2 gives a primer on non-blocking data structures and their memory management. Chapter 3 makes precise the programming model, i.e., the syntax and semantics of programs. Chapter 4 reviews an existing analysis for non-blocking data structures that we reuse and expand.

The contributions are presented in detail in Part II. Chapter 5 introduces SMR automata and a compositional verification approach. Chapter 6 lifts ownership to apply to manual memory management. Chapter 7 presents an analysis that need not explore all reallocations. Chapter 8 reduces the verification to a type check and verification under garbage collection semantics.

The thesis is concluded in Part III. Chapter 9 discusses related work. Chapter 10 offers directions for future work. Chapter 11 summarizes the results.

Part I

Preliminaries

Non-blocking Data Structures

The present thesis is concerned with the verification of high-performance concurrent data structures, more specifically, with *non-blocking* implementations [Herlihy and Shavit 2008; Michael and Scott 1996; Treiber 1986]. Non-blocking refers to the use of fine-grained, low-level synchronization rather than traditional locking techniques. To avoid ambiguities, we clarify the terminology. In the literature, there are three so-called progress guarantees [Herlihy and Shavit 2008, Section 3.7]: obstruction-freedom, lock-freedom, and wait-freedom. Obstruction-freedom is the weakest guarantee and requires, intuitively, that at any given point any given thread can make progress if it is executed in isolation, i.e., without interference from other threads. Lock-freedom requires obstruction freedom and that there always is a thread that can make progress even in the presence of interference. Wait-freedom is the strongest guarantee. It requires that all threads can make progress at any given point in time. Since we are concerned with verification, we need not distinguish between these progress guarantees. We stick with non-blocking to uniformly refer to any of the above guarantees. While we follow this convention hereafter, note that some works use the terms lock-free and non-blocking interchangeably [Agesen et al. 2000; Cohen and Petrank 2015a; Greenwald 1999] or use the term lock-free to refer to the absence of locks/mutexes [Barnes 1993; Michael and Scott 1996].

The remainder of this chapter gives a primer on non-blocking data structures—it is not strictly necessary for the understanding of the contributions presented in Chapters 5 to 8 but details the practical concepts that shaped them. The structure is as follows. Section 2.1 introduces the correctness criterion for concurrent data structures that we aim to verify. Section 2.2 examines low-level synchronization. Section 2.3 discusses memory management, a critical aspect in non-blocking data structures. Section 2.4 gives non-blocking data structure implementations from the literature which we use as benchmarks throughout this thesis.

2.1 Linearizability

We introduce linearizability [Herlihy and Wing 1990], the de-facto standard correctness criterion for concurrent data structures [Zhu et al. 2015]. Intuitively, linearizability asks for each method of a data structure to take effect instantaneously at some point—the *linearization point*—between the method’s invocation and response. This makes linearizability appealing from a user’s perspective. It provides the illusion of atomicity, allowing the user to rely on a much simpler sequential specification of the data structure. Such sequential specifications are called the *abstract data type (ADT)* of the data structure. ADTs can be given as simple sequential programs or in more general mathematical terms [Abdulla et al. 2013; Vafeiadis 2010b]. Our development does not depend on the formalism used for describing ADTs. For verification, linearizability is appealing as well. The composition of linearizable components is linearizable again [Herlihy and Shavit 2008, Section 3.5], allowing to verify them individually.

For a formal definition of linearizability we need some definitions. An execution E is a sequence of method invocation and response events evt . Invocations take the form $evt = \text{in}:meth(t, \bar{v})$ where $meth$ is the invoked method, t is the invoking thread, and \bar{v} are the actual parameters. Responses take the form $evt = \text{re}:meth(t, \bar{v})$ where $meth$ is the returning method, t is the executing thread, and \bar{v} are the return values. An invocation and a response match if they refer to the same method $meth$ and are executed by the same thread t . An execution is complete if every invocation has a matching response. A complete execution is sequential if every invocation is immediately followed by a matching response. Two executions E and E' are equivalent if all per-thread subsequences of E and E' coincide. More precisely, E and E' are equivalent if $E|_t = E'|_t$ for all threads t , where $E|_t$ is the subsequence of all events of thread t in E and similarly for E' .

To achieve linearizability, we require that every execution E can be mapped to an equivalent sequential execution S such that the real-time behavior is preserved, that is, the order of non-overlapping method calls in E is preserved in S . More formally, we say that S preserves the real-time behavior of E , if for all response events evt_1 that precede an invocation event evt_2 in E , evt_1 precedes evt_2 in S . Additionally, we require that the sequential execution S is legal, i.e., contained in the set of executions produced by the ADT. For this exposition of linearizability, we assume a procedure to check membership for that set.

Lastly, we need to take care of incomplete executions. As they might contain multiple invocations with pending responses, they cannot be mapped to a sequential execution. A completion of E is a complete execution E' that coincides with E up to invocations without matching responses being removed or receiving a matching response at the end of E' . The following definition summarizes the discussion.

Definition 2.1 (Linearizability [Herlihy and Wing 1990]). An execution E is linearizable if there are executions E' and S such that: (i) E' is a completion of E , (ii) E' is equivalent to S , (iii) S is sequential, (iv) S is legal, and (v) S preserves the real-time behavior of E' .

2.2 Fine-grained Synchronization

Non-blocking implementations avoid traditional locking techniques in favor of fine-grained, low-level synchronization primitives. Those primitives are fine-grained in that they operate over a single or a small, fixed number of words,¹ rather than critical sections of mutual exclusion which may operate over unboundedly many such words. Low-level synchronization primitives typically correspond to atomic read-modify-write operations, implemented directly in hardware. As such, fine-grained synchronization promises better performance than locking.

Compare-and-swap (CAS) [IBM 1983] is the most common synchronization primitive in non-blocking data structures. Pseudo code for a placeholder type T is given in Figure 2.2. A standard CAS takes three arguments: `&dst`, `cmp`, and `src`. The first argument, `&dst`, is a reference to a word in memory. The remaining arguments, `cmp` and `src`, are values. A CAS compares the word referenced by `&dst` with `cmp`. If equal, the word referenced by `&dst` is replaced by `src`.

¹ A *memory word* is loosely defined as a unit of the underlying hardware architecture which it can transfer in a single step [Stallings 2013, p. 14]. Modern commodity hardware usually has a word size of 32 or 64 bits [Arm Limited 2020; Intel Corporation 2016].

Figure 2.2: Standard, double-word, and two-word compare-and-swap (CAS) mock implementations for a placeholder type `T`. Modern processors implement CAS in hardware, like the `CMPXCHG` instruction on x86 [Intel Corporation 2016].

```
1 bool CAS(T& dst, T cmp, T src) { // standard
2     atomic {
3         if (dst == cmp) { dst = src; return true; }
4         else { return false; }
5     } }

6 bool CAS(T& dst1, T cmp1, T src1, T& dst2, T cmp2, T src2) { // double-word / two-word
7     // double-word version assumes that 'dst1' and 'dst2' are subsequent words in memory
8     atomic {
9         if (dst1 == cmp1 && dst2 == cmp2) { dst1 = src1; dst2 = src2; return true; }
10        else { return false; }
11    } }
```

and `true` is returned. Otherwise, no update is performed and `false` is returned. Double-word CAS is a variant which operates over two words stored consecutively in memory instead of a single word `&dst`. Another variant is two-word CAS. It is similar to double-word CAS, however, operates over two arbitrary words. While the distinction between consecutive and arbitrary words may seem unnecessarily cumbersome, it is important for data structure designers. Many modern hardware architectures, like x86, support standard and double-word CAS, but do not implement two-word CAS [Intel Corporation 2016, p. 3-181 ff.]. The more powerful two-word CAS and its generalization to k -word CAS require slower software solutions, like RDCSS [Harris et al. 2002]. Hence, data structure designers avoid them. We write 2CAS to point out two-word CAS usages.

It is worth pointing out that locks can be implemented using CAS [Herlihy and Shavit 2008, Section 7.2]. As a result, avoiding locks in favor of CAS does not necessarily make an implementation non-blocking.

Besides CAS, *load-link/store-conditional* (LL/SC) [Jensen et al. 1987] is another common synchronization primitive. It is available, for instance, on ARM processors [Arm Limited 2020, p. B2-166]. Intuitively, a load-link and subsequent store-conditional to the same address behaves like an ordinary load-store pair with the difference that the store-conditional fails if the address has been updated since the load-link was executed. Since LL/SC can be used to implement any of the above CAS [Anderson and Moir 1995] and since it is less common in the data structure literature, we restrict our presentation to CAS.

2.3 Manual Memory Reclamation

In the absence of a garbage collector, which runs in the background and automatically reclaims unused memory, it is the programmer's task to reclaim unused memory manually. In C/C++, for instance, this is done using the primitives `free` or `delete`. While manual reclamation tends to be rather simple when lock-based synchronization is used [Brown 2015; Nikolaev and Ravindran 2020], it becomes substantially harder for fine-grained, non-blocking

Figure 2.3: A simple counter with unsynchronized readers. The implementation is flawed in that it leaks memory. Naively deleting the leaked memory in Line 29, however, is unsafe.

```
12 struct Container {
13     int data;
14 }
15
16 shared Container* Counter;
17
18 atomic init() {
19     Counter = new Container();
20     Counter->data = 0;
21 }
22
23 int increment() {
24     Container* inc = new Container();
25     while (true) {
26         Container* curr = Counter;
27         int out = curr->data;
28         inc->data = out+1;
29         if (CAS(Counter, curr, inc)) {
30             // delete curr;
31             return out;
32         }
33     }
34 }
```

synchronization. As discussed in Section 2.2, fine-grained synchronization relies on CAS and the like. This leads to optimistic update patterns [Moir and Shavit 2004] where threads (i) create a local snapshot of the current state of the data structure, (ii) compute an update based on the local snapshot, and (iii) publish via CAS the update if the data structure has not changed since the snapshot was taken or retry otherwise. Optimistic update patterns, in turn, lead to unsynchronized readers. The mentioned local snapshot is typically created without regard for the updates of other threads. For memory reclamation, this means that it is the reclaiming thread's task to ensure that deletions do not harm other threads. To that end, the reclaiming thread needs to ensure that all unsynchronized readers of the to-be-deleted memory have finished their accesses. This, however, requires an unexpectedly complicated machinery [Brown 2015; Cohen and Petrank 2015a; Fraser 2004; Michael 2002b].

We illustrate the problems with non-blocking manual memory reclamation on an example. Therefore, consider the implementation of a simple counter from Figure 2.3. It consists of a shared pointer variable `Counter`, Line 16, which points to an object storing a single `int`. The `Counter`'s value is initialized to 0, Lines 19 and 20, by method `init` which we assume is executed atomically once before the counter implementation is used. Method `increment` proceeds in the aforementioned optimistic manner. It reads out the current `Counter` into the local pointer `curr`, Line 25. Next, it stores the incremented value of `curr->data` in a newly allocated object `inc`, Line 27. Then, `increment` tries to install `inc` as the new `Counter`. This is done via a CAS, Line 28, which ensures that `Counter` is still equal to `curr`. Observe that this CAS ensures that `inc` indeed contains the incremented value of the current `Counter`. If the CAS succeeds, the pre-increment value of the counter is returned, Line 30. Otherwise, `increment` restarts and retries the procedure.

Despite its simplicity, the counter implementation is flawed. It leaks memory. The object referenced by `curr` is not reclaimed after a successful CAS. The naive fix for this leak is to uncomment the deletion from Line 29. This fix, however, is *unsafe*. Other threads might access the counter concurrently. Since they do so without (read) synchronization, they will access the to-be-deleted object without any precautions. In C/C++, for example, such use-after-free accesses have undefined behavior and can result in a system crash due to a `segfault` [ISO 2011].

Figure 2.4: An implementation of free lists (FL) for a placeholder type `T`. Retired objects are added to a (sequential) thread-local list. Objects from that list can be reused immediately.

```
32  threadlocal List<T*> freeList;
33
34  void retire(T* pointer) {
35      freeList.push(pointer);
36  }
37  T* reuse() {
38      if (freeList.empty()) return NULL;
39      T* result = freeList.pop();
40      return result;
41  }
```

To avoid both memory leaks and unsafe operations, programmers employ so-called *safe memory reclamation (SMR)*. SMR algorithms provide means for deferring deletions until it is safe, that is, until all concurrent readers have finished their accesses. To that end, SMR algorithms commonly offer a function² `retire` to request the deferred deletion of an object, replacing ordinary deletion via `delete`. As is standard for `delete`, no object must be retired multiple times in order to avoid malicious double frees—all SMR implementations we are aware of rely on this. The actual deferring mechanism varies vastly among SMR algorithms. It relies on feedback from the data structure the form of which also varies among SMR algorithms.

It is worth pointing out that deferred deletion is the only viable solution for data structures to be non-blocking when manually managing their memory. The alternative would be to integrate into the dereference of a pointer a check for its integrity, i.e., a check if the referenced object has not yet been deleted. Such a check, however, typically relies on reading out part of the data structure (shared memory). Hence, it cannot be done atomically together with the dereference when relying on fine-grained synchronization primitives.

In the remainder of this section we survey essential SMR algorithms that most other techniques build upon or are derived from: free-lists (Section 2.3.1), epoch-based reclamation (Section 2.3.2), and hazard pointers (Section 2.3.3). Traditional garbage collection is not among the techniques as it is blocking [Cohen 2018]. See Chapter 9 for a broader overview of existing techniques.

2.3.1 Free Lists

The simplest approach to deferred deletion is indefinite deferral, i.e., avoiding memory reclamation altogether. To avoid leaks, retired objects are stored in a thread-local *free list* (FL) [IBM 1983; Treiber 1986]. The objects from that list can be reused in favor of allocating new memory. Figure 2.4 gives an example implementation. Notably, the implementation relies on an initially empty list, Line 32, which may be sequential as only a single thread accesses it.

To use FL with the counter implementation from above, we have to retire unused objects and, if possible, reuse retired objects instead of allocating new ones. Moreover, we have to carefully revise the CAS installing the new counter value (cf. Line 28). The possibility for memory being reused immediately after its retirement allows for

² To avoid ambiguities, we refer to the operations offered by a data structure as *methods* and to the operations offered by an SMR algorithm as *functions*.

Figure 2.5: An adaption of the simple counter to reuse memory via FL. Tagged pointers are used to avoid the ABA problem. Modifications wrt. Figure 2.3 are marked in bold font.

```

42 struct Container {
43     int data;
44 }
45
46 shared int Tag;
47 shared Container* Counter;
48
49 atomic init() {
50     Tag = 0;
51     Counter = new Container();
52     Counter->data = 0;
53 }

54 int increment() {
55     Container* inc = reuse();
56     if (inc == NULL) inc = new Container();
57     while (true) {
58         int tag = Tag;
59         Container* curr = Counter;
60         int out = curr->data;
61         inc->data = out+1;
62         if (CAS(Tag, tag, tag+1, Counter, curr, inc)) {
63             retire(curr);
64             return out;
65     } } }

```

the infamous ABA problem [Michael and Scott 1996]. Generally speaking, an ABA is a scenario where a pointer referencing address a is changed to point to address b and changed back to point to a again. A thread might erroneously conclude that the pointer has never changed if the intermediate value goes unnoticed due to a certain interleaving. Typically, the root of the problem is that address a is removed from the data structure, reused, and reenters the data structure. More specifically, an ABA may arise in the counter implementation as follows. Let thread t execute `increment` up to Line 28. That is, t has read out the current `Counter`, say at address a , has read out its value `out`, and is about to install `out+1` as the new value of the counter. Assume t is interrupted by another thread t' . Let thread t' increment the counter, installing value `out+1` and retiring address a . If t' performs another increment, it might reuse address a to install `out+2`. Now, the CAS of t succeeds although the counter has been updated: t erroneously decreases the counter's value from `out+2` to `out+1` where an increase to `out+3` was expected. It is readily checked that this violates linearizability.

Under garbage collection, the exact same code does not suffer from ABAs: a pointer referencing address a would prevent it from being reused. To overcome the problem under manual memory management, pointers are instrumented to carry an integer *tag*, or modification counter [IBM 1983; Michael and Scott 1996; Treiber 1986]. To avoid ABAs then, (i) updating a pointer also increases the tag, and (ii) comparisons of pointers take their tags into account. The solution is amenable for fine-grained synchronization: pointers and tags can be handled atomically with double-word CAS [Michael 2002a] or by *stealing* unused bits of pointers to use as storage for the tag [Herlihy and Shavit 2008, Section 9.8]. Consider Figure 2.5 for a modified counter implementation using FL and tags.

A significant drawback of FL is the fact it does not support *arbitrary reuse* [Michael 2002b]. Once allocated, memory always remains allocated for the process. Even worse, the use of tagged pointers mandates that the memory must not be used outside the data structure as otherwise tags might get corrupted and ABAs resurface. This may make FL unfavorable in practice. The SMR algorithms discussed next address this issue.

Figure 2.6: An adaption of the simple counter to reuse memory via EBR. Modifications wrt. Figure 2.3 are marked in bold font.

```
66 struct Container {
67     int data;
68 }
69
70 shared Container* Counter;
71
72 atomic init() {
73     Counter = new Container();
74     Counter->data = 0;
75 }
76
77 int increment() {
78     Container* inc = new Container();
79     while (true) {
80         Container* curr = Counter;
81         int out = curr->data;
82         inc->data = out+1;
83         if (CAS(Counter, curr, inc)) {
84             retire(curr);
85             enterQ();
86             return out;
87         }
88     }
89 }
```

2.3.2 Epoch-Based Reclamation

Epoch-based reclamation (EBR) [Fraser 2004; Harris 2001] implements a simple form of time-stamping to identify when retired objects cannot be accessed anymore and their reclamation is safe. To that end, EBR offers the two functions `leaveQ` and `enterQ`. Threads use the former to announce that they are going to access the data structure and use the latter to announce that they have finished the access. The function names, in particular the `Q`, refer to the fact that the threads are *quiescent* [McKenney and Slingwine 1998] between `enterQ` and `leaveQ`, meaning they do not modify the data structure. During the non-quiescent period, EBR guarantees that shared reachable objects are not reclaimed, even if they are removed from the data structure and retired. This makes EBR easy to apply, as illustrated by the counter implementation from Figure 2.6.

Technically, EBR relies on two assumptions to realize the aforementioned guarantee: (i) threads do not have pointers to any object during their quiescent phase, and (ii) objects are retired only after being removed from the data structure, i.e., after being made unreachable from the shared variables. Those assumptions imply that no thread has or can acquire a pointer to a removed object if every thread has been quiescent at some point since the removal. So it is safe to delete a retired object if every thread has been quiescent at some point since the retire. To detect this, EBR introduces *epoch counters*, a global one and one for each thread. Thread-local epochs are single-writer multiple-reader counters. Whenever a thread invokes a method, it reads the global epoch e and announces this value by setting its thread epoch to e . Then, it scans the epochs announced by the other threads. If all agree on e , the global epoch is advanced to $e + 1$. The fact that all threads must have announced the current epoch e for it to be updated to $e + 1$ means that all threads have invoked a method after the epoch was changed from $e - 1$ to e . That is, all threads have been in-between calls. Thus, deleting objects retired in the global epoch $e - 1$ becomes safe from the moment when the global epoch is updated from e to $e + 1$. To perform those deletions, every thread keeps a list of retired objects for every epoch and stores objects passed to `retire` in the list for the current thread-local epoch. For the actual deletion it is important to note that the thread-local epoch may lack behind the global epoch by up to 1. As a consequence, a thread may put a object retired during the global epoch e into its retired-list for

Figure 2.7: An implementation of epoch-based reclamation (EBR) for a placeholder type T . The implementation supports dynamic thread joining and parting.

```

88 struct EbrRec {
89     EbrRec* next;
90     bool used;
91     int epoch;
92     List<T*> retired0, retired1, retired2;
93 }
94
95 shared int GEpoch;
96 shared EbrRec* LEpochs;
97 threadlocal EbrRec* myEpoch;
98
99 atomic init() {
100     Epochs = NULL;
101     GlobalEpoch = 0;
102 }
103
104 void join() {
105     myEpoch = new EbrRec();
106     myEpoch->used = true;
107     myEpoch->epoch = GEpoch;
108
109     while (true) {
110         EbrRec* recs = LEpochs;
111         myEpoch->next = recs;
112         if (CAS(LEpochs, recs, myEpoch)) break;
113     }
114
115     void part() { myEpoch->used = false; }
116
117     void retire(T* ptr) { myEpoch->retired0.push(ptr); }
118
119     void leaveQ() {
120         int epoch = GEpoch;
121         myEpoch->epoch = epoch;
122
123         EbrRec* tmp = LEpochs;
124         while (tmp != NULL) {
125             if (!tmp->used) continue;
126             if (epoch != tmp->epoch) return;
127             tmp = tmp->next;
128         }
129
130         int nextEpoch = (epoch + 1) % 3;
131         if (!CAS(GEpoch, epoch, nextEpoch)) return;
132
133         myEpoch->epoch = nextEpoch;
134         for (T* ptr : myEpoch->retired2) delete ptr;
135         retired2.clear();
136         retired2.swap(retired1);
137         retired1.swap(retired0);
138     }
139
140     void enterQ() { /* do nothing */ }

```

epoch $e - 1$. So for a thread during its local epoch e it is not safe to delete the objects in the retired-list for $e - 1$ because they may have been retired during the global epoch e . It is only safe to delete the objects contained in the retired-list for epochs $e - 2$ and smaller. Hence, it suffices to maintain three retired-lists. Progressing to epoch $e + 1$ allows for deleting the objects from the local epoch $e - 2$ and to reuse that retired-list for epoch $e + 1$.

An example EBR implementation is given in Figure 2.7. The thread-local epochs and retired-lists are stored in a singly-linked list of `EbrRec` objects rooted in the shared pointer `LEpochs`, Line 96. For a proper initialization, we assume that every thread invokes `join` as part of its construction and part during its tear down. Function `retire` simply places the to-be-deleted object in the thread-local retired-list `retired0`, Line 116. Function `leaveQ` implements the epoch progression, Lines 122 to 130, and deferred deletion process, Lines 132 to 136, as described above. Besides the core EBR functionality, the implementation supports dynamic thread joining and parting. That is, new threads may be created and existing threads may be destroyed while the implementation is in use—there is no need for dedicated start up and tear down phases where all threads are present that ever wish to participate. Joining threads allocate and publish a new epoch entry, Lines 105 to 112. Parting threads mark their epoch entry as inactive via the `used` flag of `EbrRec`, Line 114. Inactive entries are skipped by `leaveQ` when scanning the epoch entry list for consensus with the global epoch. This is crucial as otherwise epoch entries of parted threads would prevent the

global epoch from being progressed, thus preventing reclamation. Notably, inactive entries are never removed and reclaimed as this would require means of safe memory reclamation. We leave it to the reader to improve the implementation so that it reuses marked entries for joining threads.

EBR improves on FL from Section 2.3.1 in many aspects. First of all, it allows for arbitrary reuse. Second, it is easier to use than FL as it usually prevents the ABA problem (given proper usage). Lastly, quite efficient implementations exist in practice [Brown 2015; Hart et al. 2007]. On the downside, EBR does not support thread failures, or more generally threads that stop executing `leaveQ` without having called `part`. As noted above, this prevents reclamation³ because the global epoch cannot be progressed anymore. Hazard pointers, which we discuss next, do not suffer from this problem.

2.3.3 Hazard Pointers

The hazard pointer (HP) [Michael 2002b] method provides a protection mechanism for individual objects. Protections signal that an object is still in use and that its deletion should be deferred. To be precise, HP guarantees that *the deletion of an object is deferred if it has been continuously protected since before it was retired* [Gotsman et al. 2013]. Figure 2.8 gives a simplified version of the HP implementation due to Michael [2004]. The implementation equips every thread with a fixed number of single-writer multiple-reader pointers, the eponymous hazard pointers. We refer to the i -th hazard pointer of thread t by `hpt[i]` and may drop the thread subscript if clear from the context. Protections are issued by a call to function `protect`. It takes as parameters an object and a hazard pointer index, and simply writes the object to the hazard pointer with the given index, Line 168. Protections can be revoked by `unprotect` which takes a hazard pointer index and resets the corresponding hazard pointer by writing `NULL` to it, Line 172. Protections are respected as follows. Function `retire` stores the passed object into a thread-local list of retired objects, Line 176. Moreover, it periodically tries to reclaim the objects from that list, Line 177. To do so, it scans the hazard pointers of all threads, collecting all the objects that are currently protected, Lines 181 to 188. Then, a retired object is reclaimed if it is not in the list of protected objects, Lines 190 to 194. Similar to the EBR implementation from Section 2.3.2, HP supports dynamic thread joining and parting. Again, parting threads do not reclaim internal `HpRec` objects. Unlike for EBR, thread failures do not stop reclamation; failures may only prevent reclamation of objects protected by crashed thread.

Figure 2.9 presents a version of the simple counter from Figure 2.3 adapted to reclaim memory using HP with a single hazard pointer per thread. While the HP method is conceptually simple, it may be non-trivial to detect whether or not an object has been protected successfully, i.e., if an object has been protected before it was retired. In the counter implementation, we need to protect `curr` because it is subsequently accessed. To that end, a protection is issued using hazard pointer `hp[0]`. At this point, we cannot guarantee that a dereference of `curr` is safe. Between reading out `curr` in Line 208 and protecting it in Line 209, an interfering thread might have updated the counter

³ The literature may deem memory reclamation schemes blocking if they allow for an unbounded number of objects awaiting reclamation [Balmau et al. 2016]. The reason for this is that an unbounded backlog of reclamation is assumed to result in the system running out of memory, making subsequent allocations block/wait until memory is reclaimed. The issue is irrelevant for the present thesis: we do not suggest how to construct non-blocking data structures but verify the correctness of existing ones. EBR remains an essential technique as many SMR algorithms are derived from it or are drop-in replacements for it (cf. Chapter 9).

Figure 2.8: A simplified version of the hazard pointer (HP) implementation by Michael [2004] for a placeholder type `T` and `K` hazard pointers per thread. The implementation supports dynamic thread joining and parting.

```

140 struct HpRec {
141     HpRec* next;
142     Array<T*, K> hp; // 0-indexed
143     List<T*> retired;
144 }
145
146 shared HpRec* HPtrs;
147 threadlocal HpRec* myHP;
148
149 atomic init() {
150     HPtrs = NULL;
151 }
152
153 void join() {
154     myHP = new HpRec();
155     while (true) {
156         HpRec* recs = HPtrs;
157         myHP->next = recs;
158         if (CAS(HPtrs, recs, myHP)) {
159             break;
160         }
161     }
162
163     void part() {
164         for (int i = 0; i < K; ++i) {
165             hp[i] = NULL;
166         }
167     }
168
169     void protect(T* ptr, int index) {
170         assert(0 <= index < K);
171         myHP->hp[index] = ptr;
172     }
173
174     void unprotect(int index) {
175         protect(NULL, index);
176     }
177
178     void retire(T* ptr, int index) {
179         myHP->retired.push(ptr);
180         if (*) reclaim();
181     }
182
183     void reclaim() {
184         List<T*> defer;
185         HpRec* tmp = HPtrs;
186         while (tmp != NULL) {
187             for (int i = 0; i < K; ++i) {
188                 defer.push(tmp->hp[i]);
189             }
190             tmp = tmp->next;
191         }
192
193         for (T* ptr : myHP->retired) {
194             if (defer.contains(ptr)) continue;
195             myHP->retired.remove(ptr);
196             delete ptr;
197         }
198     }
199 }

```

and retired the object referenced by `curr`. Because the protection was not yet announced, `curr` might have already been reclaimed. Line 210 checks that this is not the case. It does so by ensuring that `curr` coincides with the shared Counter. It is worth pointing out that this check relies on the invariant that Counter is never retired. Only after both Lines 209 and 210 have been executed, `curr` can be accessed safely. As we will see in Section 2.4, this procedure for successfully protecting pointers is common in non-blocking data structures. Unfortunately, we will also see that there are data structures that are fundamentally incompatible with this procedure and the HP method in general [Brown 2015; Michael 2002b].

The alert reader readily realizes that the counter implementation using HP, more precisely the protection check from Line 210, is prone to the ABA problem. Indeed, as noted above the object referenced by `curr` could have been reclaimed. Consequently, it could have been reused and installed as the shared Counter again. Those scenarios are not problematic since we just ensure that `curr` contains the current value of Counter and that it has been protected successfully. Put differently, Lines 208 to 210 appear as if they were executed atomically. Accesses to the content of `curr` happen only later.

Figure 2.9: An adaption of the simple counter to reuse memory via HP. A single hazard pointer per thread is required. Modifications wrt. Figure 2.3 are marked in bold font.

```

195 struct Container {
196     int data;
197 }
198
199 shared Container* Counter;
200
201 atomic init() {
202     Counter = new Container();
203     Counter->data = 0;
204 }
205
206 int increment() {
207     Container* inc = new Container();
208     while (true) {
209         Container* curr = Counter;
210         protect(curr, 0);
211         if (curr != Counter) continue;
212         int out = curr->data; inc->data = out+1;
213         if (CAS(Counter, curr, inc)) {
214             retire(curr);
215             unprotect(0);
216             return out;
217         }
218     }
219 }

```

Finally, we revisit the guarantee that *the deletion of an object is deferred if it has been continuously protected since before it was retired*. It is imperative to make precise the notion of continuous protections. A single hazard pointer's protection is continuous. More involved data structures, however, use multiple hazard pointers to protect a single object [Michael 2002a]. A common pattern first issues a protection per $hp[i]$ and later, in order to reuse $hp[i]$, issues a protection per $hp[i+1]$ and resets $hp[i]$. We say that the protection is *transferred* from $hp[i]$ to $hp[i+1]$. The order is important [Michael 2002a]: a protection can be transferred from $hp[i]$ to $hp[j]$ only if $i < j$. This is because of the scanning process from method `reclaim`, Lines 181 to 188 in Figure 2.8. It reads out hazard pointers in ascending order. Hence, protections can go unrecognized when attempting to transfer from $hp[j]$ to $hp[i]$ with $i < j$. To see this, consider a thread t protecting an object o with $hp_t[1]$. Assume that another thread t' executes function `reclaim` up to the point where it scans $hp_t[0]$ but not $hp_t[1]$. Now, let t protect o per $hp_t[0]$ and reset $hp_t[1]$. Then, t' misses the protection of o —it is not transferred from $hp_t[1]$ to $hp_t[0]$. Altogether, this means that a protection is continuous only if it is due to a single hazard pointer or due to transfers among multiple hazard pointers.

2.4 Data Structure Implementations

We give an overview of the non-blocking data structures from the literature that are used as benchmarks throughout this thesis. We focus on singly-linked stacks, queues, and sets with manual memory management via the SMR algorithms discussed in Section 2.3. All implementations use objects of type `Node` from Figure 2.10 as internal representation. A `Node` contains a single data value, field `data`, a boolean flag for marking purposes, field `mark`, and a pointer for establishing the link structure, field `next`. Some implementations do not use the `mark` field; for simplicity, we do not introduce another type without the `mark` field.

Regarding the presentation, we do not give individual implementations for each SMR technique. Instead, we mark with bold font the lines of code that are needed for SMR usage and prefix them with F, E, or H if they are specific to

Figure 2.10: Node type for singly-linked data structures. Member fields `mark` and `next` are stored consecutively in memory, so both fields can be modified with a single double-word CAS.

```
217 struct Node {
218     int data;
219     bool mark;
220     Node* next;
221
222     Node(int value) { data = value; mark = false; next = NULL; } // constructor
223 }
```

FL, EBR, or HP, respectively. For FL, we simplify the presentation further: we do not make explicit the use of tags and memory reuse. Instead, we implicitly assume that all pointers are equipped with tags and that new tries to reuse memory before allocating new one.

2.4.1 Stacks

Stack data structures are simple collections of data items with *last-in-first-out* behavior. Elements are added to and removed from the top of the stack.

Treiber’s Stack. The earliest documented non-blocking data structure is the stack due to Treiber [1986], given in Figure 2.11. The implementation maintains a NULL-terminated singly-linked list of nodes rooted in the shared top-of-stack pointer `ToS`. If the stack is empty, `ToS` points to NULL. New nodes are pushed to the stack by creating a local copy `top` of `ToS`, Line 232, linking the new node as a predecessor of `top`, Line 235, and installing node as the new `ToS` via a CAS, Line 236. The CAS checks that the stack has not changed since `top` was read out. This ensures that node, which coincides with the new value of `ToS` after the update, links to the old value of `ToS`. Existing values are popped as follows. First, a local copy `top` of `ToS` is created, Line 245. If `top` equals NULL, then the implementation signals that the stack is empty, Line 246. Otherwise, the implementation attempts to remove the top node. To that end, a pointer `next` to the second node of the stack is read out, Line 249. Then, a CAS tries to install `next` as the new value of `ToS` if the stack has not changed. In the case the CAS succeeds, the value stored in the removed top node is returned. Otherwise, the implementation retries.

Treiber’s stack can be combined with SMR algorithms easily. Common to all SMR algorithms is the need to retire popped elements, Line 252. The SMR specific modifications follow. FL requires explicit reuse of retired nodes and tags to avoid the ABA problem—as stated above, we do not make this explicit in the code, it is analogous to what we have seen in Figure 2.5. For EBR, we need to add `leaveQ` and `enterQ` calls to the methods. For HP, we have to protect the top pointer. Similarly to the counter from Figure 2.9, we do so by issuing `protect` for `hp[0]` and ensure that the protected `top` coincides to `ToS`, Lines 233 and 234 in `push` as well as Lines 247 and 248 in `pop` [Michael 2002b]. It is an invariant of Treiber’s stack (and all data structures that follow) that the shared reachable nodes are never retired. Hence, the protection is guaranteed to be successful: we can safely access the pointer and avoid the ABA problem.

Figure 2.11: Treiber’s non-blocking stack [Treiber 1986] with SMR. The HP version is by Michael [2002b].

```

224 shared Node* ToS;
225
226 atomic init() { ToS = NULL; }
227
228 void push(int input) {
229     E leaveQ();
230     Node* node = new Node(input);
231     while (true) {
232         Node* top = ToS;
233         H protect(top, 0);
234         H if (top != ToS) continue;
235         node->next = top;
236         if (CAS(&ToS, top, node)) break;
237     }
238     H unprotected(0);
239     E enterQ();
240 }

241 int pop() {
242     E leaveQ();
243     int output = EMPTY;
244     while (true) {
245         Node* top = ToS;
246         if (top == NULL) break; // empty
247         H protect(top, 0);
248         H if (top != ToS) continue;
249         Node* next = top->next;
250         if (CAS(&ToS, top, next)) {
251             output = top->data;
252             FEH retire(top);
253         }
254         H unprotected(0);
255     }
256     E enterQ();
257     return output;
258 }

```

Figure 2.12: Optimized version of Treiber’s non-blocking stack with HP [Michael 2002b]. Compared to the original version, Figure 2.11, the push operation does not take any precautions wrt. memory reclamation and the ABA problem. Yet the implementation is correct.

```

258 shared Node* ToS;
259
260 atomic init() { ToS = NULL; }
261
262 void push(int input) {
263     // no SMR needed
264     Node* node = new Node(input);
265     while (true) {
266         Node* top = ToS;
267         node->next = top;
268         if (CAS(&ToS, top, node))
269             break
270     }
271 }

271 int pop() {
272     int output = EMPTY;
273     while (true) {
274         Node* top = ToS;
275         if (top == NULL) break; // empty
276         H protect(top, 0);
277         H if (top != ToS) continue;
278         Node* next = top->next;
279         if (CAS(&ToS, top, next)) {
280             output = top->data;
281             H retire(top);
282         }
283         H unprotected(0);
284     }
285     return output;
286 }

```

Optimized Treiber’s Stack. Michael [2002b] proposed an optimized version of Treiber’s stack with HP, given in Figure 2.12. The implementation avoids protections in the push method altogether. This results in an ABA: when installing node as the new value of ToS with the CAS from Line 268 (Line 236 in the original version) the stack might have changed. More precisely, interfering threads may have inserted or deleted elements. Interestingly, this does not void the correctness of the implementation. It suffices that the newly added node is linked to ToS.

Figure 2.13: Michael&Scott’s non-blocking queue [Michael and Scott 1996] with SMR. The extension to HP is due to Michael [2002b].

```

286 shared Node* Head, Tail;
287
288 atomic init() { Head=Tail=new Node(_); }
289
290 void enqueue(int input) {
291   E leaveQ();
292   Node* node = new Node(input);
293   while (true) {
294     Node* tail = Tail;
295     H protect(tail, 0);
296     H if (tail != Tail) continue;
297     Node* next = tail->next;
298     if (tail != Tail) continue;
299     if (next != NULL) {
300       CAS(&Tail, tail, next);
301       continue;
302     }
303     if (CAS(&tail->next, next, node)) {
304       CAS(&Tail, tail, node);
305       break;
306     }
307   }
308   H unprotect(0);
309   E enterQ();
310 }
311
312 int dequeue() {
313   E leaveQ();
314   int output = EMPTY;
315   while (true) {
316     Node* head = Head;
317     H protect(head, 0);
318     H if (head != Head) continue;
319     Node* tail = Tail;
320     Node* next = head->next;
321     H protect(next, 1);
322     if (head != Head) continue;
323     if (next == NULL) break; // empty
324     if (head == tail) {
325       CAS(&Tail, tail, next); continue;
326     } else {
327       output = next->data;
328       if (CAS(&Head, head, next)) {
329         FEH retire(head);
330         break;
331       }
332     }
333   }
334   H unprotect(0); unprotect(1);
335   E enterQ();
336   return output;
337 }

```

2.4.2 Queues

Queue data structures are collections of data items with *first-in-first-out* behavior. New elements are added to the end (tail) and existing elements are removed from the front (head) of a queue.

Michael&Scott’s Queue. Figure 2.13 gives the well-known implementation by Michael and Scott [1996]. It is a practical example in that it is implemented in C++ Boost’s `lockfree::queue` [Blechmann 2011] and Java’s `ConcurrentLinkedQueue` [Oracle 2020], for instance. The queue is organized as a NULL-terminated singly-linked list of nodes. The first node in the list is a dummy node, its content is not logically part of the queue. The enqueue method appends new nodes to the end of the list. To do so, an enqueueer first moves `Tail` to the last node as it may lack behind, Line 300. Then, the new node is appended by pointing `Tail->next` to it, Line 303. Last, the enqueueer tries to move `Tail` to the new node, Line 304. This can fail as another thread may have moved `Tail` already to avoid waiting for the enqueueer. The dequeue method 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, Line 325, and then moves the `Head` to that node, Line 326. Additionally, dequeue maintains the property that `Head` does not overtake `Tail` by moving `Tail` towards the end of the list if necessary, Line 323.

Figure 2.14: The DGLM non-blocking queue [Doherty et al. 2004b] with SMR. It is similar to Michael&Scott’s non-blocking queue but allows the Head to overtake the Tail.

```

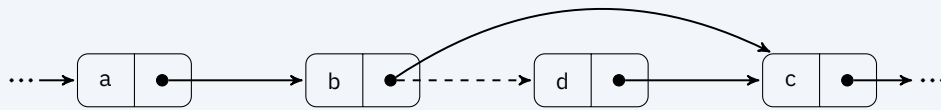
334 shared Node* Head, Tail;
335
336 atomic init() { Head=Tail=new Node(_); }
337
338 void enqueue(int input) {
339     E leaveQ;
340     Node* node = new Node(input);
341     while (true) {
342         Node* tail = Tail;
343         H protect(tail, 0);
344         H if (tail != Tail) continue;
345         Node* next = tail->next;
346         if (tail != Tail) continue;
347         if (next != NULL) {
348             CAS(&Tail, tail, next);
349             continue;
350         }
351         if (CAS(&tail->next, next, node)) break
352     }
353     CAS(&Tail, tail,node);
354     H unprotect(0);
355     E enterQ;
356 }
357
358 int dequeue() {
359     E leaveQ;
360     int output = EMPTY;
361     while (true) {
362         Node* head = Head;
363         H protect(head, 0);
364         H if (head != Head) continue;
365         Node* next = head->next;
366         H protect(next, 1);
367         if (head != Head) continue;
368         if (next == NULL) break; // empty
369         output = next->data;
370         if (CAS(&Head, head, next)) {
371             Node* tail = Tail;
372             if (head == tail) CAS(Tail, tail, next);
373             FEH retire(head);
374             break;
375         }
376     }
377     H unprotect(0); unprotect(1);
378     E enterQ;
379     return output;
380 }

```

Memory management can be added to Michael&Scott’s queue as follows. Dequeued nodes are retired after they have been made unreachable from Head, Line 327. The modifications required for FL and EBR are straight-forward, see Figure 2.13. Using HP requires more care [Michael 2002b, 2004]. We focus on the more involved dequeue method; the protections for enqueue are similar. First, head is protected with hp[0], Line 315. As before, the success of the protection needs to be ensured. This is done by checking that the shared Head still equals the local copy head. If so, the subsequent dereference of head is safe, as required for acquiring pointer next to the first non-dummy node of the queue, Line 318. Otherwise, the operation is restarted. Second, next is protected with hp[1], Line 319. If head and Head coincide, Line 320, then the queue has not changed and next is reachable from the shared pointer Head. This guarantees that next has not been retired. That is, the protection of next is successful. It is worth pointing out that ensuring the equality of next and head->next does not suffice: the fact that next is still linked to the successfully protected head does not prevent updates to the queue, removing head and next, and thus allows for next being retired.

Observe that dequeue reads out the to-be-returned data value next->value, Line 325, before the actual dequeuing, Line 326. This is done because of FL. There, it is possible that immediately after the CAS from Line 326 the node referenced by next is dequeued, retired, and reused. The reuse leads to next->value being overwritten by an interferer before the dequeuing thread can access the value that is supposed to be returned. Under garbage collection, EBR, and HP, the implementation can be optimized: moving the data read after the CAS is correct because the reuse of next is prevented.

Figure 2.15: Example memory layout of a singly-linked set. A removal of node b must ensure that the successor of b has not changed. Otherwise, an interfering insertion of node d after b (dashed line) could be lost. A simple `CAS(a->next, b, c)` is prone to this problem.



DGLM Queue. Doherty et al. [2004b] proposed a variation of Michael&Scott’s implementation, see Figure 2.14. Their dequeue method avoids congestion on the Tail pointer by ignoring the Tail until an element has been dequeued. (Michael&Scott’s queue reads out Tail in every iteration, no matter if an element is successfully dequeued or if the operation is restarted.) As a consequence, Head may overtake Tail. If so, dequeue moves Tail forward.

2.4.3 Sets

Set data structures provide collections of unique data items with insertion, removal, and lookup functionality. Singly-linked implementations typically maintain a sorted list. Sortedness poses a major challenge: unlike in stacks and queues, insertions and removals may happen anywhere in the list. To see why this is challenging, consider the list from Figure 2.15 containing subsequent nodes a, b, and c. The removal of node b requires to update the next field of node a from b to c. However, a simple `CAS(a->next, b, c)` is insufficient. Interfering threads might tamper with the link between nodes b and c. An insertion, for instance, might add a new node d after b by updating `b->next` to d (dashed line in Figure 2.15). Then, the above CAS would remove b but would also remove d unintentionally.

Several solutions for the above problem have been proposed. We present some of them, sorted by complexity in ascending order. Interestingly, however, this order opposes the chronological order of publication. Some of the simpler algorithms were proposed later in the verification literature to simplify the verification task.

Vechev&Yahav’s 2CAS Set. As demonstrated by the above example, a removal needs to check the consistency of two pointers atomically. Vechev and Yahav [2008] suggested to do so with a two-word CAS. Their implementation is given in Figure 2.16. The backbone of the implementation is the method `locate`. It is an internal helper that is not exposed to the clients of the set. For a given data value, `locate` finds two adjacent nodes `pred` and `curr` such that the value is either stored in `curr` or should be inserted between `pred` and `curr`. To find those nodes, the implementation traverses the singly-linked list from front to back. The operation restarts if a traversed node has been removed by an interfering thread. This is the case if a node’s next field is NULL, Line 439.

Lookups via `contains` check if a given value is in the set. This is done with `locate` and testing whether or not `curr` contains the searched value, Line 442. Method `insert` uses `locate` to find the appropriate insertion location. If a node with the to-be-inserted datum already exists, nothing needs to be done, Line 403. Otherwise, a new node is linked in-between `pred` and `curr`, Lines 404 and 405. Method `remove` works similarly. To ensure a correct unlinking, two-word CAS is used, Line 420. It unlinks `curr` only if `curr->next` has not changed. Moreover, the next field of the unlinked `curr` is set to NULL, making interfering threads aware of the removal.

Figure 2.16: Vechev&Yahav’s 2CAS set [Vechev and Yahav 2008, Figures 8 and 9] with SMR. The implementation of `remove` relies on a two-word CAS, Line 420.

```

379 shared Node* Head, Tail;
380
381 atomic init() {
382     Head = new Node(-∞);
383     Tail = new Node(∞);
384     Head->next = Tail;
385 }
386
387 bool contains(int value) {
388     Node* pred; Node* curr; int found;
389     E leaveQ();
390     <pred, curr, found> = locate(value);
391     H unprotect(0); unprotect(1);
392     E enterQ();
393     return found == value;
394 }
395
396 bool insert(int value) {
397     Node* pred; Node* curr; int found;
398     Node* entry = new Node(value);
399     E leaveQ();
400     bool success = false;
401     while (!success) {
402         <pred, curr, found> = locate(value);
403         if (found == value) break;
404         entry->next = curr;
405         success = CAS(pred->next, curr, entry);
406     }
407     FEH if (!success) retire(entry);
408     H unprotect(0); unprotect(1);
409     E enterQ();
410     return success;
411 }
412
413 bool remove(int value) {
414     Node* pred; Node* curr; int found;
415     E leaveQ();
416     bool success = false;
417     while (!success) {
418         <pred, curr, found> = locate(value);
419         if (found > value) break;
420         Node* next = curr->next;
421         success = 2CAS(pred->next, curr, next,
422                        curr->next, next, NULL);
423     }
424     FEH if (success) retire(curr);
425     H unprotect(0); unprotect(1);
426     E enterQ();
427     return success;
428 }
429
430 <Node*, Node*, int> locate(int value) {
431     Node* pred; Node* curr; int found;
432     assert(-∞ < value < ∞);
433     retry: // jump label
434     curr = Head;
435     do {
436         pred = curr;
437         H protect(pred, 1);
438         curr = pred->next;
439         H protect(curr, 0);
440         if (curr == NULL) goto retry;
441         found = curr->data;
442     } while (found < value);
443     return <pred, curr, found>;
444 }

```

In terms of memory management, the implementation can be adapted to use FL and EBR in the standard way. For HP, protections are issued by `locate` and revoked by the corresponding caller method, i.e., at the end of `contains`, `insert`, and `remove`. The protections in `locate` are more involved than the ones we have seen so far. The reason for this is that unboundedly many nodes may be traversed while threads have only a bounded number of hazard pointers at their disposal. To that end, `locate` uses two hazard pointers to issue protections in a *hand-over-hand* fashion [Bayer and Schkolnick 1977]. More specifically, the loop from Lines 434 to 441 assumes that pointer `curr` is protected with `hp[0]`. The protection is transferred to `hp[1]`. Recall from Section 2.3.3 that this transfer is recognized by HP. Then, `curr` is advanced to the successor node and protected with `hp[0]`. The check in Line 439 guarantees that the protection is successful as it ensures that `curr` has not been removed. For the first iteration of the loop, note that `curr` points to `Head`. Thus, no protection is needed since the dummy node `Head` is always accessible and never retired.

Figure 2.17: The ORVYY set [O’Hearn et al. 2010] with SMR. The implementation of `remove` relies on a two-word CAS, Lines 485 to 490.

```

444 shared Node* Head, Tail;
445
446 atomic init() {
447     Head = new Node(-∞);
448     Tail = new Node(∞);
449     Head->next = Tail;
450 }
451
452 bool contains(int value) {
453     Node* pred; Node* curr; int found;
454     E leaveQ();
455     <pred, curr, found> = locate(value);
456     H unprotect(0); unprotect(1);
457     E enterQ();
458     return found == value;
459 }
460
461 bool insert(int value) {
462     Node* pred; Node* curr; int found;
463     Node* entry = new Node(value);
464     E leaveQ();
465     bool success = false;
466     while (!success) {
467         <pred, curr, found> = locate(value);
468         if (found == value) break;
469         entry->next = curr;
470         success = CAS(pred->mark, false, false,
471                     pred->next, curr, entry);
472     }
473     FEH if (!success) retire(entry);
474     H unprotect(0); unprotect(1);
475     E enterQ();
476     return success;
477 }
478
479 bool remove(int value) {
480     Node* pred; Node* curr; int found;
481     E leaveQ();
482     bool success = false;
483     while (!success) {
484         <pred, curr, found> = locate(value);
485         if (found > value) break;
486         atomic { if (!pred->mark
487                 && pred->next == curr) {
488             pred->next = curr->next;
489             curr->mark = true;
490             success = true;
491         } }
492     }
493     FEH if (success) retire(curr);
494     H unprotect(0); unprotect(1);
495     E enterQ();
496     return success;
497 }
498
499 <Node*, Node*, int> locate(int value) {
500     Node* pred; Node* curr; int found;
501     assert(-∞ < value < ∞);
502     retry: // jump label
503     curr = Head;
504     do {
505         pred = curr;
506         H protect(pred, 1);
507         curr = pred->next;
508         H protect(curr, 0);
509         if (pred->mark) goto retry;
510         found = curr->data;
511     } while (found < value);
512     return <pred, curr, found>;
513 }

```

ORVYY Set. O’Hearn et al. [2010] presented a solution similar to Vechev&Yahav’s 2CAS set. Instead of indicating removed nodes via setting next fields to NULL, they use the marking technique by Prakash et al. [1994]. That is, they use the boolean mark bit of type Node and set it to true upon removal. This signals to other threads that the node is being removed and that its next field must not be changed. The implementation is given in Figure 2.17. We stick to the original atomic update proposed by O’Hearn et al. [2010], Lines 485 to 490. It can be implemented by two-word CAS. While the two-word CAS remains impractical, the marking technique brings us closer to practicality as it is essential for a standard/double-word CAS solution.

Vechev&Yahav’s CAS Set. Towards practical and non-blocking implementations, Vechev and Yahav [2008] showed that the aforementioned marking technique allows for removals with double-word CAS (or standard

Figure 2.18: Vechev&Yahav’s CAS set [Vechev and Yahav 2008, Figure 2] with SMR.

```
512 shared Node* Head, Tail;
513
514 atomic init() {
515     Head = new Node(-∞);
516     Tail = new Node(∞);
517     Head->next = Tail;
518 }
519
520 bool contains(int value) {
521     Node* pred; Node* curr; int found;
522     E leaveQ();
523     <pred, curr, found> = locate(value);
524     H unprotect(0); unprotect(1);
525     E enterQ();
526     return found == value;
527 }
528
529 bool insert(int value) {
530     Node* pred; Node* curr; int found;
531     Node* entry = new Node(value);
532     E leaveQ();
533     bool success = false;
534     while (!success) {
535         <pred, curr, found> = locate(value);
536         if (found == value) break;
537         entry->next = curr;
538         success = CAS(pred->mark, false, false,
539                      pred->next, curr, entry);
540     }
541     FEH if (!success) retire(entry);
542     H unprotect(0); unprotect(1);
543     E enterQ();
544     return success;
545 }
546
547 bool remove(int value) {
548     Node* pred; Node* curr; int found;
549     E leaveQ();
550     bool success = false;
551     while (!success) {
552         <pred, curr, found> = locate(value);
553         if (found > value) break;
554         bool flag = curr->mark;
555         Node* next = curr->next;
556         if (!CAS(curr->mark, flag, true,
557                curr->next, next, next)) continue;
558         success = CAS(pred->mark, false, false,
559                      pred->next, curr, next);
560     }
561     FEH if (success) retire(curr);
562     H unprotect(0); unprotect(1);
563     E enterQ();
564     return success;
565 }
566
567 <Node*, Node*, int> locate(int value) {
568     Node* pred; Node* curr; int found;
569     assert(-∞ < value < ∞);
570     retry: // jump label
571     curr = Head;
572     do {
573         pred = curr;
574         H protect(pred, 1);
575         curr = pred->next;
576         H protect(curr, 0); if (pred->mark) goto retry;
577         found = curr->data;
578     } while (found < value);
579     return <pred, curr, found>;
580 }
```

single-word CAS if the mark is implemented using bit stealing). Consider Figure 2.18 for the implementation. The removal of a node `curr` is performed in two steps. First, a double-word CAS sets the mark flag, Line 555. As for the ORVYY set, this prevents other threads from updating node `curr`. Then, another double-word CAS unlinks `curr` by redirecting `pred->next`, Lines 557 and 558. The latter CAS goes through only if `pred` is unmarked, ensuring that the removal does not interfere with concurrent removals of `pred`.

It is worth pointing out that the removal is considered successful only if `curr` is unlinked. The operation is restarted if any of the above CAS instructions fail. While this does not spoil correctness, it spoils the non-blocking property [Vechev and Yahav 2008]. Marking a node prevents updates of its next field. Hence, insertions and removals are blocked until the node is removed. Other threads cannot *help* to unlink the node since the unlinking (and not the marking) constitutes a successful removal. The next implementation overcomes this problem.

Figure 2.19: Michael’s set [Michael 2002a] with SMR. The extension to HP is adapted from the original implementation by Michael [2002a].

```

580 shared Node* Head;
581
582 atomic init() {
583     Head = new Node(-∞);
584 }
585
586 <Node*, Node*, int> locate(int value) {
587     Node* pred; Node* curr; int found;
588     assert(-∞ < value < ∞);
589     retry: // jump label
590     curr = Head;
591     do {
592         pred = curr;
593         H protect(pred, 1);
594         curr = pred->next;
595         H protect(curr, 0);
596         if (pred->mark) goto retry;
597         if (pred->next != curr) goto retry;
598         found = curr->data;
599         if (curr->mark) {
600             Node* next = curr->next;
601             if (CAS(pred->mark, false, false
602                 pred->next, curr, next)) {
603                 HEF retire(curr);
604                 goto retry;
605             } }
606     } while (found < value);
607     return <pred, curr, found>;
608 }
609
610 bool contains(int value) {
611     Node* pred; Node* curr; int found;
612     E leaveQ();
613     <pred, curr, found> = locate(value);
614     H unprotect(0); unprotect(1);
615     E enterQ();
616     return found == value;
617 }
618
619 bool insert(int value) {
620     Node* pred; Node* curr; int found;
621     Node* entry = new Node(value);
622     E leaveQ();
623     bool success = false;
624     while (!success) {
625         <pred, curr, found> = locate(value);
626         if (found == value) break;
627         entry->next = curr;
628         success = CAS(pred->mark, false, false,
629             pred->next, curr, entry);
630     }
631     HEF if (!success) retire(entry);
632     H unprotect(0); unprotect(1);
633     E enterQ();
634     return success;
635 }
636
637 bool remove(int value) {
638     Node* pred; Node* curr; int found;
639     E leaveQ();
640     bool success = false;
641     while (!success) {
642         <pred, curr, found> = locate(value);
643         if (found > value) break;
644         Node* next = curr->next;
645         success = CAS(curr->mark, false, true
646             curr->next, next, next);
647     }
648     if (success) {
649         if (!CAS(pred->mark, false, false
650             pred->next, curr, next))
651             locate(value);
652         FEH else retire(curr);
653     }
654     H unprotect(0); unprotect(1);
655     E enterQ();
656     return success;
657 }

```

Michael’s Set. The non-blocking implementation by Michael [2002a], a simplified version of which is given in Figure 2.19, achieves lock-freedom as follows. The first step of the removal, the marking, is considered the *logical removal*. The second step, the unlinking, is considered the *physical removal*. If the first step succeeds, then the overall removal succeeds. To allow for other threads making progress despite a node being marked, any thread may physically remove a logically removed node. To be precise, method `locate` eagerly performs physical removals of all logically removed nodes it encounters during its traversal, Lines 599 to 605, and method `remove` may return if it logically removed but failed to physically remove a node.

Figure 2.20: Harris' set [Harris 2001] with EBR. The algorithm does not support the use of HP and FL since locate traverses marked and potentially unlinked nodes.

```

657 shared Node* Head, Tail;
658
659 atomic init() {
660     Head = new Node(-∞);
661     Tail = new Node(∞);
662     Head->next = Tail;
663 }
664
665 bool unlink(Node* left, Node* lnext,
666             Node* right) {
667     if (lnext == right) return true;
668     if (CAS(left->mark, false, false
669             left->next, lnext, right)) {
670 E while (lnext != right) {
671 E     retire(lnext);
672 E     lnext = lnext->next;
673 E }
674     return true;
675 }
676 return false;
677 }
678
679 <Node*, Node*, int> locate(int value) {
680     Node* left; Node* lnext; int found;
681     assert(-∞ < value < ∞);
682     while (true) {
683         Node* right = Head;
684         bool rmark = Head->mark;
685         Node* rnext = Head->next;
686         do {
687             if (!rmark) {
688                 left = right;
689                 lnext = rnext;
690             }
691             right = rnext;
692             if (right == Tail) break;
693             rmark = right->mark;
694             rnext = right->next;
695             found = right->data;
696         } while (rmark || found < value);
697         if (unlink(left, lnext, right)
698             if (right == Tail || !right->mark)
699             return <left, right, found>;
700     } }
701
702 bool contains(int value) {
703     Node* left; Node* right; int found;
704 E leaveQ();
705     <left, right, found> locate(value);
706 E enterQ();
707     return found == value;
708 }
709
710 bool insert(int value) {
711     Node* left; Node* right; int found;
712     Node* entry = new Node(value);
713 E leaveQ();
714     bool success = false;
715     while (!success) {
716         <left, right, found> locate(value);
717         if (found == value) break;
718         entry->next = right;
719         success = CAS(left->mark, false, false,
720                     left->next, right, entry);
721     }
722 E if (!success) retire(entry);
723 E enterQ();
724     return success;
725 }
726
727 bool remove(int value) {
728     Node* left; Node* right; Node* rnext;
729     int found; bool success = false;
730 E leaveQ();
731     while (!success) {
732         <left, right, found> locate(value);
733         if (found != value) break;
734         if (right->mark) continue;
735         rnext = right->next;
736         success = CAS(right->mark, false, true,
737                     right->next, rnext, rnext);
738     }
739     if (success) {
740         if (!CAS(left->mark, false, false
741                 left->next, right, rnext))
742             locate(value);
743 E else retire(right);
744     }
745 E enterQ();
746     return success;
747 }

```

Harris's Set. Harris [2001] proposed a lazy version of the locate method for Micheal's set: instead of removing individually all logically removed nodes, sequences of subsequent logically removed nodes are deleted. To that

end, `locate` traverses over logically removed nodes to find the last unmarked node before and the first unmarked node after a sequence of marked nodes. Then, a single CAS can be used to physically remove the entire sequence. The implementation is given in Figure 2.20. Notably, the implementation is incompatible with HP [Michael 2002b]: logically removed nodes cannot be traversed with HP since one cannot guarantee that the protections of marked nodes are successful. Similarly, FL cannot be used since the retirement of logically removed nodes results in immediate reuse, potentially breaking the link structure while threads are still traversing the removed nodes.

Model of Computation

We give a formal account of the programs that the reminder of this thesis reasons about. More specifically, we introduce concurrent shared-memory programs that employ a library for safe memory reclamation (SMR).

Hereafter, we use • for irrelevant terms and values to abbreviate the exposition.

3.1 Memory, or Heaps and Stacks

Programs operate over addresses from Adr that are assigned to pointer expressions $PExp$. Pointer expressions are either pointer variables from $PVar$ or pointer selectors $a.next \in PSel$. The set of shared pointer variables accessible by every thread is $shared \subseteq PVar$. Additionally, we allow pointer expressions to hold the special value $seg \notin Adr$ denoting undefined/uninitialized pointers. There is also an underlying data domain Dom to which data expressions $DExp = DVar \uplus DSel$ evaluate. Data expressions are either data variables from $DVar$ or data selectors $a.data \in DSel$. A generalization of our development to further selectors is straightforward—we stress that our results do not rely and thus are not limited to singly-linked graph structures as the single pointer selector `next` might suggest.

We do not distinguish between the stack and the heap. Instead, we refer to both as the *memory*. It is a partial function that respects the typing:

$$m : (PExp \mapsto Adr \uplus \{seg\}) \uplus (DExp \mapsto Dom).$$

The initial memory is m_ϵ . Pointer variables p are uninitialized, $m_\epsilon(p) = seg$. Data variables u have a default value, $m_\epsilon(u) = 0$. We modify the memory with updates up of the form $[exp \mapsto v]$. Applied to a memory m , the result is the memory $m' = m[exp \mapsto v]$ defined by $m'(exp) = v$ and $m'(exp') = m(exp')$ for all $exp' \neq exp$. Below, we define computations τ which give rise to sequences of updates. We write m_τ for the memory resulting from the initial memory m_ϵ when applying the sequence of updates in τ .

3.2 Syntax of Programs

We define a core language for concurrent shared-memory programs that rely on an SMR library. Programs P using SMR implementation R , written $P(R)$, are comprised of statements $stmt$ which are defined by the following grammar:

$$\begin{aligned}
stmt &::= stmt; stmt \mid stmt \oplus stmt \mid stmt^* \mid \text{beginAtomic}; stmt; \text{endAtomic} \mid com \\
com &::= p := q \mid p := q.\text{next} \mid p.\text{next} := q \mid u := \text{op}(\bar{u}) \mid u := q.\text{data} \mid p.\text{data} := u \\
&\quad \mid \text{assume } cond \mid p := \text{malloc} \mid \text{free}(p) \mid \text{in:func}(\bar{r}) \mid \text{re:func} \mid \text{skip} \mid \text{env}(a) \\
cond &::= p = q \mid p \neq q \mid \text{pred}(\bar{u})
\end{aligned}$$

We assume a strict typing that distinguishes between data variables $u, u' \in DVar$ and pointer variables $p, q \in PVar$. Functions take pointer and data variables as parameters, $r \in PVar \cup DVar$. Notation \bar{r} is short for r_1, \dots, r_n and similarly for \bar{u} . The language includes sequential composition, non-deterministic choice, Kleene iteration, and atomic blocks. The primitive commands include assignments, memory accesses, assumptions, memory allocations and deallocations, and non-nested SMR function invocations and responses. Additionally, there are non-deterministic updates of unallocated addresses a . We assume that those updates are not part of the program itself but performed by the environment.

3.3 Semantics of Commands

We define a semantics where program $P(R)$ is executed by a possibly unbounded number of concurrently operating threads. Formally, the *standard semantics*¹ of $P(R)$ is the set $\llbracket P(R) \rrbracket_X^Y$ of computations. It is defined relative to two sets $Y \subseteq X \subseteq \text{Adr}$ of addresses allowed to be reallocated and freed, respectively. A computation is a sequence τ of actions act that are of the form $act = \langle t, com, up \rangle$. The action indicates that thread t executes command com which results in the memory update up . To make the semantics precise, let $\text{fresh}_\tau \subseteq \text{Adr}$ be the set of addresses which have never been allocated in τ and let $\text{freed}_\tau \subseteq \text{Adr}$ be the set of addresses which have been freed since their last allocation. Then, the definition of the standard semantics is by induction. The empty computation is always contained, $\epsilon \in \llbracket P(R) \rrbracket_X^Y$. An action act can be appended to a computation $\tau \in \llbracket P(R) \rrbracket_X^Y$, denoted by $\tau.act \in \llbracket P(R) \rrbracket_X^Y$, if act respects the control flow of $P(R)$ and one of the rules from Figure 3.1 applies. The semantics of commands is standard. Note that we assume a sequentially consistent memory model [Lamport 1979]. That is, memory reads always obtain the latest value written. Weaker memory models are beyond the scope of this thesis.

The above definition of the standard semantics focuses on how commands interact with the memory. What it means for an action to respect the control flow is made precise next. We separate the two aspects since the methods we propose in Chapters 5 to 8 exploit the semantics of commands rather than the structure and control flow of programs.

¹ In Chapter 5 we will define a non-standard semantics that is beneficial for verification, see Figure 5.9.

Figure 3.1: Semantics of commands.

- (Skip) If $act = \langle t, \text{skip}, \emptyset \rangle$.
- (Assign1) If $act = \langle t, p.\text{next} := q, [a.\text{next} \mapsto b] \rangle$ then $m_\tau(p) = a$ and $m_\tau(q) = b$.
- (Assign2) If $act = \langle t, p := q, [p \mapsto m_\tau(q)] \rangle$.
- (Assign3) If $act = \langle t, p := q.\text{next}, [p \mapsto m_\tau(a.\text{next})] \rangle$ with $m_\tau(q) = a \in \text{Adr}$.
- (Assign4) If $act = \langle t, u := \text{op}(u'_1, \dots, u'_n), [u \mapsto d] \rangle$ with $d = \text{op}(m_\tau(u'_1), \dots, m_\tau(u'_n))$.
- (Assign5) If $act = \langle t, p.\text{data} := u, [a.\text{data} \mapsto m_\tau(u)] \rangle$ with $m_\tau(p) = a \in \text{Adr}$.
- (Assign6) If $act = \langle t, u := q.\text{data}, [u \mapsto m_\tau(a.\text{data})] \rangle$ with $m_\tau(q) = a \in \text{Adr}$.
- (Assume) If $act = \langle t, \text{assume } exp \triangleq exp', \emptyset \rangle$ then $m_\tau(exp) \triangleq m_\tau(exp')$.
- (Malloc) If $act = \langle t, p := \text{malloc}, [p \mapsto a, a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d] \rangle$ then address a is allocatable, that is, we have $a \in \text{fresh}_\tau$ or $a \in \text{freed}_\tau \cap Y$.
- (Free) If $act = \langle t, \text{free}(p), \emptyset \rangle$ then $m_\tau(p) \in X$.
- (Call) If $act = \langle t, \text{in:func}(\bar{r}), \emptyset \rangle$, then $m_\tau(r) \in \text{Adr} \uplus \text{Dom}$ for every r in \bar{r} .
- (Return) If $act = \langle t, \text{re:func}, \emptyset \rangle$.
- (Atomic) If $act = \langle t, \text{beginAtomic}, \emptyset \rangle$ or $act = \langle t, \text{endAtomic}, \emptyset \rangle$.
- (Env) If $act = \langle \perp, \text{env}(a), [a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d] \rangle$ with $a \in \text{fresh}_\tau \cup \text{freed}_\tau$.

3.4 Semantics of Programs

We give a small-step operational semantics (SOS) for programs [Plotkin 1981]. To that end, we define a transition relation \rightarrow among pairs (pc, τ) of control locations pc and computations τ . Intuitively, \rightarrow produces the reachable control locations together with the computation that led there. A control location pc is a map from threads t to statements st . We understand st as the code that remains to be executed by t . For the SOS rules, we extend ordinary statements to:

$$st ::= stmt \mid \text{inatomic } st \mid st_1 \circ st_2 \mid \text{await } func$$

in order to make explicit the execution of atomic blocks and the call stack of functions. The commands that appear in the actions of computations remain unchanged. The transition relation \rightarrow is based on a control-flow relation \xrightarrow{com} among statements st . More precisely, $st \xrightarrow{com} st'$ indicates that performing a step of st executes command com after which st' remains to be executed. Formally, \rightarrow and \xrightarrow{com} are the smallest relations that satisfy the rules from Figure 3.2. The first set of rules, Figure 3.2a, addresses ordinary statements $stmt$ and atomic blocks. The rules are standard. The second set of rules, Figure 3.2b, manages the explicit call stack $st_1 \circ st_2$. Here, st_1 is the caller, i.e., code of the data structure P , and st_2 is the callee, i.e., code of an invoked SMR function from R or skip if no function is invoked at the moment. Rule (SOS-STD-CALL) looks up the code of the

Figure 3.2: SOS rules for the standard semantics, $\llbracket P(R) \rrbracket_X^Y$.

(a) Control-flow relation \xrightarrow{com} for ordinary statements and atomic blocks.

$$\begin{array}{c}
 \begin{array}{l}
 \text{(SOS-STD-COM)} \\
 \frac{}{com \xrightarrow{com} skip}
 \end{array}
 \quad
 \begin{array}{l}
 \text{(SOS-STD-SEQ1)} \\
 \frac{}{skip; st \xrightarrow{skip} st}
 \end{array}
 \quad
 \begin{array}{l}
 \text{(SOS-STD-SEQ2)} \\
 \frac{st_1 \xrightarrow{com} st'_1}{st_1; st_2 \xrightarrow{com} st'_1; st_2}
 \end{array}
 \quad
 \begin{array}{l}
 \text{(SOS-STD-CHOICE)} \\
 \frac{i \in \{1, 2\}}{st_1 \oplus st_2 \xrightarrow{skip} st_i}
 \end{array} \\
 \\
 \begin{array}{l}
 \text{(SOS-STD-LOOP1)} \\
 \frac{}{st^* \xrightarrow{skip} skip}
 \end{array}
 \quad
 \begin{array}{l}
 \text{(SOS-STD-LOOP2)} \\
 \frac{}{st^* \xrightarrow{skip} st; st^*}
 \end{array}
 \quad
 \begin{array}{l}
 \text{(SOS-STD-ATOMIC3)} \\
 \frac{}{inatomic \text{ endAtomic} \xrightarrow{endAtomic} skip}
 \end{array} \\
 \\
 \begin{array}{l}
 \text{(SOS-STD-ATOMIC1)} \\
 \frac{}{beginAtomic; st; endAtomic \xrightarrow{beginAtomic} inatomic \ st}
 \end{array}
 \quad
 \begin{array}{l}
 \text{(SOS-STD-ATOMIC2)} \\
 \frac{st \xrightarrow{com} st'}{inatomic \ st \xrightarrow{com} inatomic \ st'}
 \end{array}
 \end{array}$$

(b) Control-flow relation \xrightarrow{com} for managing the explicit call stack.

$$\begin{array}{c}
 \begin{array}{l}
 \text{(SOS-STD-CALL)} \\
 \frac{st \xrightarrow{in:func(\bar{r})} st'}{st \circ skip \xrightarrow{in:func(\bar{r})} st' \circ R.func; \text{await } func}
 \end{array}
 \quad
 \begin{array}{l}
 \text{(SOS-STD-RETURN)} \\
 \frac{st \xrightarrow{re:func} st'}{st \circ \text{await } func \xrightarrow{re:func} st' \circ skip}
 \end{array} \\
 \\
 \begin{array}{l}
 \text{(SOS-STD-DS)} \\
 \frac{st_1 \xrightarrow{com} st'_1 \quad in:\bullet \not\equiv com \not\equiv re:\bullet}{st_1 \circ st_2 \xrightarrow{com} st'_1 \circ st_2}
 \end{array}
 \quad
 \begin{array}{l}
 \text{(SOS-STD-SMR)} \\
 \frac{st_2 \xrightarrow{com} st'_2 \quad in:\bullet \not\equiv com \not\equiv re:\bullet}{st_1 \circ st_2 \xrightarrow{com} st_1 \circ st'_2}
 \end{array}
 \end{array}$$

(c) SOS transition relation $\dashv\vdash$.

$$\begin{array}{c}
 \begin{array}{l}
 \text{(SOS-STD-ENV)} \\
 \frac{act \in Act(\tau, \perp, env(a))}{(pc, \tau) \dashv\vdash (pc, \tau.act)}
 \end{array}
 \quad
 \begin{array}{l}
 \text{(SOS-STD-PAR)} \\
 \frac{st \xrightarrow{com} st' \quad act \in Act(\tau, t, com) \quad \nexists t'. t \neq t' \wedge locked(pc(t'))}{(pc[t \mapsto st], \tau) \dashv\vdash (pc[t \mapsto st'], \tau.act)}
 \end{array}
 \end{array}$$

invoked function, $R.func$, and appends command $\text{await } func$. We use $\text{await } func$ to synchronize the callee with the caller, Rule (SOS-STD-RETURN). This ensures that invocations $in:func(\bar{r})$ receive a matching response $re:func$. Rules (SOS-STD-DS) and (SOS-STD-SMR) handle the cases where the call stack is irrelevant, falling back to the rules from Figure 3.2a. In fact, method invocations are asynchronous as we do not impose an order in which the caller and the callee execute. Our development is oblivious to this fact. Lastly, the third set of rules, Figure 3.2c, defines the SOS transition relation. To turn commands com executed by threads t into actions, we write $Act(\tau, t, com)$ to obtain the set of actions $act = \langle t, com, up \rangle$ such that $\tau.act$ satisfies the semantics of commands defined in Section 3.3 above. Rule (SOS-STD-ENV) updates unallocated memory non-deterministically, simulating the envi-

ronment. Rule (SOS-STD-PAR) executes a step of thread t if no other thread is currently within an atomic block, as defined by:

$$\begin{aligned}
 \text{locked}(\text{inatomic } st) &:= \text{true} \\
 \text{locked}(st_1; st_2) &:= \text{locked}(st_1) \\
 \text{locked}(st_1 \circ st_2) &:= \text{locked}(st_1) \vee \text{locked}(st_2) \\
 \text{locked}(st) &:= \text{false} && \text{otherwise .}
 \end{aligned}$$

Now, we say that a computation τ respects the control flow of a program $P(R)$ if there is a control location pc that witnesses τ , that is, if:

$$(pc_{init}, \epsilon) \Rightarrow^* (pc, \tau) \quad \text{with} \quad pc_{init} = \lambda t. P^{[t]} \circ \text{skip} .$$

Here, \Rightarrow^* is the reflexive transitive closure of \Rightarrow . The initial control location pc_{init} maps every thread to execute P . Since memories do not consider threads when valuating variables, we need to rename the local variables in P . The t -renamed version of P is $P^{[t]}$. For simplicity, we omitted this renaming when calling functions, Rule (SOS-STD-CALL). Instead, we assume that $P^{[t]}$ also renames functions and that R contains an appropriately t -renamed function copy. Later, it will be convenient to access the witnesses of τ :

$$\text{ctrl}(\tau) := \{ pc \mid (pc_{init}, \epsilon) \Rightarrow^* (pc, \tau) \} .$$

Thread-Modular Analysis

Proving a data structure correct for an arbitrary number of client threads requires a thread-modular analysis [Berdine et al. 2008; Flanagan and Qadeer 2003b; Jones 1983; Owicki and Gries 1976]. Such an analysis abstracts a system state into so-called *views*, partial configurations reflecting a single thread’s perception of the system state. A view includes a thread’s control location and, in the case of shared-memory programs, the memory reachable from the shared and thread-local variables. An analysis then saturates a set V of reachable views. This is done by computing the least solution to the recursive equation

$$V = V \cup seq(V) \cup int(V) .$$

Function *seq* computes a sequential step, the views obtained from letting each thread execute an action on its own views. Function *int* accounts for interference among threads. It updates the memory of views by actions from other threads. We follow the analysis proposed by Abdulla et al. [2013, 2017]. There, *int* is computed by combining two views, letting one thread perform an action, and projecting the result to the other thread. More precisely, computing $int(V)$ requires for every pair of views $v_1, v_2 \in V$ to (i) compute a combined view ω of v_1 and v_2 , (ii) perform for ω a sequential step for the thread of v_2 , and (iii) project the result of the sequential step to the perception of the thread from v_1 . This process is required only for views v_1 and v_2 that *match*, i.e., agree on the shared memory both views have in common. Otherwise, the views are guaranteed to reflect different system states so that interference is not needed for an exhaustive state space exploration.

To check for linearizability, we assume that the program under scrutiny is annotated with linearization points. Whether or not the sequence of emitted linearization is legal, we verify with the ADTs from Abdulla et al. [2013, 2017]. They capture sequential stack, queue, and set ADTs in form of automata. The state of this specification-checking automaton is stored in the views. If they signal a specification violation by reaching a final state, verification fails.

To arrive at views of finite size, we apply a memory abstract. The abstraction we use tracks reachability predicates among the objects referenced by the local and shared pointer variables. The reachability predicates encode equality, reachability in one step, reachability in two or more steps, and unreachability. Here, a step refers to following an object’s next field. We do not go into the details of the memory abstraction as it is orthogonal to the results presented in the present thesis. For more details, we refer the reader to [Abdulla et al. 2013, 2017].

We stress that the analysis by Abdulla et al. [2013, 2017]—at the time of writing—is the most promising for fully automatically verifying non-blocking data structures with manual memory reclamation.

Part II

Contributions

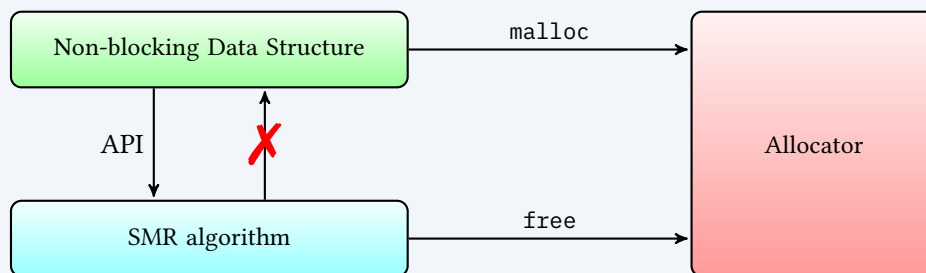
Compositional Verification

Verification of non-blocking data structures with manual memory management via an SMR algorithm is prohibitive with state-of-the-art techniques. The reason for this is the complexity that is added to the verification task by SMR implementations. As seen in Chapter 2, data structures and SMR implementations are equally complex.

To allow for verification nevertheless, we exploit the design of data structures and SMR algorithms. Typically, data structures use SMR algorithms through a well-defined API which does not expose the implementation details of the SMR algorithm. The resulting system design is depicted in Figure 5.1. This encapsulation suggests a verification approach for data structures where we replace the SMR implementation with a simpler one. For a sound approach, we have to ensure that the replacement over-approximates the behaviors of the original SMR implementation. This way, we can separate the verification of the data structure from the SMR implementation. More specifically, we (i) introduce a means for specifying SMR implementations, then (ii) verify the SMR implementation R against its specification, and (iii) verify the data structure P relative to the SMR specification rather than the SMR implementation. If both verification tasks succeed, then the data structure using the SMR implementation, $P(R)$, is correct.

Towards our result, we first introduce SMR automata for specifying SMR algorithms. Then, we discuss the two new verification tasks and show that they imply the desired correctness.

Figure 5.1: Typical system design and the interaction among components. Non-blocking data structures perform their reclamation through an SMR algorithm. The SMR algorithm does not influence the data structure directly, only indirectly through the Allocator.



5.1 SMR Automata

An SMR automaton \mathcal{O} consists of a finite set of locations, a finite set of variables, and a finite set of transitions. There is a dedicated initial location and some 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 $\bar{r} = r_1, \dots, r_n$. The guard is a Boolean formula over equalities of variables and the parameters \bar{r} . An SMR automaton state s is a tuple (l, φ) where l is a location and φ maps variables to values. Such a state is initial if l is initial, and similarly accepting if l is accepting. Then, $(l, \varphi) \xrightarrow{f(\bar{v})} (l', \varphi)$ is an SMR automaton step, if $l \xrightarrow{f(\bar{r}), g} l'$ is a transition and $\varphi(g[\bar{r} \mapsto \bar{v}])$ evaluates to true. With $\varphi(g[\bar{r} \mapsto \bar{v}])$ we mean g where the formal parameters \bar{r} are replaced with the actual values \bar{v} and where the variables are replaced by their φ -mapped values. Initially, the valuation φ is chosen non-deterministically; it is not changed by steps.

A *history* $h = f_1(\bar{v}_1) \dots f_n(\bar{v}_n)$ is a sequence of events. If there are steps $s \xrightarrow{f_1(\bar{v}_1)} \dots \xrightarrow{f_n(\bar{v}_n)} s'$, then we write $s \xrightarrow{h} s'$. If s' is accepting, we say that h is accepted by s . We use SMR automata to characterize *bad behavior*. So we say h is in the specification of s , denoted by $h \in \mathcal{S}(s)$, if it is not accepted by s . Then, the specification of \mathcal{O} , denoted by $\mathcal{S}(\mathcal{O})$, is the set of histories that are not accepted by any initial state of \mathcal{O} . Formally, we define:

$$\mathcal{S}(s) := \{ h \mid \forall s'. s \xrightarrow{h} s' \implies s' \text{ not final} \} \quad \text{and} \quad \mathcal{S}(\mathcal{O}) := \bigcap \{ \mathcal{S}(s) \mid s \text{ initial} \}.$$

The cross-product $\mathcal{O}_1 \times \mathcal{O}_2$ denotes an SMR automaton with $\mathcal{S}(\mathcal{O}_1 \times \mathcal{O}_2) = \mathcal{S}(\mathcal{O}_1) \cap \mathcal{S}(\mathcal{O}_2)$.

To simplify our development, we assume that SMR automata are complete and deterministic in the sense that each state has a unique post state for all possible events.

Assumption 5.2 (Well-formedness). SMR automata \mathcal{O} satisfy the following: (i) for all s_1 and all h there is s_2 with $s_1 \xrightarrow{h} s_2$, and (ii) if $s_1 \xrightarrow{h} s_2$ and $s_1 \xrightarrow{h} s_3$, then $s_2 = s_3$.

Hereafter, it will be useful to check specification inclusions of the form $\mathcal{S}(s) \subseteq \mathcal{S}(s')$ for SMR automata \mathcal{O} . To accomplish this efficiently, we compute a simulation relation [Milner 1971] $\leq_{\mathcal{O}}$ among the locations of \mathcal{O} which entails the desired inclusion. Technically, $\leq_{\mathcal{O}}$ is the largest relation such that for all locations $l_1 \leq_{\mathcal{O}} l_2$ the following conditions are met: (i) if l_1 is not accepting, then l_2 is not accepting, and (ii) for all transitions $l_1 \xrightarrow{evt, g} l'_1$ and $l_2 \xrightarrow{evt, g'} l'_2$ with $g \wedge g'$ satisfiable we have $l'_1 \leq_{\mathcal{O}} l'_2$. Relation $\leq_{\mathcal{O}}$ can be computed by a greatest fixed point in the standard way [Baier and Katoen 2008, Section 7.6; Cleaveland and Steffen 1991; Henzinger et al. 1995]. As for ordinary finite-state automata, the simulation relation is stronger than the specification inclusion [Baier and Katoen 2008, Section 7.4]. However, we found the simulation easier to implement and sufficient in practice.

Proposition 5.3. If $l \leq_{\mathcal{O}} l'$, then $\mathcal{S}((l, \varphi)) \subseteq \mathcal{S}((l', \varphi))$ for all φ .

5.2 SMR Specifications

To use SMR automata for specifying SMR algorithms, we have to instantiate appropriately the histories they observe. Our instantiation crucially relies on the fact that programmers of non-blocking data structures rely solely on simple temporal properties that SMR algorithms implement [Gotsman et al. 2013]. These properties are typically incognizant of the actual SMR implementation. Instead, they allow reasoning about the implementation's behavior based on the temporal order of function invocations and responses. With respect to our programming model, `in` and `re` actions provide the necessary means to deduce from the data structure computation how the SMR implementation behaves.

We instantiate SMR automata for specifying SMR algorithms as follows. Let $func_1, \dots, func_n$ be the API functions offered by the SMR algorithm. The event types are (i) $in:func_1, \dots, in:func_n$, (ii) $re:func_1, \dots, re:func_n$, and (iii) `free`. The parameters to the events depend on the type of the event. They are (i) the executing thread and the parameters to the call for type $in:func_i$, (ii) the executing thread for type $re:func_i$, and (iii) the parameters to the call for type `free`. For simplicity, we consider the hazard pointer index passed to `protect` and `unprotect` a part of the name/type. That is, we write $protect_k(p)$ and $unprotect_k()$ instead of $protect(p, k)$ and $unprotect(k)$, respectively.

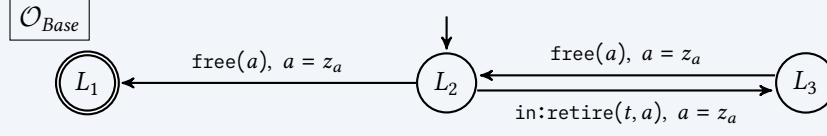
To give an example, consider the EBR specification $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$ from Figure 5.4. It consists of two SMR automata. First, \mathcal{O}_{EBR} implements the temporal property that a retired address must not be freed until all threads end their ongoing non-quiescent phase, i.e., until they invoke `enterQ` for the first time after the `retire`. Second, \mathcal{O}_{Base} specifies that no address must be freed that has not been retired yet. Further, \mathcal{O}_{Base} specifies that no address must be freed twice unless the address is retired in-between the frees. For the automaton to properly restrict the frees in a program, the program should not perform *double retires*, that is, not retire an address again before it is freed. The point is that SMR algorithms typically misbehave after a double retire (perform double frees), which is not reflected in \mathcal{O}_{Base} (it does not allow for double frees after a double retire). Our verification techniques will establish this property. To make double retires precise, let $retired_\tau \subseteq Adr$ be the addresses that have been retired in τ but not freed since.

Definition 5.5 (Double Retire). Computation $\tau.\langle t, in:retire(p), up \rangle$ performs a double retire of address a if we have $a \in retired_\tau$ and $a = m_\tau(p)$.

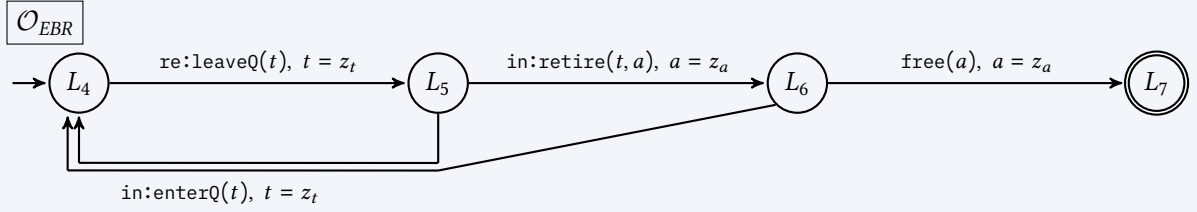
The specification of HP is a bit more involved. For two hazard pointers per thread, a first attempt is the automaton $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$. Here, \mathcal{O}_{HP}^k implements the HP-specific property that no address must be freed if it has been protected continuously since before being retired. However, the specification treats hazard pointers individually and thus misses transfers of protections among multiple hazard pointers (cf. Section 2.3.3). To briefly reiterate the issue, if a thread protects an address first with its 0-th hazard pointer, later with its 1-st hazard pointer, and resets the 0-th hazard pointer, then the address is continuously protected by *some* hazard pointer but not by a *single* hazard pointer. The above specification would allow for a spurious free. To come up with an appropriate specification that resolves the issue, we have to track all hazard pointers of a thread simultaneously. Intuitively, we have to compute a more involved cross product than $\mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$ to account for transferring protections among hazard pointers. The resulting SMR automaton is $\mathcal{O}_{HP}^{0,1}$. Due to the size of the automaton (it consists of 16 states) we

Figure 5.4: SMR automata specifications for EBR resp. HP: $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$ resp. $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$. The automata are negative specifications, they accept those histories that violate the desired property. Two automata-local variables, z_t resp. z_a , are used to capture a thread resp. an address. For better legibility we omit self-loops for every location and every event that is missing an outgoing transition from that location.

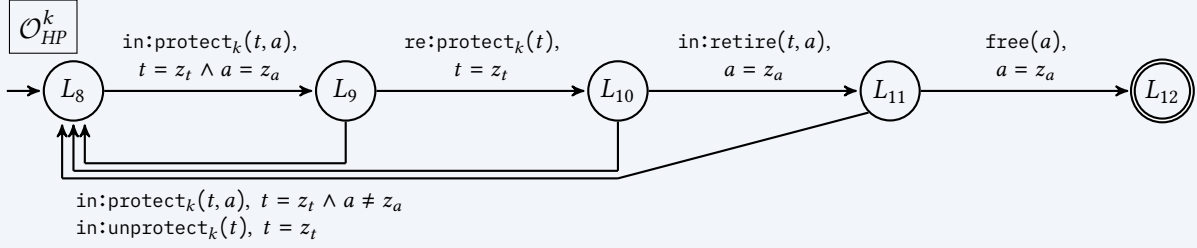
(a) SMR automaton \mathcal{O}_{Base} specifies that address z_a may be freed only if it has been retired and not been freed since.



(b) SMR automaton \mathcal{O}_{EBR} specifies how EBR defers frees: a retired address z_a may not be freed if it has been retired during a non-quiescent phase of thread z_t .



(c) SMR automaton \mathcal{O}_{HP}^k specifies how HP defers frees: a retired address z_a may not be freed if it has been protected continuously by the k -th hazard pointer of thread z_t since before being retired.



do not present it here, it can be found in Appendix A.2. It is worth noting that $\mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$ is useful nevertheless: it is smaller than $\mathcal{O}_{HP}^{0,1}$ and might thus speed up verification for data structures that do not transfer hazard pointers.

The only SMR technique we are aware of that does not fit into the SMR automaton framework out of the box is FL. Recall that FL does not free the memory it manages. Instead, it simply stores the addresses that have been retired and redistributes them for threads to reuse. This resembles a direct influence of the SMR algorithm on the client data structure, which cannot be encoded with SMR automata as they do not support return values. In order to support FL, we adopt the practice of Abdulla et al. [2013, 2017]. We assume that (i) retired addresses are freed immediately, and that (ii) freed addresses may be accessed safely. With those assumptions, \mathcal{O}_{Base} specifies FL.

While every SMR implementation has its own SMR automaton, the practically relevant SMR automata are products of \mathcal{O}_{Base} with further SMR automata. Our development relies on this.

Assumption 5.6. SMR automata \mathcal{O} are of the form $\mathcal{O} = \mathcal{O}_{Base} \times \mathcal{O}_{SMR}$ for some \mathcal{O}_{SMR} .

With an SMR specification in form of an SMR automaton \mathcal{O} at hand, our task is to check whether or not a given SMR implementation R satisfies this specification. We do this by converting a computation τ of R into its induced history $\mathcal{H}(\tau)$ and check inclusion in $\mathcal{S}(\mathcal{O})$. The induced history $\mathcal{H}(\tau)$ is a projection of τ to `in`, `re`, and `free` commands. The projection replaces formal parameters with actual values.

Definition 5.7 (Induced Histories). The history induced by a computation τ , denoted by $\mathcal{H}(\tau)$, is:

$$\begin{aligned}\mathcal{H}(\epsilon) &= \epsilon \\ \mathcal{H}(\tau.\langle t, \text{free}(p), up \rangle) &= \mathcal{H}(\tau).\text{free}(m_\tau(p)) \\ \mathcal{H}(\tau.\langle t, \text{in:func}(\bar{r}), up \rangle) &= \mathcal{H}(\tau).\text{in:func}(t, m_\tau(\bar{r})) \\ \mathcal{H}(\tau.\langle t, \text{re:func}, up \rangle) &= \mathcal{H}(\tau).\text{re:func}(t) \\ \mathcal{H}(\tau.\text{act}) &= \mathcal{H}(\tau) \quad \text{otherwise.}\end{aligned}$$

Then, τ satisfies \mathcal{O} if $\mathcal{H}(\tau) \in \mathcal{S}(\mathcal{O})$. SMR implementation R satisfies \mathcal{O} if every possible usage of R produces a computation satisfying \mathcal{O} . To generate those computations, we use a most general client (MGC) for R which concurrently executes arbitrary sequences of SMR functions.

Definition 5.8 (SMR Correctness). An SMR implementation R is correct wrt. a specification \mathcal{O} , written $R \models \mathcal{O}$, if for all $\tau \in \llbracket \text{MGC}(R) \rrbracket_{\text{Addr}}^{\text{Addr}}$ we have $\mathcal{H}(\tau) \in \mathcal{S}(\mathcal{O})$.

From the above definition follows the first new verification task: prove that the SMR implementation R cannot possibly violate the specification \mathcal{O} . Intuitively, this boils down to a reachability analysis of accepting states in the cross-product of $\text{MGC}(R)$ and \mathcal{O} . Since we can understand R as a non-blocking data structure itself, this task is similar to our next one, namely verifying the data structure relative to \mathcal{O} . We focus on this second task because it is harder than the first one. The reason for this lies in that SMR implementations typically do not reclaim the memory they use. This holds true even if the SMR implementation supports dynamic thread joining and parting. The absence of reclamation greatly simplifies the analysis. Our experiments confirm this intuition: in Chapter 7 we automatically verify the EBR and HP implementations from Chapter 2 against the corresponding SMR automaton specifications presented above.

5.3 Verification Relative to SMR Automata

The next task is to verify the data structure $P(R)$ avoiding the complexity of R . We have already established the correctness of R wrt. a specification \mathcal{O} . Intuitively, we now replace implementation R with its specification \mathcal{O} . Because \mathcal{O} is an SMR automaton, and not program code like R , we cannot just execute \mathcal{O} in place of R . Instead, we remove the SMR implementation from $P(R)$. The result is $P(\epsilon)$ the computations of which correspond to the ones of $P(R)$ with the SMR implementation-specific actions between `in` and `re` being removed. To account for the frees that R executes, we introduce *environment steps*. We non-deterministically check for every address a whether or not \mathcal{O} allows freeing it. If so, we free the address. Formally, the new SMR semantics is $\mathcal{O} \llbracket P \rrbracket_X^Y$ and

corresponds to the standard semantics $\llbracket P(\epsilon) \rrbracket_X^Y$ as defined in Section 3.3 except for an updated rule for frees from the environment.

(Free) If $\tau \in \llbracket P(\epsilon) \rrbracket_X^Y$ and $a \in \text{Adr}$ can be freed, i.e., $a \in X$ and $\mathcal{H}(\tau).\text{free}(a) \in \mathcal{S}(\mathcal{O})$, then we have $\tau.\text{act} \in \mathcal{O} \llbracket P \rrbracket_X^Y$ with $\text{act} = \langle t, \text{free}(a), \emptyset \rangle$.

The new semantics considers \mathcal{O} only when freeing memory, all other rules remain unaffected. With this definition, $\mathcal{O} \llbracket P \rrbracket_X^Y$ performs more frees than $\llbracket P(R) \rrbracket_X^Y$, provided $R \models \mathcal{O}$. In analogy to $P(R)$, we write $P(\mathcal{O})$ to refer to $P(\epsilon)$ relative to \mathcal{O} . Figure 5.9 gives the full SMR semantics.

With the semantics of data structures relative to an SMR specification rather than an SMR implementation set up, we can turn to the main result of this section. It states that the correctness of R wrt. \mathcal{O} and the correctness of $P(\epsilon)$ under \mathcal{O} entail the correctness of the original program $P(R)$. Here, we focus on the verification of safety properties. It is known that this reduces to control location reachability [Vardi 1987??].¹ So we can assume that there are dedicated *bad* control locations in P the unreachability of which is equivalent to the correctness of $P(R)$. If the bad control locations are unreachable in a computation τ , we write $\text{good}(\tau)$; the predicate naturally extends to sets. For the overall result to hold, we require that the interaction between P and R follows the one depicted in Figure 5.1 and seen on practical examples in Chapter 2. That is, frees are the only influence that R has on P . In particular, this means that R does not modify the memory accessed by P . We found this restriction satisfied by many SMR algorithms from the literature. We believe that our development can be generalized to incorporate memory modifications performed by the SMR algorithm. A proper investigation, however, is beyond the scope of this thesis.

Theorem 5.10 (Compositionality). If $R \models \mathcal{O}$ and $\text{good}(\mathcal{O} \llbracket P \rrbracket_{\text{Adr}}^{\text{Adr}})$, then $\text{good}(\llbracket P(R) \rrbracket_{\text{Adr}}^{\text{Adr}})$.

Compositionality is a powerful tool for verification. It allows us to verify the data structure and the SMR implementation independently of each other. Although this simplifies the verification, reasoning about non-blocking programs operating on a shared memory remains hard. In Chapters 6 to 8 we build upon the above result and propose sound verification techniques for $\mathcal{O} \llbracket P \rrbracket_{\text{Adr}}^{\text{Adr}}$ that need not consider the full semantics but subsets thereof. Reductions to simpler semantics are imperative as compositionality alone makes verification hardly tractable with state-of-the-art techniques, as we will see in Section 6.3.

Besides verifying the actual correctness property of P , i.e., establishing $\text{good}(\mathcal{O} \llbracket P \rrbracket_{\text{Adr}}^{\text{Adr}})$, we will also establish the absence of double retires, as required for a reasonable application of $\mathcal{O}_{\text{Base}}$. As expected, compositionality allows us to rely on the simpler SMR semantics.

Theorem 5.11. If $R \models \mathcal{O}$, then $\llbracket P(R) \rrbracket_{\text{Adr}}^{\text{Adr}}$ is free from double retires if $\mathcal{O} \llbracket P \rrbracket_{\text{Adr}}^{\text{Adr}}$ is.

¹ Bouajjani et al. [2015b] show that linearizability reduces to control location reachability as well.

Figure 5.9: Semantics of programs relative to an SMR automaton, $\mathcal{O}[[P]]_X^Y$.

(a) SMR semantics of commands. We write $act \in \overline{Act}(\tau, t, com)$ if one of the following rules applies.

(Skip) If $act = \langle t, \text{skip}, \emptyset \rangle$.

(Assign1) If $act = \langle t, p.\text{next} := q, [a.\text{next} \mapsto b] \rangle$ then $m_\tau(p) = a$ and $m_\tau(q) = b$.

(Assign2) If $act = \langle t, p := q, [p \mapsto m_\tau(q)] \rangle$.

(Assign3) If $act = \langle t, p := q.\text{next}, [p \mapsto m_\tau(a.\text{next})] \rangle$ with $m_\tau(q) = a \in \text{Adr}$.

(Assign4) If $act = \langle t, u := \text{op}(u'_1, \dots, u'_n), [u \mapsto d] \rangle$ with $d = \text{op}(m_\tau(u'_1), \dots, m_\tau(u'_n))$.

(Assign5) If $act = \langle t, p.\text{data} := u, [a.\text{data} \mapsto m_\tau(u)] \rangle$ with $m_\tau(p) = a \in \text{Adr}$.

(Assign6) If $act = \langle t, u := q.\text{data}, [u \mapsto m_\tau(a.\text{data})] \rangle$ with $m_\tau(q) = a \in \text{Adr}$.

(Assume) If $act = \langle t, \text{assume } exp \triangleq exp', \emptyset \rangle$ then $m_\tau(exp) \triangleq m_\tau(exp')$.

(Malloc) If $act = \langle t, p := \text{malloc}, [p \mapsto a, a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d] \rangle$ then address a is allocatable, that is, $a \in \text{fresh}_\tau$ or $a \in \text{freed}_\tau \cap Y$.

(Free) If $act = \langle t, \text{free}(a), \emptyset \rangle$ then $a \in \text{Adr} \cap X$ and $\mathcal{H}(\tau).\text{free}(a) \in \mathcal{S}(\mathcal{O})$.

(Call) If $act = \langle t, \text{in:func}(\bar{r}), \emptyset \rangle$, then $m_\tau(r) \in \text{Adr} \uplus \text{Dom}$ for every r in \bar{r} .

(Return) If $act = \langle t, \text{re:func}, \emptyset \rangle$.

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

(Env) If $act = \langle \perp, \text{env}(a), [a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d] \rangle$ then $a \in \text{fresh}_\tau \cup \text{freed}_\tau$.

(b) SMR semantics of programs. We define a SOS transition relation \rightarrow relative to a control-flow relation \xrightarrow{com} such that $\tau \in \mathcal{O}[[P]]_X^Y$ iff $\text{ctrl}(\tau) \neq \emptyset$ with $\text{ctrl}(\tau) := \{ pc \mid (\lambda t. P^{[t]}, \epsilon) \rightarrow^* (pc, \tau) \}$.

(SOS-COM)	(SOS-SEQ1)	(SOS-SEQ2)	(SOS-CHOICE)	(SOS-LOOP1)
$\frac{}{com \xrightarrow{com} \text{skip}}$	$\frac{}{\text{skip}; st \xrightarrow{\text{skip}} st}$	$\frac{st_1 \xrightarrow{com} st'_1}{st_1; st_2 \xrightarrow{com} st'_1; st_2}$	$\frac{i \in \{1, 2\}}{st_1 \oplus st_2 \xrightarrow{\text{skip}} st_i}$	$\frac{}{st^* \xrightarrow{\text{skip}} \text{skip}}$
(SOS-LOOP2)	(SOS-ATOMIC3)	(SOS-ATOMIC1)		
$\frac{}{st^* \xrightarrow{\text{skip}} st; st^*}$	$\frac{}{\text{inatomic skip} \xrightarrow{\text{endAtomic}} \text{skip}}$	$\frac{}{\text{beginAtomic}; st; \text{endAtomic} \xrightarrow{\text{beginAtomic}} \text{inatomic } st}$		
(SOS-ATOMIC2)	(SOS-ENV)	(SOS-FREE)		
$\frac{st \xrightarrow{com} st'}{\text{inatomic } st \xrightarrow{com} \text{inatomic } st'}$	$\frac{act \in \overline{Act}(\tau, \perp, \text{env}(a))}{(pc, \tau) \rightarrow (pc, \tau.\text{act})}$	$\frac{act \in \overline{Act}(\tau, \perp, \text{free}(a))}{(pc, \tau) \rightarrow (pc, \tau.\text{act})}$		
(SOS-PAR)				
$\frac{st \xrightarrow{com} st' \quad act \in \overline{Act}(\tau, t, com) \quad \nexists t'. t \neq t' \wedge \text{locked}(pc(t'))}{(pc[t \mapsto st], \tau) \rightarrow (pc[t \mapsto st'], \tau.\text{act})}$				

Ownership and Reclamation

Ownership reasoning is a well-known technique that is vital for thread-modular analyses: it brings the necessary precision required for successful verification. Traditionally, it is assumed that references to owned memory exist only within the owning thread. While this strong exclusivity assumption is guaranteed to hold under garbage collection, it is unsound when memory is reclaimed and reused. The reason for this are dangling pointers. They may observe how another thread reclaims and reallocates, thus owns, some part of the memory. To overcome this problem, we introduce a weaker notion of ownership. We relax the traditional exclusivity assumption for dangling pointers, and for dangling pointers only. The resulting approach is sound and makes thread-modular analyses sufficiently precise. Moreover, it comes with a relatively small overhead compared to existing solutions.

The remainder of the chapter is structured as follows. Section 6.1 demonstrates both the need for and the unsoundness of ownership reasoning for manual memory management. Section 6.2 introduces a novel notion of weak ownership and shows how it can be used to increase the precision of thread-modular analyses. Section 6.3 evaluates our approach and compares it to existing ones.

6.1 Reclamation breaks Ownership

Thread-modular analyses [Berdine et al. 2008; Jones 1983; Owicki and Gries 1976] verify each thread individually. On the one hand, this yields an efficient analysis for programs with a fixed number of threads as it avoids an explicit cross-product of all threads. On the other hand, it makes verification for an arbitrary number of threads possible. The downside of thread-modularity is its imprecision in computing thread interferences. Since threads are verified individually, the relation among thread-local information gets lost. We discuss this problem and why its common solution does not apply for manual memory management.

As we have seen in Chapter 4, thread-modular analyses abstract program configurations into sets of views which capture a thread's perception of the configuration. To compute the effect that an interfering thread has on a victim thread, the views for the two threads are combined, the interfering thread takes a step in the resulting view, which is then projected to the victim thread. Combining two views is more problematic than one might think. As already noted, views abstract away the relation among the interfering and victim threads. For an analysis to be sound, it has to consider all possible relations among those two threads. This introduces imprecision and may ultimately lead to false alarms. We illustrate the problem on an example.

Example 6.1. Consider the views from Figure 6.2 which arise during a thread-modular analysis of Michael&Scott’s queue. The threads captured by views v_1 and v_2 from Figure 6.2a are t_1 and t_2 , respectively. Thread t_1 is executing enqueue. It has already allocated a new node b , referenced by its local pointer variable $t_1:\text{node}$, and is about to execute the CAS from Line 303 in order to insert the new node after Tail. Thread t_2 is executing dequeue. It has removed node c , referenced by $t_2:\text{head}$. Its next step is to retire c .¹

Let us consider the interference t_1 is exposed to due to the actions of t_2 . The goal is to compute a new view for t_1 which captures the effect of t_2 performing the insertion. To that end, we combine the two views from Figure 6.2a. The result is given in Figure 6.2b. View v_3 is the expected one: $t_1:\text{node}$ points to b and $t_2:\text{head}$ points to c with $b \neq c$ —the threads hold pointers to distinct nodes. In v_4 , however, both threads alias the exact same node. Although peculiar, we have to consider view v_4 as well to guarantee soundness of the overall analysis, that is, guarantee that all possible views are explored. Indeed, just from inspecting v_1 and v_2 we cannot conclude that the memory layout of v_4 is spurious in the sense that it does not occur in any execution of Michael&Scott’s queue. Unfortunately, the spurious view will lead to a false alarm.

To see why view v_4 is problematic, we continue to compute the inference. To that end, we let t_2 execute its next command in v_4 . The result is view v'_4 from Figure 6.2c. In v'_4 , node b has been retired. Here, we assume that the retirement is followed immediately by a `free(b)`. Next, we project away thread t_2 from v'_4 and let t_1 execute its next command. In the resulting view, v''_4 from Figure 6.2c, the deleted node b has become the new Tail, breaking the shape invariant of the queue. Subsequent enqueue operations can now reallocate b and update it. The updates lead to unintended updates of the overall queue, changing the queue’s content or losing elements if the next field of b is modified. This constitutes a linearizability violation, verification fails. ■

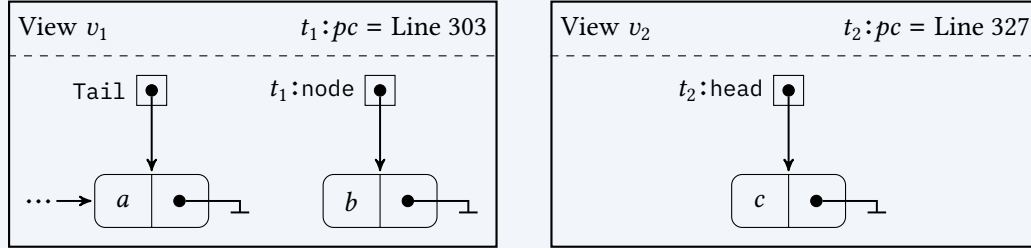
A well-known and common technique to avoid such spurious views during thread-modular analyses is *ownership* reasoning [Castegren and Wrigstad 2017; Dietl and Müller 2013; Gotsman et al. 2007; O’Hearn 2004; Vafeiadis and Parkinson 2007]. The allocation of a new node grants the allocating thread ownership over the new node. Ownership is removed as soon as the new node is published, that is, made accessible to other threads. Typically, this happens when a pointer to a node is written to a shared pointer variable or to a pointer field of another node that is reachable from the shared variables. (We refrain from a formal definition of ownership at this point.) Then, we exploit ownership to avoid spurious views and increase the precision of thread-modular analyses. To that end, we extend views to track ownership information and prevent combinations of views where an owned node is referenced by another non-owning thread. That is, we ensure that the access exclusivity granted by ownership is respected. For the above example, this means that thread t_2 cannot have a pointer to b . Hence, we can rule out view v_4 as a combination of v_1 and v_2 because $b = c$ is guaranteed to be no longer possible.

While ownership reasoning is elegantly simple and yet effective, we cannot use it in our setting. The above approach is sound only under garbage collection, when nodes are not reclaimed, but unsound otherwise. We demonstrate this with an example. Thereafter, in Section 6.2, we introduce a new variant of ownership that applies to our setting.

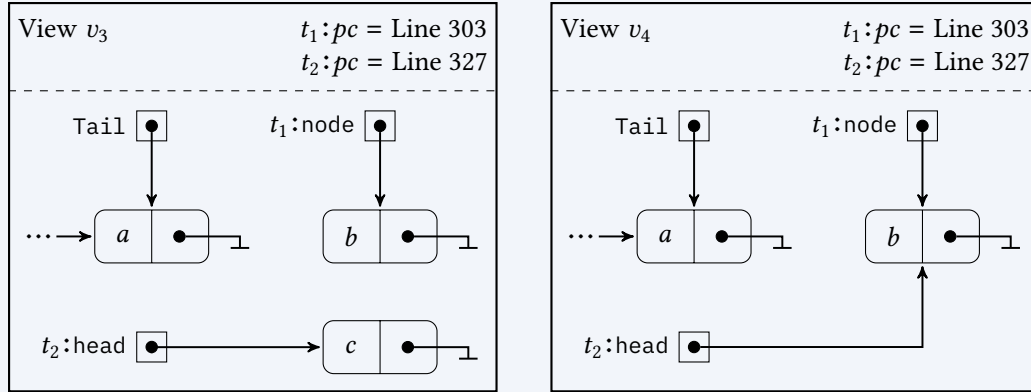
¹ The attentive reader of Chapter 2 might observe that, unlike presented in Figure 6.2 here, the next field `c.next` of the removed but not yet retired node c is never NULL in Michael&Scott’s queue. To obtain such a view where `c.next` is NULL, we require a preceding interference step which suffers from the same imprecision as the interference step presented here. For simplicity, we stick with `c.next` being NULL in this example.

Figure 6.2: Views encountered during a thread-modular analysis of Michael&Scott’s queue. Imprecision in interference steps leads to spurious verification failure.

(a) Two views the interference among which is computed. View v_1 captures thread t_1 which has allocated a new node b and is about to append it to the Tail of the queue via the CAS from Line 303. View v_2 captures thread t_2 which has removed node c from the queue and is about to retire it, Line 327.



(b) Possible combinations of views v_1 and v_2 . Judging from the view abstraction alone, a vanilla thread-modular analysis cannot know whether nodes b and c coincide in the actual program configuration the views abstract from. Interference has to consider both v_3 and v_4 , although v_4 spurious.



(c) Continuing the interference computation for v_4 , we let thread t_2 take a step and retire b , Line 327. The result is v_4' where the retirement of b has immediately freed it (marked with \dagger). Next, we project away t_2 and let t_1 execute Line 303. The result is v_4'' where the freed node b has been inserted into the queue. Subsequent reallocations of b may thus change the queue’s content unknowingly, leading to verification failure. Note that the verification failure is spurious since it is a result of the spurious v_4 .

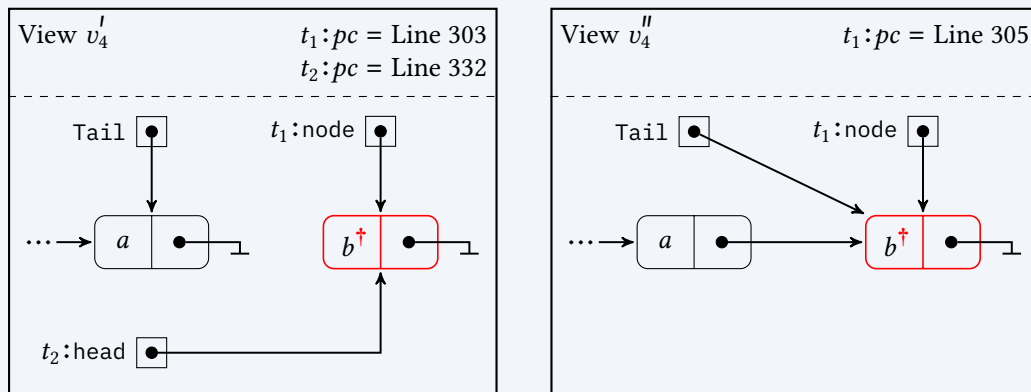
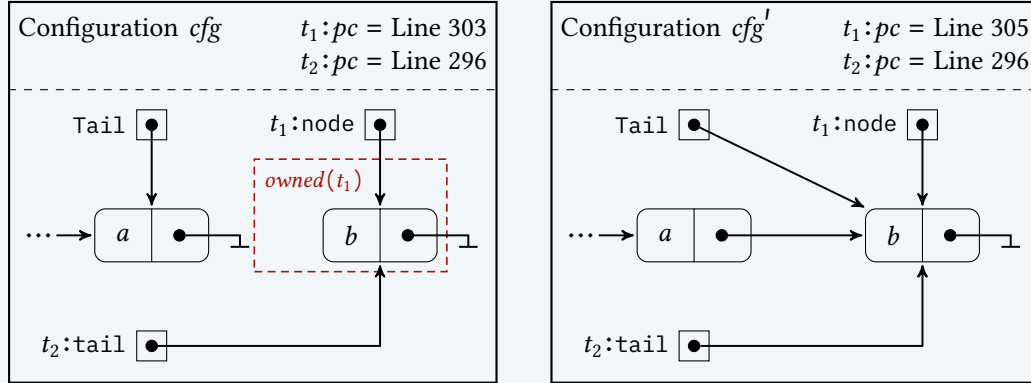
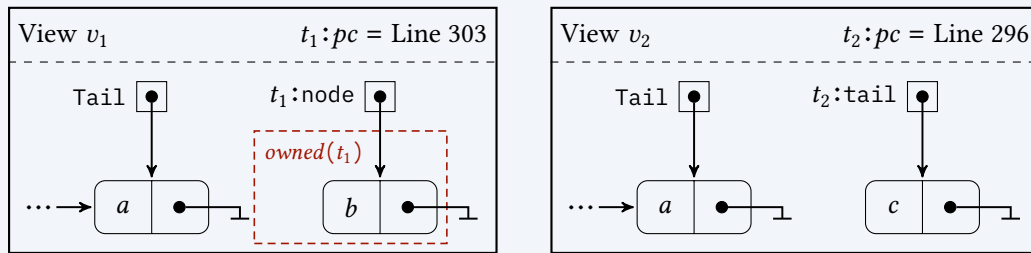


Figure 6.4: Reallocation scenario from Michael&Scott’s queue with hazard pointers where traditional ownership reasoning, as done under garbage collection, is unsound.

(a) Program configurations without view abstraction. In cfg , thread t_1 owns node b while t_2 holds a dangling pointer to it. The scenario arises if b is reclaimed and subsequently reallocated in-between t_2 acquiring and protecting pointer $t_2:\text{tail}$. Next, t_1 inserts b into the queue. The result is cfg' .



(b) View abstraction for cfg gives views v_1 and v_2 . Applying traditional ownership reasoning prevents a combined view where nodes b and c coincide, because b is owned. Hence, configuration cfg' is not guaranteed to be covered by any view. The analysis is unsound.



Example 6.3. To see why traditional ownership reasoning breaks when memory is reclaimed and reused, consider the configurations of Michael&Scott’s queue with hazard pointers depicted in Figure 6.4a. Configuration cfg corresponds to the scenario where node b used to be the Tail of the queue, however, it has subsequently been removed from the queue, reclaimed, and reallocated. The reallocating thread and owner of b is t_1 . Thread t_2 started its operation while b was still the Tail and acquired a pointer to it, $t_2:\text{tail}$. Node b was removed and reallocated before t_2 could protect it. Hence, $t_2:\text{tail}$ is a dangling pointer to the now t_1 -owned b . The view abstraction of cfg is the expected one: views v_1 and v_2 from Figure 6.4b. To make explicit that we lose any relation among threads in the abstraction, we renamed node b to c in v_2 (in practice, memory abstractions are unlikely to maintain the addresses explicitly [Chang et al. 2020]).

If we let thread t_1 continue its execution in cfg , it appends b to the end of the queue and then swings Tail to the newly added node. The result is cfg' from Figure 6.4a. For an analysis to be sound, interference has to produce from v_1 and v_2 a new view for t_2 that captures the effect of t_1 ’s actions. The first step of interference is to combine v_1 and v_2 . Intuitively, we expect the combined view to correspond to cfg . The fact that t_1 owns b , however, makes traditional ownership reasoning ignore the relevant case $b = c$. That is, v_1 and v_2 do not produce cfg although they

were obtained from it. Consequently, it is not guaranteed that a view for t_2 is explored which reflects $cfg^!$. This compromises soundness.

It is worth pointing out that under GC the same problem does not arise. Node b would have not been reclaimed due to $t_2:\text{tail}$ pointing to it. ■

To overcome the problem of an unsound analysis, we introduce a variant of ownership that allows for both soundness and precision in the presence of memory reclamation and reuse.

6.2 Regaining Ownership

The previous section demonstrated the dilemma of thread-modular analyses: interference without ownership is too imprecise for successful verification, however, exploiting ownership makes the analysis unsound. We propose a novel notion of ownership to overcome this problem. The key observation is the following: traditional ownership reasoning breaks soundness because of dangling pointers. When memory is reallocated, all preexisting pointers to the newly allocated node are dangling. We suggest to keep track of this fact and allow dangling pointers to reference nodes owned by other threads when combining views for interference. All remaining, non-dangling pointers are treated in the traditional way: they are prevented from referencing nodes owned by other threads during interference.

To make precise which pointers in a computation are dangling, we introduce the notion of *validity*. That is, we define a set of valid pointers. The dangling pointers are then the complement of the valid pointers. We take this detour since we found it easier to formalize the valid pointers.

Definition 6.5 (Valid Expressions). The valid expressions in τ , $\text{valid}_\tau \subseteq PExp$, are:

$$\begin{aligned}
& \text{valid}_\epsilon := PVar \\
& \text{valid}_{\tau, \langle t, p := q, up \rangle} := \text{valid}_\tau \cup \{p\} & \text{if } q \in \text{valid}_\tau \\
& \text{valid}_{\tau, \langle t, p := q, up \rangle} := \text{valid}_\tau \setminus \{p\} & \text{if } q \notin \text{valid}_\tau \\
& \text{valid}_{\tau, \langle t, p.\text{next} := q, up \rangle} := \text{valid}_\tau \cup \{a.\text{next}\} & \text{if } m_\tau(p) = a \in \text{Adr} \wedge q \in \text{valid}_\tau \\
& \text{valid}_{\tau, \langle t, p.\text{next} := q, up \rangle} := \text{valid}_\tau \setminus \{a.\text{next}\} & \text{if } m_\tau(p) = a \in \text{Adr} \wedge q \notin \text{valid}_\tau \\
& \text{valid}_{\tau, \langle t, p := q.\text{next}, up \rangle} := \text{valid}_\tau \cup \{p\} & \text{if } q \in \text{valid}_\tau \wedge m_\tau(q).\text{next} \in \text{valid}_\tau \\
& \text{valid}_{\tau, \langle t, p := q.\text{next}, up \rangle} := \text{valid}_\tau \setminus \{p\} & \text{if } q \notin \text{valid}_\tau \vee m_\tau(q).\text{next} \notin \text{valid}_\tau \\
& \text{valid}_{\tau, \langle t, \text{free}(a), up \rangle} := \text{valid}_\tau \setminus \text{invalid}_a \\
& \text{valid}_{\tau, \langle t, p := \text{malloc}, up \rangle} := \text{valid}_\tau \cup \{p, a.\text{next}\} & \text{if } [p \mapsto a] \in up \\
& \text{valid}_{\tau, \langle t, \text{assume } p=q, up \rangle} := \text{valid}_\tau \cup \{p, q\} & \text{if } \{p, q\} \cap \text{valid}_\tau \neq \emptyset \\
& \text{valid}_{\tau, \text{act}} := \text{valid}_\tau & \text{otherwise}
\end{aligned}$$

with $\text{invalid}_a := \{p \mid m_\tau(p) = a\} \cup \{b.\text{next} \mid m_\tau(b.\text{next}) = a\} \cup \{a.\text{next}\}$.

With this definition, all pointer variables are valid initially. A pointer variable/selector becomes valid if it receives its value from an allocation or another valid pointer. A pointer becomes invalid if its referenced memory location is deleted or it receives its value from an invalid pointer. A deletion of an address makes invalid its pointer selectors and all pointers referencing that address. A subsequent reallocation of the address makes valid only the receiving pointer; all other pointers to the address remain invalid. Assumptions of the form $p = q$ validate p if q is valid, and vice versa.

We turn to the definition of ownership. It follows our previous discussion that allocations grant ownership while publishing removes it. Technically, we remove ownership of published addresses not in the moment they are published, but later upon the first access. This reduces the computational effort of tracking ownership information in tools since there is no need to compute the reachability of addresses in order to check if ownership is lost. To prevent ownership getting lost prematurely due to dangling pointers accessing owned addresses, our ownership definition takes validity into account: only first accesses through valid pointers remove ownership. The discussion yields the following definition.

Definition 6.6 (Ownership). The addresses owned by thread t in τ , $owned_\tau(t) \subseteq \text{Adr}$, are:

$$\begin{aligned}
 & owned_\epsilon(t) := \emptyset \\
 & owned_{\tau, \langle t', p := q, [p \mapsto a] \rangle}(t) := owned_\tau(t) \setminus \{a\} && \text{if } p \in \text{shared} \wedge q \in \text{valid}_\tau \\
 & owned_{\tau, \langle t', p := q.\text{next}, [p \mapsto a] \rangle}(t) := owned_\tau(t) \setminus \{a\} && \text{if } t \neq t' \wedge q, m_\tau(q).\text{next} \in \text{valid}_\tau \\
 & owned_{\tau, \langle t', p := q.\text{next}, [p \mapsto a] \rangle}(t) := owned_\tau(t) \setminus \{a\} && \text{if } t = t' \wedge p \in \text{shared} \\
 & owned_{\tau, \langle t', p := \text{malloc}, [p \mapsto a, \bullet] \rangle}(t) := owned_\tau(t) \cup \{a\} && \text{if } t = t' \wedge p \notin \text{shared} \\
 & owned_{\tau, \langle t', \text{free}(a), \emptyset \rangle}(t) := owned_\tau(t) \setminus \{a\} \\
 & owned_{\tau, \text{act}}(t) := owned_\tau(t) && \text{otherwise .}
 \end{aligned}$$

With the above notion of ownership, we are ready to give the main result of the section. It states that a thread t can reference/access the addresses owned by another thread t' , $t \neq t'$, only if the pointer of t is invalid, i.e., dangling. This validates the soundness of ownership reasoning when combining views during interference. With respect to the example from Figure 6.2, it prevents the spurious view v_4 and thus avoids the associated false alarm.

Theorem 6.7 (Ownership Guarantee). Consider $\tau \in \mathcal{O}[\llbracket P \rrbracket_{\text{Adr}}^{\text{Adr}}]$ with $m_\tau(p) \in owned_\tau(t)$. Then, $p \in \text{valid}_\tau$ implies $p \in \text{local}_t$.

It is worth pointing out that the above theorem does not impose any restrictions on the analyzed program, like if/how it accesses freed memory. Ownership is a universal technique to make thread-modular verification more precise. That it makes thread-modular verification sufficiently precise to establish correctness for the data structures of our interest is demonstrated below.

6.3 Evaluation

We devise an automated analysis to check linearizability of non-blocking data structures. Section 6.3.1 extends the thread-modular framework from Chapter 4 to safe memory reclamation and ownership reasoning. Section 6.3.2 evaluates the approach.

6.3.1 Integrating Safe Memory Reclamation

We extend the analysis from Chapter 4 to integrate SMR. To that end, we add SMR automata to views. Note that SMR automata have a pleasant interplay with thread-modularity. In a view for thread t only those automaton states are needed where t is observed. For $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$, for example, this means that only states with $z_t = t$ need to be stored in the view for t . Similarly, the memory abstraction induces a set of reachable addresses that need to be observed (in the case of $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$ by z_a). To keep the number of SMR automaton states per view small in practice, we do not keep states for shared addresses, i.e., addresses that are reachable from the shared variables. Instead, we maintain the invariant that they are never retired nor freed. For the analysis, we then assume that the ignored states are arbitrary (but not in locations implying retiredness or freedness). We found that all benchmark programs satisfied this invariant and that the resulting precision allowed for successful verification.

As discussed in Section 5.2, we need to check for double retires so that the use of \mathcal{O}_{Base} is sound. We integrate an appropriate check: verification fails upon `in:retire(p)` if p points to a and a is currently retired, that is, if \mathcal{O}_{Base} is in state (L_3, φ) with $\varphi = \{z_a \mapsto a\}$ prior to the call.

Ownership reasoning based on Theorem 6.7 is integrated easily by tracking validity and ownership information in views. Then, combined views can be discarded during interference if a thread holds a valid pointer to an addresses owned by another thread.

6.3.2 Linearizability Experiments

We implemented the approach presented in this chapter in a C++ tool called `TMREXP`² and empirically evaluated it on Treiber’s stack, Michael&Scott’s queue, and the DGLM queue. For a base line, we also evaluated a naive stack and a naive queue implementation both of which use a single lock. Our benchmarks do not include set implementations since the memory abstraction we built on cannot handle sortedness [Abdulla et al. 2013, 2017]; we stress that this is a shortcoming of the memory abstraction, not a shortcoming of the results we have established in this chapter. As SMR algorithm we used FL. Recall from Section 5.2 that we specify FL with \mathcal{O}_{Base} and assume that freed addresses can be accessed safely.

The findings are listed in Table 6.8. They include (i) the size of the explored state space, i.e., the number of reachable views, (ii) the number of interference steps that were performed as well as the number of interference steps that were omitted due to ownership reasoning, and (iii) the running time and result of verification (✓ for success and ✗

² `TMREXP` is freely available at: <https://wolf09.github.io/phd/>

Table 6.8: Experimental results for verifying singly-linked data structures using FL. The experiments were conducted on an Intel i5-8600K@3.6GHz with 16GB of RAM using Ubuntu 16.04 and Clang 6.0.

Program	Ownership ^a	States	Interferences (pruned)	Time
Single lock stack	GC	328	3.2k (10k)	0.006s ✓
	New	703	7k (22k)	0.21s ✓
	None	16k	183k (0)	5.34s ✓
Single lock queue	GC	100	0.7k (5k)	0.04s ✓
	New	520	0.7k (31k)	0.56s ✓
	None	27k	442k (0)	32s ✓
Treiber's stack	GC	269	3.5k (16k)	0.06s ✓
	New	744	44k (96k)	2.36s ✓
	None	117k	7920k (0)	602s ✓
Michael&Scott's queue	GC	3134	47k (1237k)	2.52s ✓
	New	19553	6678k (20748k)	3h ✓
	None	≥ 69000	—	2s ✗ ^b
DGLM queue	GC	3134	47k (1237k)	2.52s ✓
	New	≥ 6500	—	30s ✗ ^b
	None	≥ 64000	—	2s ✗ ^b

^a The ownership reasoning technique used: traditional reasoning as done under garbage collection, the new approach from this chapter, or none.

^b False positive due to imprecision in the memory abstraction.

for failure). Besides the novel ownership reasoning presented in this chapter, we include benchmarks for traditional and no ownership reasoning. For traditional ownership reasoning, as done under garbage collection, we also prevent memory from being reallocated in order to achieve a sound analysis. Without ownership reasoning, we maintain two threads per view in order to achieve acceptable precision, as proposed by Abdulla et al. [2013, 2017]. All experiments were conducted on an Intel i5-8600K@3.6GHz with 16GB of RAM using Ubuntu 16.04 and Clang 6.0.

Our experiments substantiate the usefulness of the proposed ownership technique. It gives a speed-up of up to two orders of magnitude for Treiber's stack and the naive implementations. The running times are a middle ground between a GC analysis and the full analysis as suggested by Abdulla et al. [2013]. The size of the state space explored by the new technique is much closer to GC than to the full analysis. Comparing the results for Treiber's stack and Michael&Scott's queue, however, suggests that the blow up introduced by memory reclamation and subsequent reuse is too severe to handle more elaborate data structures or more elaborate SMR algorithms. In the following chapters we propose new methods to fight this problem.

Pointer Races

While the compositional verification approach from Chapter 5 abstracts away the implementation details of the SMR algorithm, it leaves the verifier with a hard task, as seen in Chapter 6: memory reclamation in the presence of fine-grained concurrency. To alleviate the impact of memory reclamation on the analysis, we show that *one can soundly verify a data structure $P(\mathcal{O})$ by considering only those computations where at most a single address is reused*. This avoids the need for an exploration of full $\mathcal{O}[\![P]\!]_{Adr}^{Adr}$ which suffers from a severe state space explosion. In fact, we were not able to make an analysis go through with only the compositional approach from Chapter 5; we need the reduction result presented in the following. The analysis from Chapter 6 as well as previous works on automated data structure verification [Abdulla et al. 2013; Holik et al. 2017] have not required such a reduction since they considered FL rather than full-featured SMR algorithms like EBR and HP.

Our results are independent of the actual safety property and the actual automaton \mathcal{O} specifying the SMR algorithm. To achieve this, we establish that for every computation from $\mathcal{O}[\![P]\!]_{Adr}^{Adr}$ there is a *similar* computation which reuses at most a single address. We construct the similar computation by eliding reuse in the original computation. With elision we mean that we replace in a computation a freed address with a fresh one. This allows a subsequent allocation to `malloc` the elided address fresh instead of reusing it. Our notion of similarity makes sure that both computations reach the same control locations. This allows for verifying safety properties.

The remainder of the chapter is structured as follows. Section 7.1 introduces our notion of computation similarity. Section 7.2 formalizes requirements on $P(\mathcal{O})$ such that similarity suffices to prove the desired reduction result. Section 7.3 discusses how the ABA problem can affect the soundness of our approach and shows how to detect those cases. Section 7.4 presents the reduction result. Section 7.5 evaluates our approach.

7.1 Similarity of Computations

Our goal is to mimic a computation τ where memory is reused arbitrarily with a computation σ where memory reuse is restricted. As noted before, we want the threads in τ and σ to reach the same control locations in order to verify safety properties of τ in σ . We introduce a *similarity relation* among computations such that τ and σ are similar if they can execute the same actions. This results in both computations reaching the same control locations, as desired. However, control location equality alone is insufficient for σ to mimic subsequent actions of τ , that is, to preserve similarity for subsequent actions. This is because most actions involve memory interaction. Since σ reuses memory differently than τ , the memory of the two computations is not equal. Similarity requires a non-trivial correspondence wrt. the memory. Towards a formal definition let us consider an example.

Example 7.1. Let τ_1 be a computation of a data structure $P(\mathcal{O}_{Base} \times \mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1)$ using HP:

$$\begin{aligned} \tau_1 = & \langle t, p := \text{malloc}, [p \mapsto a, \dots] \rangle . \langle t, \text{in:retire}(p), \emptyset \rangle . \langle t, \text{free}(a), \emptyset \rangle . \langle t, \text{re:retire}, \emptyset \rangle . \\ & \langle t, q := \text{malloc}, [q \mapsto a, \dots] \rangle . \end{aligned}$$

In this computation, thread t uses pointer p to allocate address a . The address is then retired and freed. In the subsequent allocation, t acquires another pointer q to a ; a is reused.

If σ_1 is a computation where a shall not be reused, then σ_1 is not able to execute the exact same sequence of actions as τ_1 . However, it can mimic τ_1 as follows:

$$\begin{aligned} \sigma_1 = & \langle t, p := \text{malloc}, [p \mapsto b, \dots] \rangle . \langle t, \text{in:retire}(p), \emptyset \rangle . \langle t, \text{free}(b), \emptyset \rangle . \langle t, \text{re:retire}, \emptyset \rangle . \\ & \langle t, q := \text{malloc}, [q \mapsto a, \dots] \rangle , \end{aligned}$$

where σ_1 coincides with τ_1 up to replacing the first allocation of a with another address b . We say that σ_1 elides the reuse of a . The memories of τ_1 and σ_1 differ on p and agree on q . ■

In the above example, p is a dangling pointer. Programmers typically avoid using such pointers because it is unsafe. For a definition of similarity, this practice suggests that similar computations must coincide only on the non-dangling pointers and may differ on the dangling ones. To make this precise, recall the notion of validity, Definition 6.5: the non-dangling pointers are precisely the valid pointers.

Example 7.2 (Continued). In both τ_1 and σ_1 from the previous example, the last allocation renders valid pointer q . On the other hand, the free to a in τ_1 renders p invalid. The reallocation of a does not change the validity of p , it remains invalid. In σ_1 , address b is allocated and freed rendering p invalid. It remains invalid after the subsequent allocation of a . That is, both τ_1 and σ_1 agree on the validity of q and the invalidity of p . Moreover, τ_1 and σ_1 agree on the valuation of the valid q and disagree on the valuation of the invalid p . ■

The above example illustrates that eliding reuse of memory leads to a different memory valuation. However, the elision can be performed in such a way that the valid memory is not affected. So we say that two computations are similar if they agree on the resulting control locations of threads and the valid memory. The valid memory includes the valid pointer variables, the valid pointer selectors, the data variables, and the data selectors of addresses that are referenced by a valid pointer variable-selector. Formally, this is a restriction of the entire memory to the valid pointer expressions, written $m_\tau|_{\text{valid}_\tau}$.

Definition 7.3 (Restrictions). A restriction of memory m to a set $P \subseteq PExp$, written $m|_P$, is a new memory m' with domain $\text{dom}(m') := P \cup DVar \cup \{ a.\text{data} \in DExp \mid a \in m(P) \}$ such that $m(e) = m'(e)$ for all $e \in \text{dom}(m')$.

We are now ready to formalize the notion of similarity among computations. Two computations are similar if they agree on the control location of threads and the valid memory.

Definition 7.4 (Computation Similarity). Two computations τ and σ are similar, denoted by $\tau \sim \sigma$, if we have $\text{ctrl}(\tau) = \text{ctrl}(\sigma)$ and $m_\tau|_{\text{valid}_\tau} = m_\sigma|_{\text{valid}_\sigma}$.

If two computations τ and σ are similar, then each action enabled after τ can be mimicked in σ . More precisely, action $act = \langle t, com, up \rangle$ after τ can be mimicked by $act' = \langle t, com, up' \rangle$ after σ . Both actions agree on the executing thread and the executed command, but may differ in the memory update. The reason for this is that similarity does not relate the invalid parts of the memory. This may give another update in σ if com involves invalid pointers.

Example 7.5 (Continued). Consider the following continuation of τ_1 and σ_1 :

$$\tau_2 = \tau_1 \cdot \langle t, p := p, up \rangle \quad \text{and} \quad \sigma_2 = \sigma_1 \cdot \langle t, p := p, up' \rangle$$

where we append an assignment of p to itself. The prefixes τ_1 and σ_1 are similar, $\tau_1 \sim \sigma_1$. Nevertheless, the updates up and up' differ because they involve the valuation of the invalid pointer p which differs in τ_1 and σ_1 . The updates are $up = [p \mapsto a]$ and $up' = [p \mapsto b]$. Since the assignment leaves p invalid, similarity is preserved by the appended actions, $\tau_2 \sim \sigma_2$. We say that act' mimics act .

Altogether, similarity does not guarantee that the exact same actions are executable. It guarantees that every action can be mimicked such that similarity is preserved. ■

In the above we omitted an integral part of the program semantics. Memory reclamation is not based on the control location of threads but on an SMR automaton examining the history induced by a computation. The enabledness of a `free` is not preserved by similarity. On the one hand, this is due to the fact that invalid pointers can be (and in practice are) used in SMR calls which leads to different histories. On the other hand, similar computations end up in the same control location but may perform different sequences of actions to arrive there, for instance, execute different branches of conditionals. That is, to mimic `free` actions we need to correlate the behavior of the SMR automaton rather than the behavior of the program. We motivate the definition of an appropriate relation.

Example 7.6 (Continued). Consider the computations $\tau_3 = \tau_2 \cdot \gamma$ and $\sigma_3 = \sigma_2 \cdot \gamma$ with

$$\gamma = \langle t, \text{in:protect}_k(p), \emptyset \rangle \cdot \langle t, \text{re:protect}_k, \emptyset \rangle \cdot \langle t, \text{in:retire}(q), \emptyset \rangle \cdot \langle t, \text{re:retire}, \emptyset \rangle$$

where thread t issues a protection and a retirement using p and q , respectively. The induced histories are:

$$\begin{aligned} \mathcal{H}(\tau_3) &= \mathcal{H}(\tau_2) \cdot \text{in:protect}_k(t, a) \cdot \text{re:protect}_k(t) \cdot \text{in:retire}(t, a) \cdot \text{re:retire}(t) \\ \text{and } \mathcal{H}(\sigma_3) &= \mathcal{H}(\sigma_2) \cdot \text{in:protect}_k(t, b) \cdot \text{re:protect}_k(t) \cdot \text{in:retire}(t, a) \cdot \text{re:retire}(t) . \end{aligned}$$

Recall that τ_2 and σ_2 are similar. Similarity guarantees that the events of the `retire` call coincide since q is valid. The events of the `protect` call differ because the valuations of the invalid p differ. That is, SMR calls do not necessarily emit the same event in similar computations. Consequently, the SMR automaton reaches different states after τ_3 and σ_3 . More precisely, automaton \mathcal{O}_{HP} from Figure 5.4 takes the following steps from the initial state (L_8, φ) with valuation $\varphi = \{z_t \mapsto t, z_a \mapsto a\}$:

$$\begin{aligned} (L_8, \varphi) &\xrightarrow{\mathcal{H}(\tau_2)} (L_8, \varphi) \xrightarrow{\text{in:protect}_k(t, a)} (L_9, \varphi) \xrightarrow{\text{re:protect}_k(t)} (L_{10}, \varphi) \\ &\quad \xrightarrow{\text{in:retire}(t, a)} (L_{11}, \varphi) \xrightarrow{\text{re:retire}(t)} (L_{11}, \varphi) \\ \text{and } (L_8, \varphi) &\xrightarrow{\mathcal{H}(\sigma_2)} (L_8, \varphi) \xrightarrow{\text{in:protect}_k(t, b)} (L_8, \varphi) \xrightarrow{\text{re:protect}_k(t)} (L_8, \varphi) \\ &\quad \xrightarrow{\text{in:retire}(t, a)} (L_8, \varphi) \xrightarrow{\text{re:retire}(t)} (L_8, \varphi) . \end{aligned}$$

This prevents a from being freed after τ_3 , because a $\text{free}(a)$ would lead to the final state (L_{12}, φ) and is thus not enabled, but allows for freeing it after σ_3 . ■

The above example shows that eliding memory addresses to avoid reuse may change SMR automaton steps. The affected steps involve freed addresses. Like for computation similarity, we define a relation among computations which captures the SMR behavior on the *valid addresses*, i.e., those addresses that are referenced by valid pointers, and ignores all other addresses. Here, we do not use an equivalence relation. That is, we do not require SMR automata to reach the exact same state for valid addresses. Instead, we express that the mimicking σ allows for more behavior on the valid addresses than the mimicked τ . We define an *SMR behavior inclusion* among computations. This is motivated by the above example. There, address a is valid because it is referenced by the valid pointer q . Yet the SMR automaton steps for a differ in τ_3 and σ_3 . After σ_3 strictly more behavior is possible: σ_3 can free a while τ_3 cannot.

To make this intuition precise, we need a notion of *behavior on an address*. Recall that the goal of the desired behavior inclusion is to enable us to mimic frees. Intuitively, the behavior allowed by \mathcal{O} on address a is the set of those histories that *lead* to a free of a .

Definition 7.7 (SMR Behavior). The behavior allowed by automaton \mathcal{O} on address a after history h is the set $\mathcal{F}_{\mathcal{O}}(h, a) := \{ h' \mid h.h' \in \mathcal{S}(\mathcal{O}) \wedge \text{frees}_{h'} \subseteq a \}$.

Note that $h' \in \mathcal{F}_{\mathcal{O}}(h, a)$ contains free events for address a only, as dictated by $\text{frees}_{h'} \subseteq a$. This side condition is necessary because an address may become invalid before being freed if, for instance, the address becomes unreachable from the valid pointers. Despite similarity, the mimicking computation σ may have already freed such an address while τ has not. Hence, the free is no longer allowed after σ but still possible after τ . To prevent such invalid addresses from breaking the desired inclusion on valid addresses, we strip from $\mathcal{F}_{\mathcal{O}}(h, a)$ all frees that do not target a . Note that we do not even retain frees of valid addresses here. This way, only SMR-related actions influence $\mathcal{F}_{\mathcal{O}}(h, a)$. To be more precise, we have $\mathcal{F}_{\mathcal{O}}(\mathcal{H}(\tau), a) = \mathcal{F}_{\mathcal{O}}(\mathcal{H}(\tau.\text{act}), a)$ for all actions act which do not emit an event.

The SMR behavior inclusion among computations is defined such that σ includes at least the behavior of τ on the valid addresses. To make this formal, we define the addresses that are in use in a memory m , denoted $\text{adr}(m)$, by

$$\text{adr}(m) := (\text{dom}(m) \cup \text{range}(m)) \cap \text{Adr}$$

where we use $\{ a.\text{next} \} \cap \text{Adr} = a$ and likewise for data selectors. Then, the valid addresses in τ are $\text{adr}(m_\tau|_{\text{valid}_\tau})$.

Definition 7.8 (SMR Behavior Inclusion). Computation σ includes the SMR behavior of τ , denoted by $\tau \leq \sigma$, if $\mathcal{F}_{\mathcal{O}}(\tau, a) \subseteq \mathcal{F}_{\mathcal{O}}(\sigma, a)$ holds for all $a \in \text{adr}(m_\tau|_{\text{valid}_\tau})$.

7.2 Preserving Similarity

The development in Section 7.1 is idealized. There are cases where the introduced relations do not guarantee that an action can be mimicked. All such cases have in common that they involve invalid pointers. More precisely, (i) the computation similarity may not be strong enough to mimic actions that dereference invalid pointers, and (ii) the SMR behavior inclusion may not be strong enough to mimic calls involving invalid pointers. For each of those cases we give an example and restrict our development. We argue throughout this section that our restrictions are reasonable. Our experiments confirm this. We start with the computation similarity.

Example 7.9 (Continued). Consider the following continuation of τ_3 and σ_3 :

$$\begin{aligned} \tau_4 &= \tau_3.\langle t, q.\text{next} := q, [a.\text{next} \mapsto a] \rangle.\langle t, p.\text{next} := p, [a.\text{next} \mapsto a] \rangle \\ \text{and } \sigma_4 &= \sigma_3.\langle t, q.\text{next} := q, [a.\text{next} \mapsto a] \rangle.\langle t, p.\text{next} := p, [b.\text{next} \mapsto b] \rangle. \end{aligned}$$

The first appended action updates $a.\text{next}$ in both computations to a . Since q is valid after both τ_3 and σ_3 this assignment renders valid $a.\text{next}$. The second action updates $a.\text{next}$ in τ_4 . This results in $a.\text{next}$ being invalid after τ_4 because the right-hand side of the assignment is the invalid p . In σ_4 the second action updates $b.\text{next}$ which is why $a.\text{next}$ remains valid. That is, the valid memories of τ_4 and σ_4 differ. We have executed an action that cannot be mimicked on the valid memory despite the computations being similar. ■

The problem in the above example is the dereference of an invalid pointer. The computation similarity does not give any guarantees about the valuation of such pointers. Consequently, it cannot guarantee that an action using invalid pointers can be mimicked. To avoid such problems, we forbid programs to dereference invalid pointers.

The rationale behind this is as follows. Recall that an invalid pointer is dangling. That is, the memory it references has been freed. If the memory has been returned to the underlying operating system, then a subsequent dereference is unsafe, i.e., prone to a system crash due to a `segfault`. Hence, such dereferences should be avoided. The dereference is only safe if the memory is guaranteed to be accessible. To decide this, the invalid pointer needs to be compared with a definitely valid pointer. Such a comparison renders valid the invalid pointer (cf. Definition 6.5). This means that dereferences of invalid pointers are always unsafe. We let verification fail if unsafe accesses are performed. That performance-critical and non-blocking code is free from unsafe accesses is confirmed by our experiments.

Definition 7.10 (Unsafe Access). A computation $\tau.\langle t, com, up \rangle$ performs an unsafe access if com contains $p.\text{data}$ or $p.\text{next}$ with $p \notin \text{valid}_\tau$.

Forbidding unsafe accesses makes the computation similarity strong enough to mimic all desired actions. A discussion of cases where the SMR behavior inclusion cannot be preserved is in order. We start with an example.

Example 7.11 (Continued). Consider the following continuations of τ_1, σ_1 from Example 7.1:

$$\begin{aligned} \tau_5 &= \tau_1.\langle t, \text{in:retire}(p), \emptyset \rangle \quad \text{with} \quad \mathcal{H}(\tau_5) = \mathcal{H}(\tau_1).\text{in:retire}(t, a) \\ \text{and} \quad \sigma_5 &= \sigma_1.\langle t, \text{in:retire}(p), \emptyset \rangle \quad \text{with} \quad \mathcal{H}(\sigma_5) = \mathcal{H}(\sigma_1).\text{in:retire}(t, b). \end{aligned}$$

The SMR behavior of τ_1 is included in σ_1 , that is, $\tau_1 < \sigma_1$. After τ_5 a deletion of a is possible because it is retired. After σ_5 a deletion of a is prevented by \mathcal{O}_{Base} because a has not been retired. Formally, we have $\text{free}(a) \in \mathcal{F}_\mathcal{O}(\tau_5, a)$ and $\text{free}(a) \notin \mathcal{F}_\mathcal{O}(\sigma_5, a)$. However, a is a valid address because it is referenced by the valid pointer q . That is, the behavior inclusion among τ_1 and σ_1 is not preserved by the subsequent action. ■

The above example showcases that calls to the SMR algorithm can break the SMR behavior inclusion. This is the case because an action can emit different events in similar computations. The event emitted by an SMR call differs only if it involves invalid pointers.

The naive solution would be to prevent using invalid pointers in calls altogether. In practice, this is too strong a requirement. As seen in Chapter 2, a common pattern for protecting an address with hazard pointers is to (i) read a pointer p into a local variable q , (ii) issue a protection using q , and (iii) repeat the process if p and q do not coincide.¹ After reading into q and before protecting q the referenced memory may be freed. Hence, the protection is prone to use invalid pointers. Forbidding such protections would render our theory inapplicable to non-blocking data structures using hazard pointers.

To fight this problem, we forbid only those calls involving invalid pointers which are prone to *break* the SMR behavior inclusion. Intuitively, this is the case if there exists another call which differs only on the invalid pointer arguments and allows for more behavior on the valid addresses than the original call. To regain precision and support more scenarios where invalid pointers are used, we keep unchanged the address the behavior of which is considered.

Definition 7.12 (Racy Call). A Computation $\tau.\langle t, \text{in:func}(\bar{r}), \emptyset \rangle$ with $\mathcal{H}(\tau) = h$ and $m_\tau(\bar{r}) = \bar{v}$ performs a racy call if:

$$\begin{aligned} \exists a \exists \bar{w}. \quad & (\forall i. (v_i = a \vee r_i \in \text{valid}_\tau \vee r_i \in \text{DExp}) \implies v_i = w_i) \\ & \wedge \mathcal{F}_\mathcal{O}(h.\text{in:func}(t, \bar{w}), a) \not\subseteq \mathcal{F}_\mathcal{O}(h.\text{in:func}(t, \bar{v}), a) \end{aligned}$$

It follows immediately that calls containing only valid pointers are not racy. Using the SMR automaton for HP, $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$, we deem racy the call to `retire` from Example 7.11—we reject the program and let verification fail. Indeed, requesting the deferred deletion of an invalid pointer might lead to a double free, resulting in a system crash. For that reason, `retire` is always called using valid pointers in practice. For `protect` calls one can show that they never race. We have already seen this in Example 7.6. There, a call to `protect` with invalid pointers did not break the SMR behavior inclusion. Instead, the mimicking computation σ_3 could perform strictly more frees than the computation it mimicked τ_3 .

¹ For an instantiation of this pattern, consider Lines 314 to 316 of Michael&Scott's queue from Figure 2.13.

We uniformly refer to the above situations where the usage of an invalid pointer can break the ability to mimic an action as a *pointer race*. It is a race indeed because the usage and the reclamation of a pointer are not properly synchronized.

Definition 7.13 (Pointer Race). A computation $\tau.act$ is a pointer race if act performs (i) an unsafe access, or (ii) a racy SMR call.

With pointer races we restrict the class of supported programs. The restriction to pointer race free programs is reasonable in that we can handle common non-blocking data structures from the literature as shown in our experiments. Since we want to give the main result of this section in a general fashion that does not rely on the actual SMR automaton used to specify the SMR implementation, we have to restrict the class of supported SMR automata as well.

We require that the SMR automaton supports the elision of reused addresses, as done in Example 7.1. Intuitively, elision is a two-step process the automaton must be insensitive to. First, an address a is swapped with a fresh address b upon an allocation where a should be reused but cannot. In the resulting computation, a is fresh and thus the allocation can be performed without reusing a . The process of swapping a with b must not affect the behavior of the automaton on addresses other than a and b . Second, the automaton must allow for more behavior on the fresh address than on the reused address. This is required to preserve the SMR behavior inclusion because the allocation renders a valid.

Additionally, we require a third property: the SMR automaton behavior on an address must not be influenced by frees of another address. This is needed because computation similarity and SMR behavior inclusion do not guarantee that frees of invalid addresses can be mimicked, as discussed before. Since such frees do not affect the valid memory, there is no need to mimic them. The SMR automaton has to allow us to simply *skip* such frees when mimicking a computation.

For a formal definition of our intuition we write $h[a/b]$ to denote the history that is constructed from h by swapping every occurrence of a and b . Moreover, we write $a \in \text{fresh}_h$ and mean that address a does not appear in any of the events of h .

Definition 7.14 (Elision Support). SMR automaton $\mathcal{O} = \mathcal{O}_{Base} \times \mathcal{O}_{SMR}$ supports elision of memory reuse if for all $h, h' \in \mathcal{S}(\mathcal{O}_{Base})$ and $a, b, c \in \text{Adr}$ the following conditions are met:

- (i) $a \neq c \neq b$ implies $\mathcal{F}_{\mathcal{O}_{SMR}}(h, c) = \mathcal{F}_{\mathcal{O}_{SMR}}(h[a/b], c)$,
- (ii) $\mathcal{F}_{\mathcal{O}}(h, a) \subseteq \mathcal{F}_{\mathcal{O}}(h', a)$ and $b \in \text{fresh}_{h'}$ implies $\mathcal{F}_{\mathcal{O}_{SMR}}(h, b) \subseteq \mathcal{F}_{\mathcal{O}_{SMR}}(h', b)$, and
- (iii) $a \neq b$ and $h.\text{free}(a) \in \mathcal{S}(\mathcal{O})$ implies $\mathcal{F}_{\mathcal{O}_{SMR}}(h, b) = \mathcal{F}_{\mathcal{O}_{SMR}}(h.\text{free}(a), b)$.

Crucially, the above definition is concerned with the SMR-specific part \mathcal{O}_{SMR} only. Automaton \mathcal{O}_{Base} does not satisfy Property (ii) for arbitrary histories h, h' . The problem is that an $\text{in:retire}(t, b)$ in history h takes \mathcal{O}_{Base} from its initial location L_2 to location L_3 . In L_3 a $\text{free}(b)$ is allowed. However, that b is fresh in h' means that \mathcal{O}_{Base} is in L_2 where $\text{free}(b)$ is not allowed. So, h' does not include all of h 's behavior. When constructing for τ a mimicking computation σ that elides the reuse of b , the problematic scenario occurs only if b is both freed and retired after τ .

This, in turn, gives rise to a double retire on b . To see this, observe that b must have been retired after being freed as otherwise \mathcal{O}_{Base} would be in L_2 rather than L_3 after $h = \mathcal{H}(\tau)$. For b to be freed, there must be a preceding retirement according to \mathcal{O}_{Base} . Removing the free, we obtain a computation where b remains retired and is thus retired twice. Hence, a check for double retires, which Section 5.2 mandates anyways, yields a lift of Property (ii) to full \mathcal{O} in the relevant situations. It is worth pointing out that we cannot rely on pointer races here. While the SMR automata we use would deem racy a retirement of a freed and thus invalid address, this is not guaranteed in general.

We found Definition 7.14 practical in that the SMR automata for specifying HP and EBR from Figure 5.4, which we use for our experiments in Section 7.5, support elision.

Proposition 7.15. The SMR automata $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$ and $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$ from Figure 5.4 as well as $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^{0,1}$ support elision.

7.3 Detecting ABAs

So far we have introduced restrictions, namely pointer race freedom and elision support, to rule out cases where our idea of eliding memory reuse does not work, that is, breaks similarity or the SMR behavior inclusion. If those restrictions were strong enough to carry out our development, then we could remove any reuse from a computation and get a similar one where no memory is reused. That the resulting computation does not reuse memory means, intuitively, that it is executed under garbage collection. As shown in the literature [Michael and Scott 1996], the ABA problem is a subtle bug caused by manual memory management which is prevented by garbage collection. So, eliding all reuses jeopardizes soundness of the analysis—it could miss ABAs which result in a safety violation. With this observation, we elide all reuses except for one address per computation. This way we analyze a semantics that is close to garbage collection, can detect ABA problems, and is much simpler than full $\mathcal{O}[[P]]_{Adr}^{Adr}$.

The semantics that we suggest to analyze is $\mathcal{O}[[P]]_{Adr}^{one} := \bigcup_{a \in Adr} \mathcal{O}[[P]]_{Adr}^{\{a\}}$. It is the set of all computations that reuse at most a single address. A single address suffices to detect the ABA problem. The ABA problem manifests as an assumption of the form $p = q$ where the addresses held by p and q coincide but stem from different allocations. That is, one of the pointers has received its address, the address was freed and then reallocated, before the pointer is used in the assumption. Note that this implies that for an assumption to be ABA one of the involved pointers must be invalid. Pointer race freedom does not forbid this. Nor do we want to forbid such assumptions. In fact, most programs using hazard pointers contain ABAs. They are written in a way that ensures that the ABA is *harmless*.

Example 7.16 (ABAs in Michael&Scott’s queue using hazard pointers). Consider the following code, repeated from Michael&Scott’s queue from Figure 2.13:

```

314 Node* head = Head;
315 protect0(head);
316 if (head != Head) continue;
```

In Line 314 the value of the shared pointer `Head` is read into the local pointer `head`. Then, a hazard pointer is used in Line 315 to protect `head` from being freed. In between reading and protecting `head`, its address could have been deleted, reused, and reentered the queue. That is, when executing Line 316 the pointers `Head` and `head` can coincide although the `head` pointer stems from an earlier allocation. This scenario is an ABA. Nevertheless, the queue's correctness is not affected by it. The ABA prone assumption is only used to guarantee that the address protected in Line 315 is indeed protected after Line 316. With respect to the SMR automaton \mathcal{O}_{HP} , the assumption guarantees that the protection was issued before a retirement (after the latest reallocation) so that \mathcal{O}_{HP} is guaranteed to be in L_{10} and thus prevents future retirements from freeing the protected memory. The ABA does not void this guarantee, it is harmless. ■

The above example shows that non-blocking data structures may perform ABAs which do not affect their correctness. To soundly verify such algorithms, our approach is to detect every ABA and decide whether it is harmless indeed. If so, our verification is sound. Otherwise, we report to the programmer that the implementation suffers from a harmful ABA problem.

A discussion of how to detect ABAs is in order. Let $\tau \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$ and $\sigma \in \mathcal{O}[\![P]\!]_{Adr}^{\{a\}}$ be similar computations. Intuitively, σ is a computation which elides the reuses from τ except for address a . Address a can be used in σ in exactly the same way as it is used in τ . Let act be an ABA prone assumption of the form $act = \langle t, \text{assume } p = q, \emptyset \rangle$. Assume act is enabled after τ . To detect this ABA under $\mathcal{O}[\![P]\!]_{Adr}^{\{a\}}$ we need act to be enabled after σ . We seek to have $\sigma.act \in \mathcal{O}[\![P]\!]_{Adr}^{\{a\}}$. This is not guaranteed. Since act is an ABA it involves at least one invalid pointer, say p . Computation similarity does not guarantee that p has the same valuation in both τ and σ . However, if p points to a in τ , then it does so in σ because a is (re)used in σ in the same way as in τ . Thus, we end up with $m_\tau(p) = m_\sigma(p)$ although p is invalid. In order to guarantee this, we introduce an *address alignment* relation which precisely tracks how the reusable address a is used.

Definition 7.17 (Address Alignment). Computations τ and σ are a -aligned, $\tau \preceq_a \sigma$, if:

$$\begin{aligned}
& \forall p \in PVar. \ m_\tau(p) = a \iff m_\sigma(p) = a \\
& \text{and } \forall b \in m_\tau(\text{valid}_\tau). \ m_\tau(b.\text{next}) = a \iff m_\sigma(b.\text{next}) = a \\
& \text{and } a \in \text{fresh}_\tau \cup \text{freed}_\tau \iff a \in \text{fresh}_\sigma \cup \text{freed}_\sigma \\
& \text{and } \mathcal{F}_\mathcal{O}(\tau, a) \subseteq \mathcal{F}_\mathcal{O}(\sigma, a) \\
& \text{and } a \in \text{retired}_\tau \iff a \in \text{retired}_\sigma.
\end{aligned}$$

The first line in this definition states that the same pointer variables in τ and σ are pointing to a . Similarly, the second line states this for the pointer selectors of valid addresses. We have to exclude the invalid addresses here because τ and σ may differ on the in-use addresses due to eliding reuse. The third line states that a can be allocated in τ iff it can be allocated in σ . The fourth line states that the SMR automaton allows for more behavior on a in σ than in τ . These properties combined guarantee that σ can mimic actions of τ involving a no matter if invalid pointers are used. The last line requires that a is retired in τ iff it is retired in σ . This property makes double retires performed after τ visible in the mimicking σ .

The address alignment lets us detect ABAs in $\mathcal{O}[[P]]_{Adr}^{one}$. Intuitively, we can only detect *first* ABAs because we allow for only a single address to be reused. Subsequent ABAs on different addresses cannot be detected. To detect ABA sequences of arbitrary length, an arbitrary number of reusable addresses is required. To avoid this, i.e., to avoid an analysis of full $\mathcal{O}[[P]]_{Adr}^{Adr}$, we formalize the idea of *harmless ABAs* from before. We say that an ABA is harmless if executing it leads to a system state which can be explored (by another computation) without performing an ABA. That the system state can be explored without performing an ABA means that every ABA is also a first ABA. Thus, any sequence of ABAs is explored by considering only first ABAs. Note that this definition is independent of the actual correctness notion.

Definition 7.18 (Harmful ABA). The semantics $\mathcal{O}[[P]]_{Adr}^{one}$ is free from harmful ABAs if:

$$\forall \sigma_a.act \in \mathcal{O}[[P]]_{Adr}^{\{a\}} \forall \sigma_b \in \mathcal{O}[[P]]_{Adr}^{\{b\}} \exists \sigma'_b \in \mathcal{O}[[P]]_{Adr}^{\{b\}}.$$

$$\sigma_a \sim \sigma_b \wedge act = \langle \bullet, \text{assume } \bullet, \bullet \rangle \implies \sigma_a.act \sim \sigma'_b \wedge \sigma_b \preceq_b \sigma'_b \wedge \sigma_a.act < \sigma'_b.$$

To understand how the definition implements our intuition, consider $\tau.act \in \mathcal{O}[[P]]_{Adr}^{Adr}$ where act performs an ABA on address a . Our goal is to mimic $\tau.act$ in $\mathcal{O}[[P]]_{Adr}^{\{b\}}$, that is, we want to mimic the ABA without reusing address a (for instance, to detect subsequent ABAs on address b). Assume we are given $\sigma_b \in \mathcal{O}[[P]]_{Adr}^{\{b\}}$ which is similar and b -aligned to τ . This does not guarantee that act can be mimicked after σ_b ; the ABA may not be enabled because it involves invalid pointers the valuation of which differs in τ and σ_b . However, we can construct a computation σ_a which is similar and a -aligned to τ . After σ_a the ABA is enabled, i.e., $\sigma_a.act \in \mathcal{O}[[P]]_{Adr}^{\{a\}}$. For those two computations $\sigma_a.act$ and σ_b we invoke the above definition. It yields another computation $\sigma'_b \in \mathcal{O}[[P]]_{Adr}^{\{b\}}$ which, intuitively, coincides with σ_b but where the ABA has already been executed. Put differently, σ'_b is a computation which mimics the execution of act after σ_b although act is not enabled.

Example 7.19 (Continued). Consider the computation $\tau.act$ of Michael&Scott's queue with:

$$\tau = \tau_6.\langle t, \text{head} := \text{Head}, [\text{head} \mapsto a] \rangle.\tau_7.\text{free}(a).\tau_8.\langle t, \text{in:protect}_0(\text{head}), \emptyset \rangle.\langle t, \text{re:protect}_0, \emptyset \rangle$$

$$\text{and } act = \langle t, \text{assume head} = \text{Head}, \emptyset \rangle.$$

This computation resembles a thread t executing Lines 314 to 316 while an interferer frees address a referenced by head , reallocates it, and makes it the Head of the queue again; we assume that τ_6, τ_7, τ_8 consist of the interferer's actions the precise form of which does not matter here. The assume in act resembles the conditional from Line 316 and states that the condition evaluates to true . That is, act is a potential ABA on address a .

Reusing address a allows us to mimic τ with an a -aligned computation $\sigma_a \in \mathcal{O}[[P]]_{Adr}^{\{a\}}$. The ABA prone action act is guaranteed to be enabled after σ_a , so $\sigma_a.act$ mimics $\tau.act$. Reusing another address b yields a b -aligned $\sigma_b \in \mathcal{O}[[P]]_{Adr}^{\{b\}}$ mimicking τ . After σ_b , act may not be enabled. The reason for this is that σ_b elides allocations of a to avoid it being reused. The interferer's reallocation of a (in τ_8) forces σ_b to elide its previous allocation. Hence, thread t 's head does not point to a while Head still does. The ABA prone act is not enabled after σ_b .

To see that the above ABA is harmless, consider the following rescheduling of the actions in τ :

$$\tau' = \tau_6.\tau_7.\text{free}(a).\tau_8.\langle t, \text{head} := \text{Head}, [\text{head} \mapsto a] \rangle.\langle t, \text{in:protect}_0(\text{head}), \emptyset \rangle.\langle t, \text{re:protect}_0, \emptyset \rangle.$$

Here, thread t reads the latest version of `Head`. This gives rise to a computation $\sigma'_b \in \mathcal{O}[\![P]\!]_{Adr}^{\{b\}}$ mimicking τ' . Unlike σ_b , however, σ'_b can execute `act` since the later read of `head` is not affected by the elision of reallocations. Finally, $\sigma'_b.act$ mimics $\tau.act$. Requiring the existence of such a σ'_b guarantees that an analysis can *see past* ABAs on address a , although a is not reused. ■

A key aspect of the above definition is that checking for harmful ABAs can be done in the simpler semantics $\mathcal{O}[\![P]\!]_{Adr}^{one}$. Altogether, this means that we can rely on $\mathcal{O}[\![P]\!]_{Adr}^{one}$ for both the actual analysis and a soundness (absence of harmful ABAs) check. Our experiments show that the above definition is practical. There were no harmful ABAs in the benchmarks we considered.

7.4 Reduction Result

We show how to exploit the concepts introduced so far to soundly verify safety properties and establish the absence of double retires in the simpler semantics $\mathcal{O}[\![P]\!]_{Adr}^{one}$ instead of full $\mathcal{O}[\![P]\!]_{Adr}^{Adr}$.

Theorem 7.20. Let \mathcal{O} support elision. Let $\mathcal{O}[\![P]\!]_{Adr}^{one}$ be free from pointer races, double retires, and harmful ABAs. Then, for all $\tau \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$ and for all $a \in Adr$ there is $\sigma \in \mathcal{O}[\![P]\!]_{Adr}^{\{a\}}$ with $\tau \sim \sigma$, $\tau < \sigma$, and $\tau \preceq_a \sigma$.

Proof Sketch. We construct σ inductively by mimicking every action from τ and eliding reuses as needed. For the construction, consider $\tau.act \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$ and assume we have already constructed, for every $a \in Adr$, an appropriate $\sigma_a \in \mathcal{O}[\![P]\!]_{Adr}^{\{a\}}$. Consider some address $a \in Adr$. The task is to mimic `act` in σ_a . If `act` is an assignment or an SMR call, then pointer race freedom guarantees that we can mimic `act` by executing the same command with a possibly different update. We discussed this in Section 7.2. The interesting cases are ABAs, frees, and allocations.

First, consider the case where `act` executes an ABA assumption `assume p = q`. That the assumption is an ABA means that at least one of the pointers is invalid, say p . Hence, `act` may not be enabled after σ_a . Let p point to b in τ . By induction, we have already constructed σ_b for τ . The ABA is enabled after σ_b . This is due to $\tau \preceq_b \sigma_b$. It implies that p points to b in τ iff p points to b in σ_b (independent of the validity), and likewise for q . That is, the comparison has the same outcome in both computations. Now, we can exploit the absence of harmful ABAs to find a computation mimicking $\tau.act$ for a . Applying Definition 7.18 to $\sigma_b.act$ and σ_a yields some σ'_a that satisfies the required properties.

Second, consider the case of `act` performing a `free(b)`. If `act` is enabled after σ_a nothing needs to be shown. In particular, this is the case if b is a valid address or $a = b$. Otherwise, b must be an invalid address. Freeing an invalid address does not change the valid memory. It also does not change the control location of threads as frees are performed by the environment. Hence, we have $\tau.act \sim \sigma_a$. By the definition of elision support, Definition 7.14iii, the `free` does not affect the behavior of the SMR automaton on other addresses. We get $\tau.act < \sigma_a$. With the same arguments we conclude $\tau.act \preceq_a \sigma_a$. That is, we do not need to mimic frees of invalid addresses.

Last, consider *act* executing an allocation $p := \text{malloc}$ of address b . If b is fresh in σ_a or $a = b$, then *act* is enabled after σ_a . The allocation makes b a valid address. That \prec holds for this address follows from elision support, Definition 7.14ii. As argued earlier, elision support applies to full \mathcal{O} here because there are no double retires on b by assumption: a double retire on b in τ would manifest as a double retire in some $\sigma_b \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{one}}$ with $\tau \preceq_b \sigma_b$ which exists by induction. Consider now the remaining case where *act* is not enabled after σ_a because b is not fresh. We replace in σ_a every occurrence of b with a fresh address c . Let us denote the result with $\sigma_a[b/c]$. Relying on elision support, Definition 7.14i, one can show $\sigma_a < \sigma_a[b/c]$ and thus $\tau < \sigma_a[b/c]$ for all $\prec \in \{\sim, \prec, \preceq_a\}$. Since b is fresh in $\sigma_a[b/c]$, we conclude by enabledness of *act* as in the previous case. ■

From the above follows the overall reduction result. It states that safety properties can be verified under the much simpler semantics which reuses at most a single address. We stress that the result is independent of the actual SMR automaton used.

Theorem 7.21 (Reduction 1). If \mathcal{O} supports elision and $\mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{one}}$ is free from pointer races, double retires, and harmful ABAs, then $\text{good}(\mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}) \iff \text{good}(\mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{one}})$.

We can also prove the absence of double retires in the simpler semantics, as mandated by SMR algorithms in general (cf. Section 2.3) and SMR automaton $\mathcal{O}_{\text{Base}}$ in particular (cf. Section 5.2).

Theorem 7.22. If \mathcal{O} supports elision and $\mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{one}}$ is free from pointer races, double retires, and harmful ABAs, then $\mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$ is free from double retires.

In the next section, we put the results to practice and demonstrate how to verify non-blocking data structures with memory reclamation.

7.5 Evaluation

We propose an automated analysis that is capable of checking linearizability of non-blocking data structures as well as checking compliance of SMR implementations with SMR automaton specifications. The analysis extends the one from Section 6.3 as described in Section 7.5.1. The linearizability benchmarks are discussed in Section 7.5.2. SMR implementations are verified against SMR automata in Section 7.5.3.

7.5.1 Soundness checks

To guarantee that the restriction of reuse to a single address is sound, we have to check for pointer races and harmful ABAs, as demanded by Theorem 7.21. To check for pointer races we rely on the validity information we already integrated in Chapter 6. If a pointer race is detected, verification fails. For this check, we rely on Proposition 7.23 below and deem racy any invocation of *retire* with invalid pointers. That is, the pointer race check boils down

to scanning dereferences and `retire` invocations for invalid pointers. A more general check for racy calls can be implemented by using the technique from Proposition 5.3.

Proposition 7.23. If a call is racy wrt. $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$ or $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$ or $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^{0,1}$, then it is a call to function `retire` with an invalid pointer as its argument.

Next, we add a check for harmful ABAs on top of the state space exploration. This check has to implement Definition 7.18. That a computation $\sigma_a.act$ contains a harmful ABA can be detected in the view v_a for thread t which performs act . Like for computations, the view abstraction v_b of σ_b for t cannot perform the ABA. To prove the ABA harmless, we seek a view v'_b which is similar to v_a , b -aligned to v_b , and includes the SMR behavior of v_b . (The relations introduced in Sections 7.1 to 7.3 naturally extend to views.) If no such v'_b exists, verification fails.

In the thread-modular setting one has to be careful with the choice of v'_b . It is not sufficient to find *just some* v'_b satisfying the desired relations. The reason lies in that we perform the ABA check on a thread-modular abstraction of computations. To see this, assume the view abstraction of σ_b is $\alpha(\sigma_b) = \{v_b, v\}$ where v_b is the view for thread t which performs the ABA in $\sigma_a.act$. For *just some* v'_b it is not guaranteed that there is a computation σ'_b such that $\alpha(\sigma'_b) = \{v'_b, v\}$. The sheer existence of the individual views v'_b and v in V does not guarantee that there is a computation the abstraction of which yields those two views. Put differently, we cannot construct computations from views. The existence of v'_b does not prove the existence of the required σ'_b .

To overcome this problem, we use a method to search for a v'_b that guarantees the existence of σ'_b ; in terms of the above example, guarantees that there is σ'_b with $\alpha(\sigma'_b) = \{v'_b, v\}$. We take the view v_b that cannot perform the ABA. We apply sequential steps to v_b until it *comes back* to the same program location. The rational behind this approach is that ABAs are typically conditionals that restart the operation if the ABA is not executable. Restarting the operation results in reading out pointers anew (this time without interference from other threads, cf. Example 7.19). Consequently, the ABA is now executable. The resulting view is a candidate for v'_b . If it does not satisfy Definition 7.18, verification fails. Although simple, this approach succeeded in all of our benchmarks.

7.5.2 Linearizability Experiments

We implemented the approach presented in this chapter in a C++ tool called `TMREXP`.² We empirically evaluated the tool on Treiber's stack, Michael&Scott's queue, and the DGLM queue. We reiterate from Section 6.3 that the analysis we build on cannot handle sets [Abdulla et al. 2013, 2017]; this is a shortcoming of the original approach, not a shortcoming of the results from the present chapter. As SMR algorithms we considered EBR and HP as specified by the SMR automata $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$ and $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$, respectively. We did not consider FL. The reason for this is that the approach suggested in Chapter 5 and implemented in Chapter 6, namely specifying FL via \mathcal{O}_{Base} and having retired addresses freed, inevitably leads to pointer races (unsafe accesses) and thus to verification failure. We believe that one can generalize and tailor the results presented here to support FL. Such a generalization, however, is beyond the scope of this thesis.

² `TMREXP` is freely available at: <https://wolf09.github.io/phd/>

Table 7.24: Experimental results for verifying singly-linked data structures using SMR. The experiments were conducted on an Intel i5-8600K@3.6GHz with 16GB of RAM using Ubuntu 16.04 and Clang 6.0.

SMR	Program	States	ABAs	Linearizability	ABA Check
EBR	Treiber’s stack	1822	0	16s ✓	0s ✓
	Michael&Scott’s queue	7613	0	2630s ✓	0s ✓
	DGLM queue	27132	0	3754s ✓ ^b	0s ✓
HP	Treiber’s stack	2606	186	19s ✓	0.06s ✓
	Opt. Treiber’s stack	—	—	0.8s ✗ ^a	—
	Michael&Scott’s queue	19028	536	7075s ✓	0.9s ✓
	DGLM queue	41753	2824	7010s ✓ ^b	26s ✓

^a Pointer race due to an ABA in push. The ABA does not affect correctness.

^b Imprecision in the memory abstraction required hinting.

The findings are listed in Table 7.24. They include (i) the size of the explored state space, i.e., the number of reachable views, (ii) the number of ABA prone views, i.e., views where a thread is about to perform an assumption containing an invalid pointer, (iii) the running time and result of verification, i.e., the exhaustive exploration of the state space and linearizability check, and (iv) the running time and result of proving the absence of harmful ABAs. We mark tasks with ✓ if they were successful and with ✗ if they failed. All experiments were conducted on an Intel i5-8600K@3.6GHz with 16GB of RAM using Ubuntu 16.04 and Clang 6.0.

Our approach is capable of verifying non-blocking data structures using HP and EBR. We were able to automatically verify Treiber’s stack, Michael&Scott’s queue, and the DGLM queue. To the best of our knowledge, we are the first to verify data structures using the aforementioned SMR algorithms fully automatically. Moreover, we are also the first to verify automatically the DGLM queue under any manual memory management technique.

An interesting observation throughout the entire test suite is that the number of ABA prone views is rather small compared to the total number of reachable views. Consequently, the time needed to check for harmful ABAs is insignificant compared to the verification time. This substantiates the usefulness of *ignoring* ABAs during the actual analysis and checking afterwards that no harmful ABA exists.

Our tool could not establish linearizability for the optimized version of Treiber’s stack with hazard pointers. The reason for this is that the push operation does not use any hazard pointers. This leads to pointer races and thus verification failure although the implementation is correct. To see why, consider the following excerpt of push, repeated from Figure 2.12:

```

266 Node* top = ToS;
267 node->next = top;
268 if (CAS(&ToS, top, node))
269     break

```


The operation reads the top-of-stack pointer into a local variable `top` in Line 266, links the newly allocated node to the top-of-stack in Line 267, and swings the top-of-stack pointer to the new node in Line 268. Between Line 266 and Line 268 the node referenced by `top` can be popped, reclaimed, reused, and reinserted by an interferer. Consequently, the CAS in Line 268 is ABA prone. The reclamation of the node referenced by `top` renders both `top` and `node->next` invalid. As specified by Definition 6.5, the comparison of the valid ToS with the invalid `top` in the CAS from Line 268 makes `top` valid again. However, `node->next` remains invalid. That is, the push succeeds and leaves the stack in a state with `ToS->next` being invalid. This leads to pointer races because no thread can acquire valid pointers to the nodes following ToS. Hence, reading out data of such subsequent nodes in the pop procedure, for example, raises a pointer race.

To solve this issue, the CAS in Line 268 has to validate the pointer `node->next`. One could annotate the CAS with an invariant `Tos == node->next`. Treating invariants and assumptions alike would result in the CAS validating `node->next`. That the annotation is an invariant indeed, could be checked during the analysis. We consider a proper investigation as future work.

For the DGLM queue, our tool required hints. The DGLM queue is similar to Michael&Scott's queue but allows the Head pointer to overtake the Tail pointer by at most one node. Due to imprecision in the memory abstraction, our tool explored states with malformed lists where Head overtook Tail by more than one node. We implemented a switch to increase the precision of the abstraction and ignore cases where Head overtakes Tail by more than one node. This simple hint made verification of the DGLM queue possible. While this change is ad hoc, it does not jeopardize the principledness of our approach because it affects only the memory abstraction which we took from the literature.

7.5.3 Verifying SMR Implementations

It remains to verify that a given SMR implementation is correct wrt. an SMR automaton \mathcal{O} . As noted in Chapter 5, an SMR implementation can be viewed as a non-blocking data structure where the stored *data* are pointers. So we can reuse the above analysis. We extended our tool `TMREXP` with an abstraction for (sets of) data values.³ The main insight for a concise abstraction is that it suffices to track a single SMR automaton state per view. If the SMR implementation is not correct wrt. \mathcal{O} , then by definition there is $\tau \in \mathcal{O} \llbracket MGC(R) \rrbracket_{Adr}^{Adr}$ with $\mathcal{H}(\tau) \notin \mathcal{S}(\mathcal{O})$. Hence, there must be some state s with $\mathcal{H}(\tau) \notin \mathcal{S}(s)$. Consider the specifications $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$ and $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$. There, state s is of the form $s = (l, \varphi)$ with $\varphi = \{z_t \mapsto t, z_a \mapsto a\}$. State s induces an abstraction of data values d : either $d = a$ or $d \neq a$. Similarly, an abstraction of sets of data values simply tracks whether or not the set contains a .

To gain adequate precision, we retain in every view the thread-local pointers of the thread t that violates the specification, $t = \varphi(z_t)$. With respect to the HP implementation from Figure 2.8, this keeps the thread- t -local `HPRec` in every view. It makes the analysis recognize that t has indeed protected a . Moreover, we store in every view whether or not the last `retire` invocation stems from the thread of that view. With this information, we avoid

³ `TMREXP` is freely available at: <https://wolff09.github.io/phd/>

Table 7.25: Experimental results for verifying SMR implementations against their SMR automaton specifications. The experiments were conducted on an Intel i5-8600K@3.6GHz with 16GB of RAM using Ubuntu 16.04 and Clang 6.0.

SMR Implementation	Specification	States	Verification
Hazard Pointers	$\mathcal{O}_{Base} \times \mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$	5437	1.5s ✓
Hazard Pointers	$\mathcal{O}_{Base} \times \mathcal{O}_{HP}^{0,1}$	5304	1.5s ✓
Epoch-Based Reclamation	$\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$	11528	11.2s ✓

unnecessary matches during interference of views v_1 and v_2 : if both threads t_1 of v_1 and t_2 of v_2 have performed the last `retire` invocation, then t_1 and t_2 are the exact same thread. Hence, interference is not needed as threads have unique identities. We found this extension necessary to gain the precision required to verify our benchmarks.

Table 7.25 shows the experimental results for verifying the implementations of EBR from Figure 2.7 and HP with two hazard pointers per thread from Figure 2.8. Both SMR implementations allow threads to dynamically join and part. We conducted the experiments in the same setup as before. Our experiments reveal that the verification of SMR implementations is simpler and more efficient than verifying non-blocking data structures using SMR. This is unsurprising since, as discussed in Section 2.3, SMR implementations do not reclaim the memory they use internally for bookkeeping.

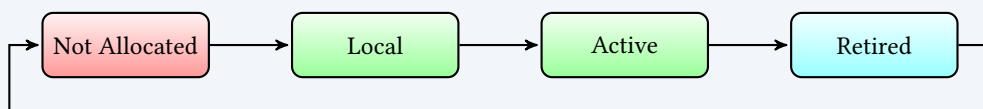
Strong Pointer Races

The reduction result from Chapter 7 has demonstrated that large parts of the state space can be ignored during an analysis. Soundness of the result crucially relies on a pointer race check and an ABA check. Upon a closer inspection, one may presume that the ABA check holds back the reduction result. Indeed, one can establish pointer race freedom and verify the program under scrutiny ignoring reallocations altogether, i.e., under $\mathcal{O}[\llbracket P \rrbracket_{Adr}^\emptyset]$. The ABA check, however, mandates an analysis of $\mathcal{O}[\llbracket P \rrbracket_{Adr}^{one}]$. As we have seen, dealing with deletions and reallocations is notoriously difficult and expensive—this was the very reason for our endeavor in Chapter 7.

In this chapter, we utilize the full potential of the reduction from Chapter 7. We show that the actual verification can be conducted under the garbage collected semantics $\llbracket P \rrbracket_\emptyset^\emptyset$, using off-the-shelf GC verifiers. To avoid an expensive state space exploration (a semantic analysis) of a semantics larger than $\llbracket P \rrbracket_\emptyset^\emptyset$, we present a type system a successful type check of which guarantees $\mathcal{O}[\llbracket P \rrbracket_{Adr}^\emptyset$ to be free from pointer races and harmful ABAs, as required by the reduction result. We stress that the type check is syntax-centric—it need not explore the state space of $\mathcal{O}[\llbracket P \rrbracket_{Adr}^\emptyset$. To enable such a type check, we deem ABAs unsafe, whether harmless or not. This limits applicability: allowing for harmless ABAs was motivated by real-world implementations. To support these implementations nevertheless, we employ the theory of movers [Lipton 1975] as an enabling technique.

The idea behind our type system is a memory life cycle common to non-blocking data structures using SMR [Brown 2015]. The life cycle, depicted in Figure 8.1, has four stages: (i) local, (ii) active, (iii) retired, and (iv) not allocated. Newly allocated objects are in the local stage. The object is known only to the allocating thread; it has exclusive read/write access. The goal of the local stage is to prepare objects for being published. When an object is published, it enters the active stage. In this stage, accesses to the object are safe because it is guaranteed to be allocated. However, no thread has exclusive access and thus must fear interference by others. It is worth pointing out that a publication is irreversible. Once an object becomes active it cannot become local again. A thread, even if it removes the active object from the shared structures, must account for other threads that have already acquired a pointer to that object. Removed objects are eventually retired. Depending on the SMR algorithm, retired objects may still be safely accessible. Finally, the SMR algorithm reclaims retired objects. Then, the memory can be reused and the life cycle begins anew.

Figure 8.1: Memory life cycle of objects in non-blocking data structures using SMR.



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 objects while they are active such that the SMR guarantees safe access even after the object is retired. To cope with this, our types integrate knowledge about the SMR algorithm. A core aspect of our development is that the actual SMR algorithm is an input to our type system—it is not tailored towards a specific SMR algorithm.

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

In Chapter 2 we have already seen that detecting activeness of objects is non-trivial. Between acquiring a pointer to the object and the protection, an interferer may retire the object and thus void the protection. SMR algorithms usually offer no means to check whether or not a protection was successful. Instead, programmers perform this check by exploiting intricate data structure invariants, like *all shared reachable objects are active*. A type system, however, 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 light-weight annotations to mark pointers that point to active objects. To relieve the programmer from arguing about their correctness, we automate the correctness check. Interestingly, this can be done with off-the-shelf GC verifiers. It is worth pointing out that the ability to automatically refute incorrect annotations allows for an automated guess-and-check approach for placing invariant annotations [Flanagan and Leino 2001].

The remainder of this chapter is structured as follows: Section 8.1 introduces invariant annotations, Section 8.2 presents the generalized reduction result, Section 8.3 presents the type system, Section 8.4 applies the type system to an example, Section 8.5 shows how to verify annotations, Section 8.6 gives a type inference algorithm, Section 8.7 discusses movers, and Section 8.8 evaluates the approach.

8.1 Annotations

We introduce invariant annotations to guide the type check. An annotation construct that is new to our model are angels, ghost (auxiliary) variables [Owicki and Gries 1976] with an angelic semantics. Like for ghosts, their purpose is verification: angels store information about the computation that can be used in invariants but that cannot be used to influence the control flow. This information is a set of addresses, which means angels are second-order pointers. The set of addresses is determined by an angelic choice, a non-deterministic assignment that is beneficial for the future of the computation.

The idea behind angels is the following. Consider EBR’s `leaveQ` function. It guarantees that the potentially infinitely many addresses accessible during a non-quiescent phase 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

Figure 8.2: Formula encoding the correctness of invariant annotations in a computation τ .

$inv(\tau) ::= inv_\epsilon(\tau)$	
$inv_\sigma(\epsilon) ::= \text{true}$	
$inv_\sigma(act.\tau) ::= \exists r. inv_{\sigma.act}(\tau)$	if $act = \langle t, @\text{inv angel } r, \emptyset \rangle$
$inv_\sigma(act.\tau) ::= m_\sigma(p) = m_\sigma(q) \wedge inv_{\sigma.act}(\tau)$	if $act = \langle t, @\text{inv } p = q, \emptyset \rangle$
$inv_\sigma(act.\tau) ::= m_\sigma(p) \in r \wedge inv_{\sigma.act}(\tau)$	if $act = \langle t, @\text{inv } p \text{ in } r, \emptyset \rangle$
$inv_\sigma(act.\tau) ::= m_\sigma(p) \in \text{active}(\sigma) \wedge inv_{\sigma.act}(\tau)$	if $act = \langle t, @\text{inv active}(p), \emptyset \rangle$
$inv_\sigma(act.\tau) ::= r \subseteq \text{active}(\sigma) \wedge inv_{\sigma.act}(\tau)$	if $act = \langle t, @\text{inv active}(r), \emptyset \rangle$
$inv_\sigma(act.\tau) ::= inv_{\sigma.act}(\tau)$	otherwise.

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 from Chapter 5, we generalize the set of commands as follows:

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

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

In the SMR semantics $\mathcal{O}[[P]]_X^Y$, the new commands do not lead to memory updates:

(Invariant) If $act = \langle t, @\text{inv } \bullet, \emptyset \rangle$.

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 8.2. There, $\text{active}(\sigma)$ is the set of addresses that are neither freed nor retired after computation σ .¹ 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. With this understanding, we let memories evaluate angels: $m_\tau(r)$ gives the largest set of addresses that $inv(\tau)$ allows for. It remains to verify annotations. Section 8.5 shows how to automatically prove the correctness of invariant annotations for all computations.

¹ In Section 8.3 we will additionally require *active* addresses to be allocated, as suggested by Figure 8.1. The type system will take care of the discrepancy and ensure that addresses are allocated whenever it relies on $\text{active}(\bullet)$ annotations. The existence of a pointer to a non-freed address, for instance, guarantees that the address is allocated (and not fresh). Skipping the allocation check in the encoding of invariants here makes the check simpler and more widely applicable.

8.2 Avoiding All Reallocations

Our goal is to strengthen the reduction from Chapter 7 such that verification of full $\mathcal{O}[\llbracket P \rrbracket_{Adr}^{Adr}]$ becomes possible under $\llbracket P \rrbracket_{\emptyset}^{\emptyset}$. Recall that the soundness of the previous reduction crucially relied on a semantics exploring reallocations (of a single address) in order to detect harmful ABAs, i.e., assumptions involving invalid pointers. In order to avoid reallocations altogether, we forbid ABA prone assumptions in addition to pointer race freedom. Section 8.7 will bridge the gap to implementations that perform harmless ABAs, like the ones we have seen in Chapter 7.

In order to detect ABA prone assumptions in a computation τ , we need a lookahead of commands that could be executed next, whether or not they are actually enabled. This contrasts our approach from Chapter 7. There, we were guaranteed that every (first) ABA is enabled and thus executed in a computation, provided we chose an appropriate address for reallocation. Now, without reallocating any addresses, it is no longer guaranteed that the ABA is enabled and thus appears in a computation from $\llbracket P \rrbracket_{\emptyset}^{\emptyset}$. Hence, we do not check the actually executed commands but those that the control-flow could choose next. Formally, we say that a command com is control-flow-enabled after τ , denoted by $com \in next-com(\tau)$, if there is $pc \in ctrl(\tau)$ with $pc(t) \xrightarrow{com} \bullet$ for some thread t . Then, τ is prone to an unsafe assumption if an ABA prone assumption is control-flow-enabled.

Definition 8.3 (Unsafe Assumption). A computation τ is prone to an unsafe assumption if there is assume $p = q \in next-com(\tau)$ with $p \notin valid_{\tau}$ or $q \notin valid_{\tau}$.

Strong pointer races extend ordinary pointer races, Definition 7.13, with unsafe assumptions.

Definition 8.4 (Strong Pointer Race). A computation $\tau.act$ is a strong pointer race if act performs (i) an ordinary pointer race, or (ii) an unsafe assumption.

Now, we strengthen the reduction result from Theorem 7.20. Relying on the absence of strong pointer races, we do not need to deal with ABAs as they are deemed unsafe and ruled out thus.

Theorem 8.5. Let \mathcal{O} support elision and let $\mathcal{O}[\llbracket P \rrbracket_{Adr}^{\emptyset}]$ be free from strong pointer races and double retires. Then, for all $\tau \in \mathcal{O}[\llbracket P \rrbracket_{Adr}^{Adr}]$ there is some $\sigma \in \mathcal{O}[\llbracket P \rrbracket_{Adr}^{\emptyset}]$ such that $\tau \sim \sigma$, $\tau < \sigma$, and $retired_{\tau} \subseteq retired_{\sigma}$.

Besides similarity and SMR behavior inclusion, the theorem yields $retired_{\tau} \subseteq retired_{\sigma}$. We rely on this inclusion to establish under GC that P does not perform double retires, as required for a meaningful application of SMR automaton \mathcal{O}_{Base} . The proof of the theorem is analogous to the one of Theorem 7.20.

Next, we remove free commands. We simply strip them away from the computations of $\mathcal{O}[\llbracket P \rrbracket_{Adr}^{\emptyset}]$. The result are GC computations from $\llbracket P \rrbracket_{\emptyset}^{\emptyset}$. Interestingly, the resulting GC computation allows to draw conclusions about the annotations in the original computation. We exploit this in Section 8.5 in order to discharge invariants under GC.

Theorem 8.6. If $\sigma \in \mathcal{O}[\llbracket P \rrbracket_{Adr}^{\emptyset}]$ is free from strong pointer races, then there is some $\gamma \in \llbracket P \rrbracket_{\emptyset}^{\emptyset}$ such that $ctrl(\sigma) = ctrl(\gamma)$ and $m_{\sigma}|_{valid_{\sigma}} = m_{\gamma}|_{valid_{\sigma}}$ and $inv(\gamma) \implies inv(\sigma)$.

Proof Sketch. To see the theorem, we proceed in two steps. First, we remove `env` commands from σ . Since their only effect on the computations is an update of selectors of invalid addresses, strong pointer race freedom guarantees that no thread can observe the update: accessing the invalid address requires an invalid pointer and thus raises a (strong) pointer race. We obtain an intermediate σ' that satisfies $\sigma \sim \sigma'$ and $\text{inv}(\sigma) \iff \text{inv}(\sigma')$. Next, we remove `free` commands from σ' and arrive at γ . As no memory is reused in $\mathcal{O}[\![P]\!]_{\text{Adr}}^\emptyset$, all allocations remain enabled. The remaining commands ignore \mathcal{O} , so they remain enabled as well. The only consequence of the removal is that no expressions are invalidated and previously freed addresses remaining retired. The former means $m_\sigma|_{\text{valid}_\sigma} = m_\gamma|_{\text{valid}_\sigma}$ where m_γ is restricted to the valid expressions of σ rather than γ . The latter establishes that invariant violations in γ carry over to σ . ■

Finally, we arrive at the overall reduction result which allows for verification under GC, the simplest semantics a tool can assume.

Theorem 8.7 (Reduction 2). If \mathcal{O} supports elision and $\mathcal{O}[\![P]\!]_{\text{Adr}}^\emptyset$ is free from strong pointer races and double retires, then $\text{good}(\mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}) \iff \text{good}([\![P]\!]_\emptyset^\emptyset)$ and $\mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$ is free from double retires.

We turn towards checking $\mathcal{O}[\![P]\!]_{\text{Adr}}^\emptyset$ for strong pointer races and double retires.

8.3 A Type System to Prove Strong Pointer Race Freedom

We present a type system a successful type check of which entails strong pointer race freedom as required by Theorem 8.7. The guiding idea of our types is to under-approximate the validity of pointers. To achieve this, our types incorporate the SMR algorithm in use and the guarantees it provides. It does so in a modular way: a parameter of the type system definition is an SMR automaton specifying the SMR algorithm.

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

Throughout the remainder of the section we fix an SMR automaton \mathcal{O} relative to which we describe the type system. We assume that \mathcal{O} contains exactly two variables z_t and z_a . Intuitively, z_t stores the thread for which \mathcal{O} tracks the protection of the address stored in z_a . All SMR algorithms we are aware of can be specified with only two variables. A possible explanation is that SMR algorithms do not seem to use helping [Herlihy and Shavit 2008, Section 6.4] to protect pointers.

Assumption 8.8. SMR automata have two variables z_t resp. z_a tracking a thread resp. an address.

We believe that the results presented hereafter can be generalized to SMR automata with more variables and consider a closer investigation of the matter as future work.

8.3.1 Guarantees

Towards a definition of our type system, recall the memory life cycle from Figure 8.1. The transition from the active to the retired stage requires care. The type system has to detect that a thread is guaranteed safe access to a retired node. This means finding out that an SMR protection was successful. Additionally, types need to be stable under interference. Nodes can be retired without a thread noticing. Hence, types need to ensure that the guarantees they provide cannot be spoiled by actions of other threads.

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

- \mathbb{L} : Thread-local pointers referencing nodes in the local stage. The guarantee comes with two more properties. There are no valid aliases of the pointer and the referenced node is not retired. This gives the thread holding the pointer exclusive access.
- \mathbb{A} Pointers to nodes in the active stage. Active pointers are guaranteed to be valid, they can be accessed safely.
- \mathbb{S} Pointers to nodes which are protected by the SMR algorithm from being reclaimed. Such pointers can be accessed safely although the referenced node might be in the retired stage.
- \mathbb{E}_L SMR-specific guarantee that depends on a set of locations in the given SMR automaton. The idea is to track the history of SMR calls performed so far. This history is guaranteed to reach a location in L . The information about L bridges the (SMR-specific) gap between \mathbb{A} and \mathbb{S} . Accesses to the pointer are potentially unsafe.

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

8.3.2 Types

The input SMR automaton \mathcal{O} induces a set of intersection types [Coppo and Dezani-Ciancaglini 1978; Pierce 2002, Section 15.7] defined by the following grammar:

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

The meaning of 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 $T \cap \{\mathbb{S}, \mathbb{L}, \mathbb{A}\} \neq \emptyset$. The three guarantees serve as syntactic under-approximations of the semantic notion of validity.

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

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 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 \emptyset 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 judgments take the form:

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

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. Consequently, 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 the used SMR automaton, $\mathcal{O} = \mathcal{O}_{Base} \times \mathcal{O}_{SMR}$. 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 $SafeLoc(\mathcal{O})$ to be the largest set of locations in \mathcal{O} that is closed under interference from other threads and for which all transition $l \xrightarrow{\text{free}(a), g} l'$ with $l \in SafeLoc(\mathcal{O})$ and $g \wedge a = z_a$ satisfiable lead to an accepting location l' . Guarantee \mathbb{A} is characterized by location L_2 in \mathcal{O}_{Base} . Technically, location L_2 does not imply activeness of address z_a . It implies a strictly weaker property, namely that z_a is not retired. Then, z_a is active only if it is allocated. However, SMR automata cannot detect whether or not z_a is allocated at the moment (there is no event for `malloc`). Hence, we separately ensure that z_a is allocated, and thus active indeed, whenever \mathbb{A} is assigned. Along the same lines, we also use location L_2 for \mathbb{L} . The discussion yields the following definition.

Definition 8.9 (Meaning of Types). The locations associated with types T , $Loc(T)$, are:

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

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 capture those transformations with the *type transformer relation* $T, x, com \rightsquigarrow T'$. To make it precise, we first define the post-image $post_{p,com}(L)$ on the locations of the SMR automaton. The post-image yields a set of locations L' reachable by taking a com -labeled transition from L . The considered transition is restricted in two ways. First, its guard g must allow z_t to track thread t executing com . Second, if p appears as a parameter in com , then guard g must allow z_a to track p . Formally, these requirements translate to the satisfiability of $g \wedge t = z_t$ and $g \wedge p = z_a$, respectively. The parameterization in p makes the post-image precise. To see this, consider \mathcal{O}_{Base} and the command $com = \text{in:retire}(p)$. 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 . Now, we are ready to formalize the type transformer relation.

Definition 8.10 (Type Transformer). The type transformer relation $T, x, com \rightsquigarrow T'$ is defined by the following conditions:

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

The over-approximation of the post-image is the first inclusion. The implication states that SMR commands cannot validate pointers. We can, however, deduce from the fact that the address has not been retired (\mathbb{A} or \mathbb{L}) and the SMR command has been executed, that it is safe to access the address (\mathbb{S}). The last inclusion states that SMR commands cannot establish the guarantees \mathbb{L} and \mathbb{A} . It is worth pointing out that the relation $T, x, com \rightsquigarrow T'$ only depends on the SMR automaton, up to a choice of variable names. This means we can tabulate it to guarantee quick access when typing a program. We also write $\Gamma, com \rightsquigarrow \Gamma'$ if we have $\Gamma(x), x, com \rightsquigarrow \Gamma'(x)$ for all pointers/angels x . We write $\Gamma \rightsquigarrow \Gamma'$ if we take the post-image to be the identity. For an example, refer to Section 8.4.1.

Guarantees \mathbb{L} and \mathbb{A} are special in that their sets of locations, $Loc(\mathbb{L})$ and $Loc(\mathbb{A})$, are not closed under interference. For \mathbb{L} , the type rules ensure interference freedom. They do so by enforcing that `retire` is not invoked with invalid pointers. Hence, the fact that \mathbb{L} -pointers have no valid aliases implies that other threads cannot retire them. So \mathcal{O}_{Base} remains in L_2 no matter the interference. For \mathbb{A} , the type rules account for interference. We define an operation $rm(\Gamma)$ that takes an environment and removes all \mathbb{A} guarantees for thread-local pointers and angels:

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

The operation also has an effect on shared pointers and angels where it removes all guarantees. The reasoning is as follows. 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/angel receives the empty set of guarantees.

8.3.3 Type Rules

The type rules of our type system are given in Figures 8.12 and 8.13. 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 a type environment Γ so that $\vdash \{\Gamma_{init}\} stmt \{\Gamma\}$. A program P *type checks* if $\vdash P$. Soundness will show that a type check entails the absence of strong pointer races and double retires.

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

Assumption 8.11. Programs adhere to the following restricted syntax:

$$stmt ::= stmt; stmt \mid stmt \oplus stmt \mid stmt^* \mid \text{beginAtomic}; stmt; \text{endAtomic} \\ \mid \text{beginAtomic}; com; \text{endAtomic} .$$

The rules for primitive commands, Figure 8.12, that are not related to SMR are straightforward. Rule (SKIP) has no effect on the type environment. Allocations grant the target pointer the \mathbb{L} guarantee, Rule (MALLOC). 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. Assumptions of the form $p = q$ check that both pointers are valid and join the type information, Rule (ASSUME1). Guarantee \mathbb{L} is removed due to the alias. All other assumptions have no effect on the type environment, Rule (ASSUME2). Similarly, Rule (EQUAL) joins type information in the case of assertions. However, no validity check is performed and \mathbb{L} is not removed. Rule (ACTIVE) adds the \mathbb{A} guarantee. Note that x is a pointer or an angel. Angels are always local variables. Their allocation does not justify any guarantees, in particular not \mathbb{L} , as they hold full sets of addresses, Rule (ANGEL). We can also assert membership of an address held by a pointer in a set of addresses held by an angel, Rule (MEMBER).

SMR-related commands may change the entire type environment, rather than manipulating only the pointers that occur syntactically in the command. This is because of pointer aliasing on the one hand, and because of the SMR automaton on the other hand (for instance, `enterQ` has an effect on all pointers). The post type environment of Rules (ENTER) and (EXIT) simply infers guarantees wrt. the pre type environment and the emitted event. Note that this is the only way to infer SMR-specific guarantees \mathbb{E}_L , i.e., these guarantees solely depend on the SMR commands. Moreover, Rule (ENTER) performs a strong pointer race check. We define predicate $\text{SafeCall}(\Gamma, func(\bar{r}))$ to hold iff the command `in:func(\bar{r})` is guaranteed not to be a racy call given the types in Γ . The formalization coincides with

Figure 8.12: Type rules for primitive commands.

(SKIP)	(MALLOC)	(ASSIGN1)
$\frac{}{\{\Gamma\} \text{skip} \{\Gamma\}}$	$\frac{p \notin \text{shared} \quad T = \{\mathbb{L}\}}{\{\Gamma, p\} p := \text{malloc} \{\Gamma, p : T\}}$	$\frac{T' = T \setminus \{\mathbb{L}\}}{\{\Gamma, p, q : T\} p := q \{\Gamma, p : T', q : T'\}}$
(ASSIGN2)	(ASSIGN3)	(ASSIGN4)
$\frac{\Gamma(q) = T \quad \text{isValid}(T)}{\{\Gamma, p\} p := q.\text{next} \{\Gamma, p : \emptyset\}}$	$\frac{\Gamma(p) = T \quad \text{isValid}(T) \quad T'' = T' \setminus \{\mathbb{L}\}}{\{\Gamma, q : T'\} p.\text{next} := q \{\Gamma, q : T''\}}$	$\frac{}{\{\Gamma\} u := \text{op}(\bar{u}) \{\Gamma\}}$
(ASSIGN5)	(ASSIGN6)	
$\frac{\Gamma(q) = T \quad \text{isValid}(T)}{\{\Gamma\} u := q.\text{data} \{\Gamma\}}$	$\frac{\Gamma(p) = T \quad \text{isValid}(T)}{\{\Gamma\} p.\text{data} := u \{\Gamma\}}$	
(ASSUME1)	(ASSUME2)	
$\frac{\text{isValid}(T) \quad \text{isValid}(T') \quad T'' = (T \wedge T') \setminus \{\mathbb{L}\}}{\{\Gamma, p : T, q : T'\} \text{assume } p = q \{\Gamma, p : T'', q : T''\}}$	$\frac{\text{cond} \not\equiv p = q}{\{\Gamma\} \text{assume cond} \{\Gamma\}}$	
(EQUAL)	(ACTIVE)	
$\frac{T'' = T \wedge T'}{\{\Gamma, p : T, q : T'\} @\text{inv } p = q \{\Gamma, p : T'', q : T''\}}$	$\frac{T' = T \wedge \{\mathbb{A}\}}{\{\Gamma, x : T\} @\text{inv active}(x) \{\Gamma, x : T'\}}$	
(ANGEL)	(MEMBER)	
$\frac{r \notin \text{shared}}{\{\Gamma, r\} @\text{inv angel } r \{\Gamma, r : \emptyset\}}$	$\frac{\Gamma(r) = T' \quad T'' = T \wedge T'}{\{\Gamma, p : T\} @\text{inv } p \text{ in } r \{\Gamma, p : T''\}}$	
(ENTER)	(EXIT)	
$\frac{\text{SafeCall}(\Gamma, \text{func}(\bar{r})) \quad \Gamma, \text{in}:\text{func}(\bar{r}) \rightsquigarrow \Gamma' \quad \text{func}(\bar{r}) \equiv \text{retire}(p) \implies \mathbb{A} \in \Gamma(p)}{\{\Gamma\} \text{in}:\text{func}(\bar{r}) \{\Gamma'\}}$	$\frac{\Gamma, \text{re}:\text{func} \rightsquigarrow \Gamma'}{\{\Gamma\} \text{re}:\text{func} \{\Gamma'\}}$	

the one of racy calls from before, Definition 7.12, except that it replaces the actual validity valid_τ in a computation τ by the under-approximation $\text{isValid}(\bullet)$. A special case of Rule (ENTER) is the invocation of $\text{retire}(p)$, which requires the argument p to be active. This prevents both racy retires and double retires.

The rules for statements are given in Figure 8.13. Rule (INFER) allows for type transformations at any point, in particular, to establish the proper pre/post environments for Rules (CHOICE) and (LOOP). Entering an atomic block, Rule (BEGIN), has no effect on the type environment. Exiting an atomic block allows for interference. Hence, Rule (EXIT) removes any type information from the type environment that can be tampered with by other threads. Sequences of statements are straightforward, Rule (SEQ). Choices require a common pre and post type environment, Rule (CHOICE). Loops require a type environment that is stable under the loop body, Rule (LOOP).

Figure 8.13: Type rules for statements.

$\frac{(\text{INFER}) \quad \Gamma_1 \rightsquigarrow \Gamma_2 \quad \{\Gamma_2\} \text{stmt} \{\Gamma_3\} \quad \Gamma_3 \rightsquigarrow \Gamma_4}{\{\Gamma_1\} \text{stmt} \{\Gamma_4\}}$	$\frac{(\text{BEGIN})}{\{\Gamma\} \text{beginAtomic} \{\Gamma\}}$	$\frac{(\text{END})}{\{\Gamma\} \text{endAtomic} \{rm(\Gamma)\}}$
$\frac{(\text{SEQ}) \quad \{\Gamma\} \text{stmt}_1 \{\Gamma'\} \quad \{\Gamma'\} \text{stmt}_2 \{\Gamma''\}}{\{\Gamma\} \text{stmt}_1; \text{stmt}_2 \{\Gamma''\}}$	$\frac{(\text{CHOICE}) \quad \{\Gamma\} \text{stmt}_1 \{\Gamma'\} \quad \{\Gamma\} \text{stmt}_2 \{\Gamma'\}}{\{\Gamma\} \text{stmt}_1 \oplus \text{stmt}_2 \{\Gamma'\}}$	$\frac{(\text{LOOP}) \quad \{\Gamma\} \text{stmt} \{\Gamma\}}{\{\Gamma\} \text{stmt}^* \{\Gamma\}}$

8.3.4 Soundness

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

Soundness states that a type environment annotating a program point approximates the history of every computation reaching this point. Moreover, $isValid(\bullet)$ approximates validity. To make this precise, we define the straight-line version $\text{stmt}(\tau, t)$ of program P induced by τ and t . It is obtained by projecting τ to the commands of thread t . Furthermore, we define the relation $\models \{\Gamma_{init}\} \text{stmt}(\tau, t) \{\Gamma\}$. It requires that (i) $\tau, t \models \Gamma$ holds and (ii) for every $p : T \in \Gamma$ with $isValid(T)$ we have $p \in \text{valid}_{\tau}$. The soundness result now lifts the syntactic derivation relation \vdash to the semantic soundness relation \models .

Theorem 8.14 (Soundness). For all threads t and all $\tau \in \mathcal{O}[\![P]\!]_{Adr}^{\emptyset}$ with $inv(\tau)$ we have:

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

Proof Sketch. We proceed by induction on the length of $\tau \in \mathcal{O}[\![P]\!]_{Adr}^{\emptyset}$. Let t be a thread with:

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

The induction hypothesis links the current type environment Γ derived for the straight-line program to the semantic information carried by the computation. The hypothesis strengthens the requirements (i) and (ii) in the definition of soundness by the following two conditions, where we assume $\Gamma(x) = T$:

- (iii) If $\mathbb{L} \in T$, then x is a pointer that does not have valid aliases. That is, $m_\tau(x) = m_\tau(q)$ entails that we have $q \notin \text{valid}_\tau$. Note that angels cannot obtain \mathbb{L} according to the type rules.
- (iv) If $\mathbb{A} \in T$, then thread t is in an atomic block.

The interesting argumentation in the induction step is in the case when another thread appends an action, $\tau.act$. It can be summarized as follows. Property (i) continues to hold for $\tau.act$ because the type T of x is closed under interference; for \mathbb{L} and \mathbb{A} we argue separately in the following. If $\mathbb{L} \in T$, then act cannot use a valid alias of x . In particular, it cannot retire x according to the premise of Rule (ENTER). If $\mathbb{A} \in T$, then thread t is in an atomic block and there is no chance to append action act of another thread. The case does not occur. Consider property (ii). Assume $\text{isValid}(T)$ holds. That is, T contains one of $\mathbb{A}, \mathbb{L}, \mathbb{S}$. If $\mathbb{L} \in T$ or $\mathbb{A} \in T$, then the above reasoning for (i) already implies (ii). Otherwise, we have $\mathbb{S} \in T$. It implies (ii) because \mathbb{S} is closed under interference. Property (iii) follows from the fact that act cannot contain, and thus cannot create, a valid alias of x . Lastly, to conclude Property (iv), note that another thread cannot append an action while t is inside an atomic block. ■

The first consequence of soundness is that a successful type check implies strong pointer race freedom. Phrased differently, the rules from Figures 8.12 and 8.13 allow for a successful typing only if there are no strong pointer races. That is, our type system performs a strong pointer race freedom check indeed.

Theorem 8.15. If $\text{inv}(\mathcal{O}\llbracket P \rrbracket_{\text{Adr}}^\emptyset)$ and $\vdash P$, then $\mathcal{O}\llbracket P \rrbracket_{\text{Adr}}^\emptyset$ is free from strong pointer races.

The theorem gives effective means of checking the premise of Theorem 8.7: discharge the invariant annotations using an off-the-shelf verification tool (Section 8.5) and determine a typing using the proposed type system (Section 8.6).

Proof Sketch. To see the theorem, consider $\tau.act \in \mathcal{O}\llbracket P \rrbracket_{\text{Adr}}^\emptyset$. We focus on the case where the last action is a dereference, say due to command com being $p := q.\text{next}$. The remaining cases in the definition of strong pointer races are similar. We show that the dereference is safe, $q \in \text{valid}_\tau$. Let thread t perform the dereference. Let $\text{stmt}(\tau.act, t) = \text{stmt}; com$ be the induced straight-line program. One can show that if we can derive a typing for the program P , then we can derive one for the induced straight-line program as well:

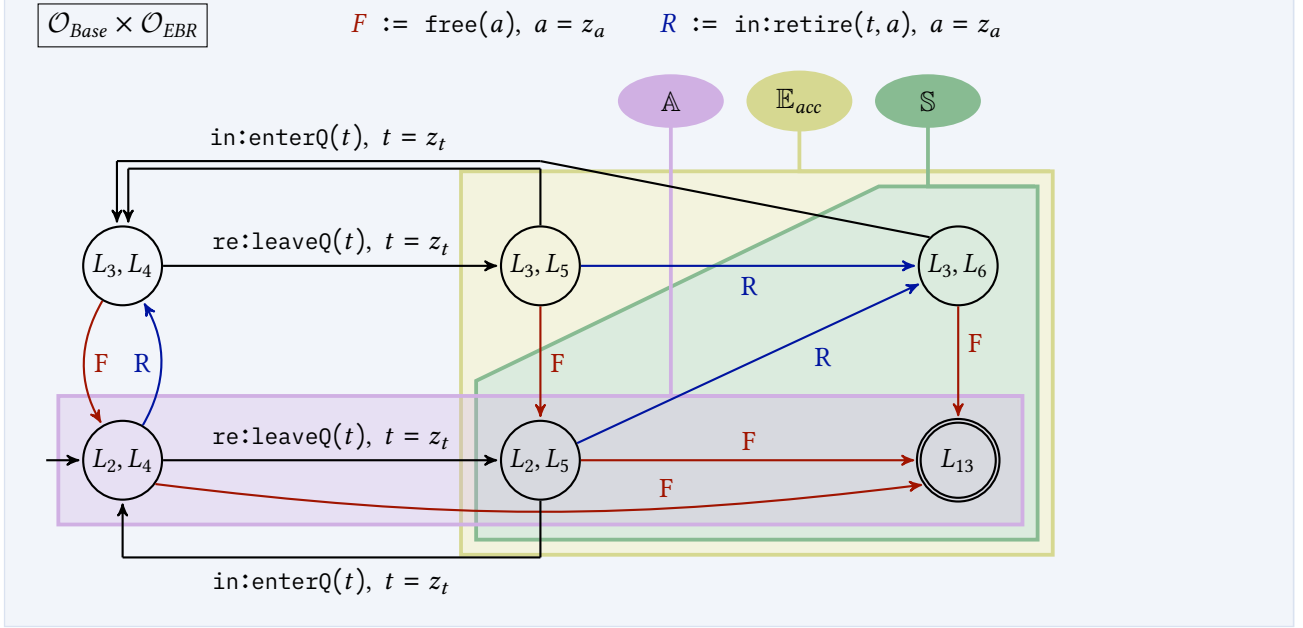
$$\vdash P \quad \text{implies} \quad \exists \Gamma. \vdash \{ \Gamma_{\text{init}} \} \text{stmt}(\tau.act, t) \{ \Gamma \}.$$

The implication should be intuitive. The typing of the overall program can be seen as an intersection over the typings of the induced straight-line programs. Consider some Γ with $\{ \Gamma_{\text{init}} \} \text{stmt}; com \{ \Gamma \}$. The only way to type the sequential composition $\text{stmt}; com$ is Rule (SEQ). It requires a type environment Γ' so that both $\{ \Gamma_{\text{init}} \} \text{stmt} \{ \Gamma' \}$ and $\{ \Gamma' \} com \{ \Gamma \}$ are derivable. The only way to type $p := q.\text{next}$ is Rule (ASSIGN2). By its premise, $\Gamma'(q) = T$ with $\text{isValid}(T)$. Theorem 8.14 yields $q \in \text{valid}_\tau$. The dereference of q is safe. ■

The second consequence of soundness is that a successful type check means that the program does not perform double retires. This is the precondition for a meaningful application of SMR algorithms, and $\mathcal{O}_{\text{Base}}$ in particular.

Theorem 8.16. If $\text{inv}(\mathcal{O}\llbracket P \rrbracket_{\text{Adr}}^\emptyset)$ and $\vdash P$, then $\mathcal{O}\llbracket P \rrbracket_{\text{Adr}}^{\text{Adr}}$ is free from double retires.

Figure 8.17: Cross-product SMR automaton for $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$ and EBR-specific types.



Proof Sketch. The argumentation is along the lines of Theorem 8.15. However, we have to deal with computations from full $\mathcal{O}[P]_{Adr}^{Adr}$. To the contrary, assume there is $\tau.act \in \mathcal{O}[P]_{Adr}^{Adr}$ which performs a double retire on address a . That is, act executes command $retire(p)$ with $m_\tau(p) = a$ and $a \in retired_\tau$. By Theorem 8.15, there are no strong pointer races. Then, Theorem 8.5 yields another computation $\sigma.act \in \mathcal{O}[P]_{Adr}^\emptyset$ such that $\tau \sim \sigma$ and $retired_\tau \subseteq retired_\sigma$. In order to retire p , Rule (ENTER) requires pointer p to hold guarantee \mathbb{A} . This, in turn, means p is valid. We have $m_\sigma(p) = a$. Furthermore, $a \in retired_\tau$ implies $a \in retired_\sigma$ and thus we conclude that \mathcal{O}_{Base} is in location L_3 . This, however, contradicts the activeness of p , which guarantees that \mathcal{O}_{Base} is in location L_2 . ■

The next section gives an in-depth example on how to apply our type system. The two sections thereafter automate the checks in Theorem 8.14: we show how to discharge the invariants $inv(\mathcal{O}[P]_{Adr}^\emptyset)$ with the help of off-the-shelf verification tools for garbage collection and give an efficient algorithm for type inference $\vdash P$.

8.4 Example

We apply our type system to Michael&Scott's queue with EBR (cf. Figure 2.13). Here, a single custom guarantee \mathbb{E}_{acc} is sufficient. We define $Loc(\mathbb{E}_{acc})$ to be those locations where thread z_t is guarantee to have returned from a call to `leaveQ` but has not yet invoked `enterQ`. That is, guarantee \mathbb{E}_{acc} captures when z_t is accessing the data structure. The sets of locations represented by \mathbb{A} , \mathbb{S} , and \mathbb{E}_{acc} can be read of the cross-product SMR automaton $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$ in Figure 8.17. It is worth pointing out that $Loc(\mathbb{S})$ does not contain location (L_2, L_4) . For a set containing (L_2, L_4) to be closed under interference we would need to have (L_3, L_4) in that set. However, (L_3, L_4) allows for a free of z_a and thus must not belong to $Loc(\mathbb{S})$ by definition.

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

8.4.1 Type Transformer Relation

We illustrate the computation of the type transformer relation for command re:leaveQ and the inference of guarantee \mathbb{S} . First, we establish the type transformer relation $\emptyset, x, \text{re:leaveQ} \rightsquigarrow \mathbb{E}_{acc}$. This boils down to checking the inclusion:

$$\text{post}_{x, \text{re:leaveQ}}(\text{Loc}(\emptyset)) \subseteq \text{Loc}(\mathbb{E}_{acc})$$

because the remaining properties of the type transformer relation are trivially satisfied (we do not add any of $\{\mathbb{A}, \mathbb{L}, \mathbb{S}\}$). The empty type corresponds to no knowledge about previously executed SMR commands, which means $\text{Loc}(\emptyset) = L$ with L the set of all locations of $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$ from Figure 8.17. We compute the post-image of L wrt. x and re:leaveQ in $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$. To this end, we consider all transitions labeled with $\text{re:leaveQ}(t)$. The pointer or angel x does not play a role. We derive the desired inclusion as follows:

$$\text{post}_{x, \text{re:leaveQ}}(\text{Loc}(\emptyset)) = \text{post}_{x, \text{re:leaveQ}}(L) = L \setminus \{(L_2, L_4), (L_3, L_4)\} = \text{Loc}(\mathbb{E}_{acc}).$$

Second, we show how to infer \mathbb{S} . From Figure 8.17 we know that \mathbb{E}_{acc} alone does not yield \mathbb{S} because of location (L_3, L_5) ; we also need \mathbb{A} . We establish $\mathbb{E}_{acc} \wedge \mathbb{A} \rightsquigarrow \mathbb{E}_{acc} \wedge \mathbb{A} \wedge \mathbb{S}$. Since $\mathbb{E}_{acc} \wedge \mathbb{A}$ is valid and we do not add \mathbb{L} , the key task is to establish

$$\text{Loc}(\mathbb{E}_{acc} \wedge \mathbb{A}) \subseteq \text{Loc}(\mathbb{E}_{acc} \wedge \mathbb{A} \wedge \mathbb{S}).$$

Because $\text{Loc}(\mathbb{E}_{acc} \wedge \mathbb{A}) \subseteq \text{Loc}(\mathbb{E}_{acc} \wedge \mathbb{A})$ is trivially true, it remains to show that guarantee \mathbb{S} can be obtained indeed, i.e., $\text{Loc}(\mathbb{E}_{acc} \wedge \mathbb{A}) \subseteq \text{Loc}(\mathbb{S})$:

$$\text{Loc}(\mathbb{E}_{acc} \wedge \mathbb{A}) = \text{Loc}(\mathbb{E}_{acc}) \cap \text{Loc}(\mathbb{A}) = \{(L_2, L_5), L_{13}\} \subseteq \{(L_2, L_5), (L_3, L_6), L_{13}\} = \text{Loc}(\mathbb{S}).$$

8.4.2 Angels

To illustrate the use of angels, consider the excerpt of Michael&Scott's dequeue method depicted in Figure 8.18. For the sake of legibility, we omit the enclosing `beginAtomic` and `endAtomic` for all commands. The call to `leaveQ` guarantees that no currently active address is reclaimed until `enterQ` is called. It thus protects an unbounded number of addresses before a thread acquires a pointer to them. Later, when a thread acquired a pointer to such an address in order to access it, the address may no longer be active and thus the type system may not be able to infer \mathbb{S} , as seen in Section 8.4.1 above. To overcome this problem, we use an angel r . Given its angelic semantics, r will capture all addresses that are protected by the `leaveQ` call, Lines 747 to 750. Later, upon accessing/dereferencing a pointer p , we make sure that r captures the address pointed to by p , Lines 754 and 757.

Figure 8.18: Excerpt of the dequeue method from Michael&Scott’s queue with EBR, Lines 314 to 325 from Figure 2.13. To guide the type check, we added annotations involving angel r . The added lines are typeset in bold font.

```

747  @inv angel r;           752  Node* head = Head;           757  @inv next in r;
748  in:leaveQ();             753  Node* tail = Tail;           758  int output = next->data;
749  re:leaveQ();             754  @inv head in r;           759  // ...
750  @inv active(r);         755  Node* next = head->next;       760  in:exitQ();
751  // ...                   756  // ...                       761  re:exitQ();

```

Figure 8.19: A typing for the excerpt of dequeue from Michael&Scott’s queue with EBR introduced in Figure 8.18.

```

762  { Head,head,next,r:∅ }      777  { Head,head,next:∅; r:ℰacc ∧ S }
763  @inv angel r;              778  @inv head in r;
764  { Head,head,next,r:∅ }      779  { Head,next:∅; head,r:ℰacc ∧ S }
765  in:leaveQ();                780  Node* next = head->next;
766  { Head,head,next,r:∅ }      781  { Head,next:∅; head,r:ℰacc ∧ S }
767  re:leaveQ();                782  // ...
768  { Head,head,next:∅; r:ℰacc } 783  @inv next in r;
769  @inv active(r);             784  { Head:∅; next,head,r:ℰacc ∧ S }
770  { Head,head,next:∅; r:ℰacc ∧ A } 785  int output = next->data;
771  { Head,head,next:∅; r:ℰacc ∧ A ∧ S } 786  { Head:∅; next,head,r:ℰacc ∧ S }
772  { Head,head,next:∅; r:ℰacc ∧ S } 787  // ...
773  // ...                      788  in:exitQ();
774  Node* head = Head;          789  { Head,head,next,r:∅ }
775  { Head,head,next:∅; r:ℰacc ∧ S } 790  re:exitQ();
776  // ...                      791  { Head,head,next,r:∅ }

```

Note that, conceptually, we want to execute Lines 749 and 750 atomically so that angel r can precisely capture the addresses active when `leaveQ` returns, Line 749. However, there is no need to introduce this atomic block: the dequeue operation cannot acquire pointers to those addresses that become inactive between Lines 749 and 750 and thus we need not capture them.

8.4.3 Typing

We give a typing for the code from Figure 8.18 in Figure 8.19. Again, we omit the enclosing `beginAtomic` and `endAtomic` of commands for better legibility. We start in Line 762 with type \emptyset for all pointers and the angel r . The allocation of r in Line 763 has no effect on the type assignment. Line 765 invokes `leaveQ`. Again, the types are not affected because the SMR automaton has no transitions labeled with `in:leaveQ`. Next, the invocation returns, Line 767. Following the discussion from Section 8.4.1, we obtain \mathbb{E}_{acc} for r , Line 768. It is worth pointing out that r is treated like an ordinary pointer when it comes to the type transformer relation.

To capture in the type system the set of addresses that can be safely accessed in the subsequent code, we want to lift \mathbb{E}_{acc} of r to \mathbb{S} . We annotate r to hold a set of active addresses, Line 769. This yields type $\mathbb{E}_{acc} \wedge \mathbb{A}$ for r , Line 770. As explained above, we can now lift this type to $\mathbb{E}_{acc} \wedge \mathbb{A} \wedge \mathbb{S}$, Line 771. Recall that the allocation of r in Line 763 is angelic. So the addresses held by r will indeed be chosen to be active. In Line 772, we loose \mathbb{A} since we are not inside an atomic block.

In the subsequent code, we already added annotations (cf. Section 8.4.2) ensuring that dereferenced pointers are captured by the angel r . For instance, Line 778 requires the address of `head` to be captured by r . That this is the case indeed is established when the annotations are discharged. For the typing, we can copy $\mathbb{E}_{acc} \wedge \mathbb{S}$ from r over to `head`. As a consequence, the dereference of `head` in Line 780 is safe. Similarly, we require `next` to be captured by r in Line 783 such that the dereference in Line 785 is safe.

8.4.4 Annotations

We explain our algorithm to automatically add to the program in Figure 2.13 the annotations from Figure 8.18 in order to arrive at the typing in Figure 8.19. We focus on the dereference of `head` in Line 755 (Line 318 in Figure 2.13). Without annotations, the type inference will fail because it cannot conclude that `head` is guaranteed to be valid. To fix this, we implemented a sequence of tactics that we invoke one after the other. If none of them fixes the issue, we give up the type inference and report the failure to the user.

The first tactic simply adds an `@inv active(head)` annotation to Line 755. This makes `head` valid and the type inference go through for Line 755. However, we should only add the annotation if it actually holds. To check this, we employ the technique from Section 8.5. In this particular case, we will find that the annotation does not hold; we try to fix the problem with another tactic.

The second tactic adds an angel r to the (syntactically) most recent `leaveQ` call. We use a template to transform the sequence `in:leaveQ(); re:leaveQ` to the code from Lines 747 to 750. (A subsequent use of this tactic will skip this step and reuse the existing angel.) Then, we fix Line 755 by prepending the annotation `@inv head in r`, Line 754. This makes `head` valid. Whether or not the annotation holds is again checked with the technique from Section 8.5.

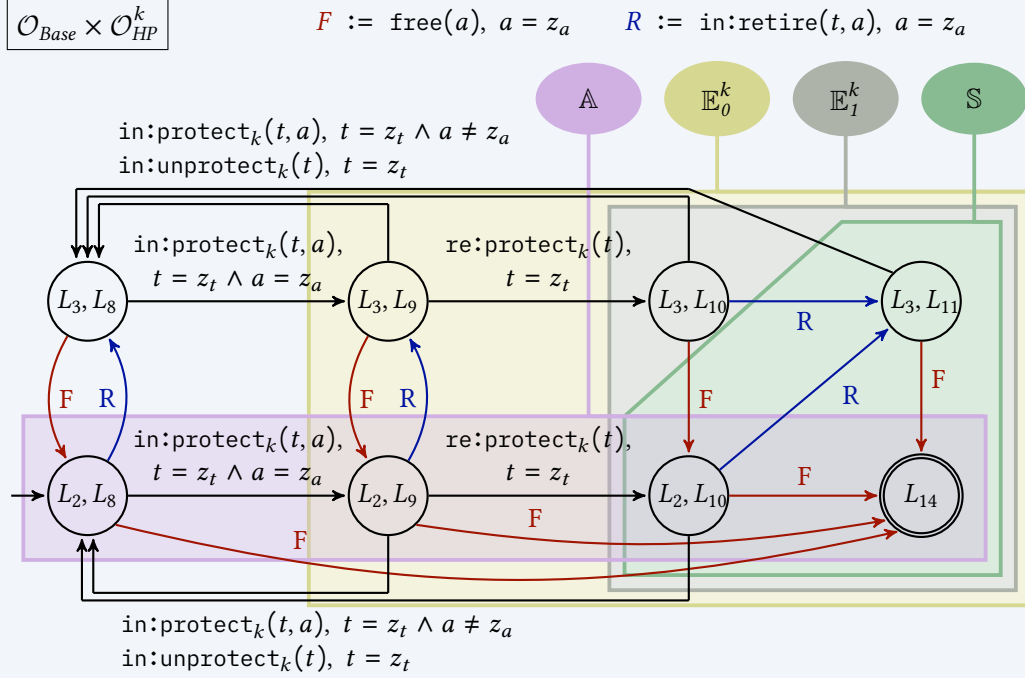
It is worth pointing out that the second tactic is EBR-specific. From our experience, every SMR automaton comes with a small set of tactics that significantly help finding the right annotations—EBR requires the above tactic and HP requires two specific tactics (see Section 8.4.5 below). We do not believe that there is a silver bullet of tactics since SMR algorithms may vary greatly, as seen in the cases of EBR and HP. Theoretically speaking, one could find the annotations by an exhaustive search (finitely many angels will suffice), but this will not scale.

8.4.5 Hazard Pointers

Our approach applies to non-blocking data structures using HP just as well as in the case of EBR. The main difference is that HP typically does not require angels because pointers are protected after they are acquired. Figure 8.20 gives

Figure 8.20: An example of HP-specific types and an application to Micheal&Scott's queue.

(a) Cross-product SMR automaton for $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^k$. The given types are specific to the k -th HP.



(b) A typing for an excerpt of the dequeue function from Michael&Scott's queue with HP, Lines 314 to 325 from Figure 2.13. To guide the type check, we added annotations. The added lines are typeset in bold font. The atomic blocks, Lines 793 to 804 and Lines 815 to 823, can be obtained with the technique from Section 8.7 plus an HP-specific tactic for inserting active annotations.

```

792 { Head,head,next:∅ }
793   beginAtomic;
794 { Head,head,next:∅ }
795   @inv active(Head)
796 { Head:A; head,next:∅ }
797   Node* head = Head;
798 { Head,head:A; next:∅ }
799   in:protect0(head);
800 { Head:A; head:A ∧ E00; next:∅ }
801   re:protect0();
802 { Head:A; head:A ∧ E00 ∧ E10; next:∅ }
803 { Head:A; head:S; next:∅ }
804   endAtomic;
805 { Head,next:∅; head:S }
806 // ...
807 { Head,next:∅; head:S }
808   Node* next = head->next;
809 { Head,next:∅; head:S }

810 { Head,next:∅; head:S }
811   in:protect1(next);
812 { Head:∅; head:S; next:E01 }
813   re:protect1();
814 { Head:∅; head:S; next:E01 ∧ E11 }
815   beginAtomic;
816 { Head:∅; head:S; next:E01 ∧ E11 }
817   @inv active(Head)
818 { Head:A; head:S; next:E01 ∧ E11 }
819   assume(head == Head);
820 { Head:A; head:S; next:E01 ∧ E11 }
821   @inv active(next)
822 { Head:A; head:S; next:S }
823   endAtomic;
824 { Head:∅; head:S; next:S }
825 // ...
826   int output = next->data;
827 { Head:∅; head:S; next:S }

```

an example typing of Michael&Scott’s dequeue method. There, we use $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$ as specification; for the types obtained from $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^{0,1}$ refer to Appendix A.2. As noted above, we use two HP-specific tactics to annotate programs. The first tactic produces candidate locations for active annotations based on the control flow of the program. The rationale behind the tactic is that conditionals that do not restart an operation oftentimes implement consistency checks that ensure activeness. The correctness of candidate annotations is checked with the technique from Section 8.5. The second tactic introduces atomic blocks to ease the type check and is discussed in Section 8.7.

8.5 Invariant Checking

The type system from Section 8.3 relies on invariant annotations 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. More precisely, the premises of Theorems 8.15 and 8.16 require the annotations to be correct under $\mathcal{O}[[P]]_{Adr}^\emptyset$. Interestingly, we can use an off-the-shelf GC verifier to discharge the invariant annotations fully automatically. The following theorem shows that checking invariants under GC, that is, in $[[P]]_\emptyset^\emptyset$, suffices indeed. Technically, we extend Theorem 8.15 because the reduction from Theorem 8.6 requires strong pointer race freedom.

Theorem 8.21. If $inv([[P]]_\emptyset^\emptyset)$ and $\vdash P$, then $inv(\mathcal{O}[[P]]_{Adr}^\emptyset)$ holds and $\mathcal{O}[[P]]_{Adr}^\emptyset$ is free from strong pointer races.

Proof Sketch. Towards a contradiction, assume that the claim does not hold. Then, there exists a shortest computation $\tau \in \mathcal{O}[[P]]_{Adr}^\emptyset$ such that τ is a strong pointer race or $\neg inv(\tau)$. With the same reasoning as in the proof of Theorem 8.15, we conclude that τ is free from strong pointer races. Hence, Theorem 8.6 yields some $\sigma \in [[P]]_\emptyset^\emptyset$ with $inv(\sigma) \implies inv(\tau)$. We get $\neg inv(\sigma)$, a contradiction to the assumption. ■

Now, we are ready to automatically discharge invariant annotations with the help of GC verifiers. In our experiments, we rely on `CAVE` [Vafeiadis 2009, 2010a,b]. Making the link to tools, however, 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 `havoc(p)` to pointers. Both are usually available in verification tools.

The correspondence between the original program P and its translation $inst(P)$ is documented in Theorem 8.22 and as required. Predicate $safe(\cdot)$ evaluates to true iff the assertions hold, i.e., verification is successful. Recall that $[[P]]_\emptyset^\emptyset$ is the GC semantics where addresses are neither freed nor reclaimed. Note that this semantics is the simplest a tool can assume. Our instrumentation also works if the GC tool collects and subsequently reuses garbage addresses.

Theorem 8.22 (Soundness and Completeness). We have $inv([[P]]_\emptyset^\emptyset)$ iff $safe([[inst(P)]]_\emptyset^\emptyset)$. The source-to-source translation is linear in size.

Figure 8.23: Source-to-source translation replacing SMR commands and annotations.

$$\begin{aligned}
 inst(stmt^*) &:= inst(stmt)^* & inst(in:func(\bar{r})) &:= skip \\
 inst(stmt_1 \oplus stmt_2) &:= inst(stmt_1) \oplus inst(stmt_2) & inst(re:func) &:= skip \\
 inst(stmt_1; stmt_2) &:= inst(stmt_1); inst(stmt_2) & inst(com) &:= com \\
 inst(@inv p = q) &:= assert p = q \\
 \\
 inst(in:retire(q)) &:= skip \oplus (retire_ptr := q; retire_flag := true) \\
 inst(@inv active(p)) &:= assert !retire_flag \vee retire_ptr \neq p \\
 \\
 inst(@inv angel r) &:= havoc(r); included_r := false; failed_r := false \\
 inst(@inv q in r) &:= skip \oplus (assume q = r; assert !failed_r; included_r := true) \\
 inst(@inv active(r)) &:= skip \oplus (assume retire_flag \wedge retire_ptr = r; \\
 &\quad assert !included_r; failed_r := true)
 \end{aligned}$$

The source-to-source translation is defined in Figure 8.23. It preserves the structure of the program and does not modify ordinary commands. SMR function invocations and responses will be taken care of by the type system; they are ignored, except for invocations of `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` call 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. An invocation of `retire` then translates into a non-deterministic choice between skipping the command or being the call 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 `retire` invocations is not atomic. Hence, there may be an interleaving where a pointer has been stored in `retire_ptr` but the flag has not yet been raised. The assertion would consider this interleaving safe. However, if there is such an interleaving, there is also one where the assertion fails. Hence, atomicity is not needed.

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

source of difficulty is the fact that angels are second-order variables storing sets. We tackle the problem by guessing an element in the set for which verification fails.

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

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

8.6 Type Inference

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

Theorem 8.24. Given a program P , the type inference $\vdash P$ is computable in time $\mathcal{O}(|P|^2)$.

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

Our type inference algorithm will be a fixed-point computation. The canonical choice for a domain over which to compute would be the set *Types* of all types, ordered by \rightsquigarrow . The problem is that types \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 $\mathbb{E}_L \wedge \mathbb{E}_{L'} \rightsquigarrow \mathbb{E}_L$, yet distinct. This is not merely a theoretical issue of the domain being a quasi order instead of a partial order.² It means we compute over too large a domain, namely a powerset lattice where we should have used a lattice of antichains [Wulf et al. 2006]. We factorize the set of all types along such equivalences $\rightsquigarrow \cap \rightsquigarrow^{-1}$. The resulting *AntiChainTypes* $:= (Types / \rightsquigarrow \cap \rightsquigarrow^{-1}, \rightsquigarrow)$ is a complete lattice [Birkhoff 1948, Theorem 3].

² Quasi orders are reflexive and transitive; partial orders are antisymmetric quasi orders [Birkhoff 1948, p. 4].

Figure 8.25: Constraint system $\Phi(X, P, Y)$, defined inductively over the structure of P .

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

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

$$Env_{\top} := (AntiChainTypes^{Var} \cup \{\top\}, \sqsubseteq).$$

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

The constraint system $\Phi(\Gamma_{init}, P, X)$ is defined in Figure 8.25. We proceed by induction over the structure of the statements in P 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. Furthermore, 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 Figures 8.12 and 8.13 and extract pre_{com} and up_{com} . The former is a predicate on environments capturing the premise of the rule associate with command com . 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 an example, consider Rule (ASSIGN2), repeated from Figure 8.12:

$$\frac{\Gamma(q) = T \quad isValid(T)}{\{\Gamma, p\} \ p := q.next \ \{\Gamma, p : \emptyset\}}$$

For this particular rule, the premise $pre_{com}(\Gamma)$ is $isValid(T)$ with $T = \Gamma(q)$. The update $up_{com}(\Gamma)$ is $\Gamma[p \mapsto \emptyset]$. The strongest enriched environment preserves the information that a type inference has failed, $sp(\top, com) := \top$, for all commands com . For a given environment, we set

$$sp(\Gamma, com) := \begin{cases} up_{com}(\Gamma) & \text{if } pre_{com}(\Gamma) \\ \top & \text{otherwise} \end{cases}.$$

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

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

Theorem 8.26 (Principle Types). For $\Phi(\Gamma_{init}, P, X)$ we have $lsol(X) = \bigsqcap_{\Gamma \vdash \{ \Gamma_{init} \} P \{ \Gamma \}} \Gamma$. Hence, $lsol(X) \neq \top$ if and only if $\vdash P$.

Lastly, we check the complexity of the Kleene iteration. In the lattice of enriched type environments, chains have length at most $|Var| \cdot |\{ \mathbb{A}, \mathbb{L}, \mathbb{S}, \mathbb{E}_{L_1}, \dots, \mathbb{E}_{L_n} \}| + 1$. This is linear in the size of the program as the guarantees only depend on the SMR algorithm, which is not part of the input. With one variable for each program point, also the number of variables in the constraint system is linear in the size of the program. It remains to compute $sp(\bullet, com)$ 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.

8.7 Avoiding Strong Pointer Races

The approach presented so far in this chapter evolves around strong pointer races: our type check establishes strong pointer race freedom so that the actual analysis can be performed under GC. Contrast this to the development from Chapter 7. There, we require ordinary, not strong, pointer race freedom. As a consequence, we have to deal with unsafe assumptions. Unsafe assumptions, in turn, lead to ABAs and the need to prove them harmless, a task which crucially requires to analyze reallocations (of a single address) and cannot be done under GC.

Ruling out code performing unsafe assumptions allows for the more efficient GC analysis. The price to pay is applicability: programmers may exploit intricate invariants of the data structure to ensure that unsafe assumptions do not harm the correctness of the implementation. We have already seen an example of this in Micheal&Scott's queue with hazard pointer (cf. Section 7.3): when protecting nodes, the queue performs assumptions which we deem unsafe although they are correct (harmless ABAs). Therefore, the type check will fail—our approach is not

Figure 8.27: Making a program more atomic to avoid strong pointer races (unsafe assumption) while retaining the original behavior.

(a) Protection scheme from Michael&Scott’s queue, Lines 314 to 316. For the sake of presentation, the original condition from Line 316 is rewritten into the assumptions from Lines 832 and 835. Line 835 exhibits a strong pointer race; more precisely, an unsafe assumption.

```

828 unprotect0();
829 head = Head;
830 protect0(head);
831 if (*) {
832     assume(head != Head);
833     // restart procedure
834 }
835 assume(head == Head);
836 // continue procedure

```

(b) More atomic version of the protection mechanism from Michael&Scott’s queue. The new version avoids the strong pointer race from Line 835, yet it retains all behaviors from the original implementation.

```

837 unprotect0();
838 if (*) {
839     head = Head;
840     protect0(head);
841     assume(head != Head);
842     // restart procedure
843 }
844 atomic { head = Head;
845         protect0(head); }
846 // continue procedure

```

applicable out of the box. In the following, we mitigate the additional restriction introduced by strong pointer race freedom. To that end, we suggest to transform implementations on which the type check fails. The goal of the transformation is to avoid the problematic unsafe assumption while retaining soundness. To be more specific, we transform a given program P into a more atomic program Q . Then, we apply our approach to Q . Because Q is more atomic, it may not observe certain *intermediate* computation steps of P , i.e., $\mathcal{O}\llbracket P \rrbracket_{Adr}^{Adr} \not\subseteq \mathcal{O}\llbracket Q \rrbracket_{Adr}^{Adr}$. Instead, we just require that the correctness of Q entails the correctness of P .

To motivate our choice of program transformations, we elaborate on the popular retry idiom, a programming pattern under safe memory reclamation that relies on unsafe assumptions. Consider Lines 314 to 316 of Michael&Scott’s queue the code of which is repeated in Figure 8.27a (for simplicity, we added the latest `unprotect0` in Line 828). There, (i) the shared `Head` is copied into a local pointer `head`, Line 829, (ii) a protection is issued for that local pointer `head`, Line 830, and then (iii) the procedure is restarted if `Head` has changed since it was read out, Line 832, and continued otherwise, Line 835. The comparison of `Head` and `head` in Line 835 raises a pointer race: `head` may be invalid despite the protection in Line 830. The reason is that Lines 829 and 830 are not executed atomically. The node referenced by `head` could be retired and freed by another thread before the protection takes place. This renders `head` invalid. In fact, Line 835 is an ABA. The ABA, however, is harmless in terms of Definition 7.18: the reallocated address can be accessed safely. (The comparison in Line 832 does not raise a strong pointer race according to Definition 8.4.) The retry idiom for protecting nodes is not limited to hazard pointers but applied in combination with other SMR algorithms as well.

To overcome the problem of harmless strong pointer races, we propose to transform the program into a more atomic version. This idea is referred to as *atomicity abstraction* and well-known in the literature, with contributions ranging from Lipton [1975] to Hawblitzel et al. [2015] to Flanagan and Freund [2020]. We refer the reader to Chapter 9 for an overview. In the above example, we execute the read and the protection atomically, as done in

Figure 8.27b. While this atomicity cannot be achieved in practice, it is useful for verification. In fact, the transformed code has the same behavior. Yet, it is free from strong pointer races (unsafe assumptions). In the following, we develop an atomicity abstraction tailored towards programs with safe memory reclamation. We invoke it whenever the type check fails.

The fundamental technique behind atomicity abstraction are movers [Lipton 1975]. 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 subsequent command of the same thread, effectively constructing an atomic block containing both commands. The difference to existing works [Elmas et al. 2009; Flanagan and Qadeer 2003a; Kragl and Qadeer 2018] is that our programs contain SMR commands the semantics of which depends on the underlying SMR automaton. We propose the following notion of moverness that takes into account an SMR automaton \mathcal{O} . To make it precise, we associate with every command com a unique label lab . Intuitively, the label lab is the line number in which com appears in program P . We use the label to distinguish between syntactically equal commands that appear at different locations in P .

Definition 8.28 (Right Mover). Command com at label lab is a right mover if, for all $\tau.act_1.act_2 \in \mathcal{O} \llbracket P \rrbracket_{Adr}^{Adr}$ with $act_i = \langle t_i, com_i, up_i \rangle$ and $t_1 \neq t_2$ where com_1 is command com at label lab , there is $\tau.act'_2.act'_1 \in \mathcal{O} \llbracket P \rrbracket_{Adr}^{Adr}$ with $act'_i = \langle t_i, com_i, up'_i \rangle$ so that:

$$\begin{aligned} & m_{\tau.act_1.act_2} = m_{\tau.act'_2.act'_1} \\ \text{and} \quad & good(\tau.act'_2.act'_1) \implies good(\tau.act_1) \wedge good(\tau.act_1.act_2) \\ \text{and} \quad & \forall a \in \text{Adr}. \mathcal{F}_{\mathcal{O}}(\tau.act_1.act_2, a) \subseteq \mathcal{F}_{\mathcal{O}}(\tau.act'_2.act'_1, a) . \end{aligned}$$

The equality (first line) is the expected requirement: the memories resulting from the two computations coincide. The implication (second line) asks for the correctness of the new computation to carry over to the original computation and the intermediate step. The inclusion (third line) is SMR specific. It requires that we obtain only more behavior after moving act_1 to the right. It is worth pointing out that swapping the order of act_1 and act_2 might change the updates they perform. This is why we use actions act'_1 and act'_2 after moving act_1 to the right: actions act_i and act'_i coincide up to a potentially different update.

What is remarkable about the definition of moverness is that SMR commands (`in`, `re`, `free`) do not modify the memory (relevant for the equality and the implication) while the remaining memory commands do not affect the history of the computation (relevant for the inclusion). As a consequence, a memory command com always right-moves over an SMR command com' . The reverse is true as well, with one exception: a `free` of address a does not right-move over a `malloc` reusing a . To be precise, both directions require that com does not change the valuation of pointers used by com' . Since only commands of different threads are considered, this is guaranteed to hold if SMR commands com' never access shared variables. This holds for all data structures we are aware of. One reason is that shared variables are subject to interferences, and hence it is not clear what value would be passed to the SMR algorithm. Another reason is that local variables are cheaper to access.

It remains to establish moverness for memory commands and for SMR commands. For memory commands, we apply the moverness proof techniques from the literature [Elmas et al. 2009]. Many of them remain applicable although we have to consider computations from $\mathcal{O}[\![P]\!]_{Adr}^{Adr}$. The reason for this is that moverness techniques heavily rely on rewriting patterns which are proven sound in any context. That is, the rewriting does not rely on the overall program but holds for any memory m_τ . We do not reiterate existing techniques here.

For an SMR command to be a mover, we have to establish the inclusion from the above definition. Consider the interesting case where both act_1 and act_2 execute SMR commands. Since neither of the actions updates the memory, we have $act_i = act_i'$. The inclusion boils down to:

$$\mathcal{F}_O(\tau.act_1.act_2, a) \subseteq \mathcal{F}_O(\tau.act_2.act_1, a) .$$




Let the histories of the involved computations be $h.evt_1.evt_2$ resp. $h.evt_2.evt_1$. Runs of the SMR automaton \mathcal{O} on those histories take the form

$$s_0 \xrightarrow{h} s_1 \xrightarrow{evt_1} s_2 \xrightarrow{evt_2} s_3 \quad \text{and} \quad s_0 \xrightarrow{h} s_1 \xrightarrow{evt_2} s_2' \xrightarrow{evt_1} s_3' .$$

Consequently, it suffices to show that state s_3' allows for more behavior than s_3 , formally $\mathcal{F}_O(s_3, a) \subseteq \mathcal{F}_O(s_3', a)$. The inclusion can be checked with the technique from Proposition 5.3.

8.8 Evaluation

We implemented the approach presented in this chapter in a C++ tool called `SEAL`.³ As stated before, we use the state-of-the-art tool `CAVE` [Vafeiadis 2009, 2010a,b] as a back-end verifier for discharging annotations and checking linearizability. For the type inference, our tool computes the most precise guarantees \mathbb{E}_L on the fly; there is no need for the user to manually specify them. To substantiate the usefulness of our approach, we empirically evaluated `SEAL` on the following data structures: Treiber's stack, Michael&Scott's queue, the DGLM queue, the Vechev&Yahav 2CAS set, the Vechev&Yahav CAS set, the ORVYY set, Michael's set, and Harris' set. Our benchmarks include a version of each data structure for EBR and HP as specified by the SMR automata $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$ and $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^{0,1}$, respectively.

Our findings are listed in Table 8.29. The table includes the time taken (i) for the type inference, (ii) for discharging the invariant annotations, and (iii) to check linearizability. We mark tasks with  if they were successful, with  if they failed, and with  if they timed out after 12h wall time. All experiments were conducted on an Intel i5-8600K@3.6GHz with 16GB of RAM using Ubuntu 16.04 and Clang 6.0.

Our approach is capable of verifying most of the non-blocking data structures we considered. Comparing the total runtime with the approach from Section 8.7, which relies on an exhaustive state space exploration of $\mathcal{O}[\![\bullet]\!]_{Adr}^{one}$, we experience a speed-up of over two orders of magnitude on examples like Michael&Scott's queue. Besides the speed-up, we are the first to automatically verify non-blocking set algorithms that use SMR.

³ `SEAL` is freely available at:  <https://wolff09.github.io/phd/>

Table 8.29: Experimental results for verifying singly-linked data structures using safe memory reclamation. The experiments were conducted on an Intel i5-8600K@3.6GHz with 16GB of RAM using Ubuntu 16.04 and Clang 6.0.

SMR	Program	Type	Inference	Annotations	Linearizability
HP	Treiber's stack	0.7s	✓	12s ✓	1s ✓
	Opt. Treiber's stack	0.5s	✓	11s ✓	1s ✓
	Michael&Scott's queue	0.6s	✓	12s ✓	4s ✓
	DGLM queue	0.6s	✓	1s ✗ ^a	5s ✓
	Vechev&Yahav 2CAS set	1.2s	✓	13s ✓	98s ✓
	Vechev&Yahav CAS set	1.2s	✓	3.5h ✓	42m ✓
	ORVYY set	1.2s	✓	3.2h ✓	47m ✓
	Michael's set	1.2s	✓	90s ✗ ^a	— 🕒
EBR	Treiber's stack	0.6s	✓	10s ✓	1s ✓
	Michael&Scott's queue	0.7s	✓	16s ✓	5s ✓
	DGLM queue	0.7s	✓	1s ✗ ^a	6s ✓
	Vechev&Yahav 2CAS set	0.8s	✓	38s ✓	200s ✓
	Vechev&Yahav CAS set	0.8s	✓	7h ✓	42m ✓
	ORVYY set	0.9s	✓	7h ✓	47m ✓
	Michael's set	0.2s	✓	22s ✗ ^a	— 🕒
	Harris' set	0.5s	✓	1s ✗ ^a	— 🕒

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

We were not able to discharge the annotations of the DGLM queue and Michael's set. Imprecision in the thread-modular abstraction of our back-end verifier, `CAVE`, resulted in false-positives being reported. Hence, we cannot guarantee the soundness of our analysis in these cases. This is no limitation of our approach, it is a shortcoming of the back-end verifier. We reported on a similar issue in Section 7.5 before; there, we needed hints to obtain the necessary precision in some benchmarks.

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

For the benchmarks from Table 8.29 we preprocessed the implementations by applying atomicity abstraction, as discussed in Section 8.7. Our tool is able to apply the preprocessing automatically. We excluded this step to avoid distortions of the reported running times.

Part III

Discussion

Related Work

We discuss works related to the various aspects touched by this thesis. Section 9.1 briefly reviews further data structures that are challenging for verification. Section 9.2 gives an overview of safe memory reclamation. Section 9.3 discusses related analysis and verification techniques.

9.1 Data Structures

Our benchmark set is on par with related works on verification [Abdulla et al. 2013, 2016; Vafeiadis 2010a,b]. Besides the stack and queue implementations we considered, there are more implementations that are worth verifying. For example, elimination-backoff stacks [Bar-Nissan et al. 2011; Hendler et al. 2004], time-stamped stacks and queues [Dodds et al. 2015], bounded array-based queues [Gong and Wing 1990; Shann et al. 2000], and optimized queue implementations [Kogan and Petrank 2011, 2012; Morrison and Afek 2013; Tsigas and Zhang 2001].

Beyond singly-linked structures, there are, for instance, priority queues [Barnes 1992; Israeli and Rappoport 1993; Sundell and Tsigas 2003], skip lists [Fomitchev and Ruppert 2004; Fraser 2004; Sundell and Tsigas 2003], doubly-linked and double-ended queues [Agesen et al. 2000; Arora et al. 1998; Greenwald 1999; Haas 2015; Michael 2003; Shafiei 2015; Sundell and Tsigas 2004], and search trees [Barnes 1992; Braginsky and Petrank 2012; Levandoski et al. 2013; Natarajan and Mittal 2014; Natarajan et al. 2020; Ramachandran and Mittal 2015]. We are not aware of fully automatic techniques that have verified any of those data structures, neither under garbage collection nor under manual memory management.

Depending on the work load, lock-based data structure implementations may outperform non-blocking ones [Hendler et al. 2010]. We did not consider lock-based implementations since memory management is typically easier in the presence of mutual exclusion [Brown 2015; Nikolaev and Ravindran 2020]. It is worth pointing out that our results do not rely on the progress guarantee of the data structure under scrutiny. That is, applying our results to lock-based implementations can justify verification under garbage collection just as well as in the case of non-blocking data structures. Herlihy and Shavit [2008] give an overview of lock-based implementations.

Data structures may obtain further performance improvements when implementing more relaxed correctness criteria than linearizability [Haas et al. 2015; Henzinger et al. 2013a]. We are not aware of automated verification for such data structures. Again, we point out that our results are independent of the correctness criterion that is verified: compositionality and the reduction results guarantee that the simpler semantics reaches the same program locations as the target semantics.

9.2 Memory Reclamation

Besides free lists, EBR, and HP, there is another basic SMR technique: reference counting (RC). RC extends objects with an integer field counting the number of pointers to the object. Safely modifying counters in a non-blocking manner, however, comes with impractical performance penalties [Cohen 2018; Hart et al. 2007], requires hazard pointers [Herlihy et al. 2005], or requires 2CAS [Detlefs et al. 2001] which is unavailable on modern hardware [Brown 2015].

Recent efforts in developing SMR algorithms have mostly combined existing techniques. For example, *DEBRA* [Brown 2015] is an optimized EBR implementation. Harris [2001] modifies EBR to store epochs inside objects. *Hyaline* [Nikolaev and Ravindran 2019] is used like EBR. Herlihy et al. [2002] presented a HP variation. Optimized HP implementations include the work by Aghazadeh et al. [2014] and the work by Dice et al. [2016] as well as *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], *Interval-based Reclamation* [Wen et al. 2018], *Wait-Free Eras* [Nikolaev and Ravindran 2020], and *PEBR* [Kang and Jung 2020] combine EBR and HP. *Free Access* [Cohen 2018] automates the application of Automatic Optimistic Access. While the method promises to be correct by construction, we believe that performance-critical applications choose the SMR technique based on performance rather than ease of use. The demand for automated verification remains. *Beware&Cleanup* [Gidenstam et al. 2005] combines HP and RC. *Isolde* [Yang and Wrigstad 2017] combines EBR and RC. We believe our approach can handle other SMR algorithms besides EBR and HP as well.

9.3 Reasoning and Verification

We give an overview of static analyses related to the techniques presented in this thesis.

9.3.1 Memory Safety

We use our techniques to show that a program is free from (strong) pointer races, meaning that it is memory safe. There are a number of related tools that can check pointer programs for memory safety: a combination of *CCURED* [Necula et al. 2002] and *BLAST* [Henzinger et al. 2003] due to Beyer et al. [2005], *INVADER* [Yang et al. 2008], *XISA* [Laviron et al. 2010], *SLAYER* [Berdine et al. 2011], *INFER* [Calcagno and Distefano 2011], *FORESTER* [Holík et al. 2013], *PREDATOR* [Dudka et al. 2013; Holík et al. 2016], and *APROVE* [Ströder et al. 2017]. These tools can only handle sequential code. Moreover, unlike our type system, they include memory/shape abstractions to identify unsafe pointer operations. We delegate this task to a back-end verifier with the help of

annotations. That is, if the related tools were to support concurrent programs, they were candidates for the back-end. We used `CAVE` [Vafeiadis 2010a,b] as it can also prove linearizability.

Despite the differences, we point out that the combination of `BLAST` and `CCURED` [Beyer et al. 2005] is closest 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 the 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 algorithm. Finally, our type system is more lightweight and we discharge the invariants in a simpler semantics without memory deletions.

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

Besides memory safety, tools like `INVADER`, `SLAYER`, `INFER`, `FORESTER`, `PREDATOR`, and the type system by Kuru and Gordon [2019] discover memory leaks. A successful type check with our type system does not imply the absence of memory leaks. We believe that the outcome of our analysis could help a leak detection tool. For example, by performing a linearizability check to find the abstract data type the data structure under consideration implements. We consider a closer investigation of the matter as future work.

9.3.2 Typestate

Typestate extends an object's static (compile-time) type with a notion of abstract state which reflects the dynamic (run-time) context the object appears in. The methods of an object can be annotated to modify this state and to be available only in a certain state. This can refute syntactically correct programs as semantically incorrect. Analyses checking for methods being called only in the appropriate state include the works by Bierhoff and Aldrich [2007], DeLine and Fähndrich [2004], Fähndrich and DeLine [2002], Fink et al. [2006], and Foster et al. [2002]. Our types can be understood as typestates for pointers (and the objects they reference) geared towards SMR. However, whereas an object's typestate has a global character, our types reflect a thread's local perception. Das et al. [2002] give a typestate analysis based on symbolic execution to increase precision. Similarly, we increase the applicability of our approach by using annotations that are discharged by a back-end verifier. For a more detailed overview on typestate, refer to Ancona et al. [2016].

9.3.3 Program Logics

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

Oftentimes, invariant-based reasoning about interference turns out too restrictive for verification. To overcome this problem, logics like `CARESL` [Turon et al. 2013], `FCSL` [Nanevski et al. 2014], `ICAP` [Svendsen and Birkedal 2014], `TADA` [da Rocha Pinto et al. 2014], `GPS` [Turon et al. 2014], and `IRIS` [Jung et al. 2018, 2015] make use of protocols. A protocol captures possible thread interference, for example, using state transition systems. (Rely-guarantee is a particular instantiation of a protocol [Jung et al. 2015; Turon et al. 2013].) In our approach, SMR automata are protocols that govern memory deletions and protections, that is, describe the influence of SMR-related actions among threads. Our types describe a thread’s local, per-pointer perception of that global protocol.

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

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

9.3.4 Linearizability

Linearizability testing [Burckhardt et al. 2010; Cerný et al. 2010; Emmi and Enea 2018; Emmi et al. 2015; Horn and Kroening 2015; Liu et al. 2009, 2013; Lowe 2017; Travkin et al. 2013; Vechev and Yahav 2008; Yang et al. 2017; Zhang 2011] is a bug hunting technique to find non-linearizable executions in large code bases. Since we focus on verification, we do not go into the details of linearizability testing. However, it could be worthwhile to use a linearizability tester instead of a verification back-end in our type system to provide faster feedback during the development process and only use a verifier once the development is considered finished.

Verification techniques for linearizability fall into two categories: manual techniques (including tool-supported but not fully automated techniques) and automatic techniques. Manual approaches require the human checker to have a deep understanding of the proof technique as well as the program under scrutiny—in our case, this includes a deep understanding of the non-blocking data structure as well as the SMR implementation. This may be the reason why manual proofs rarely consider reclamation [Bäumler et al. 2011; Bouajjani et al. 2017a; Colvin et al. 2005, 2006; Delbianco et al. 2017; Derrick et al. 2011; Doherty and Moir 2009; Elmas et al. 2010; Feldman et al. 2018; Groves 2007, 2008; Hemed et al. 2015; Henzinger et al. 2013b; Jonsson 2012; Khyzha et al. 2017; Liang and Feng 2013; Liang et al. 2012, 2014; O’Hearn et al. 2010; Schellhorn et al. 2012; Sergey et al. 2015a,b]. There are fewer works that consider reclamation [Dodds et al. 2015; Doherty et al. 2004b; Fu et al. 2010; Gotsman et al. 2013; Krishna et al. 2018; Parkinson et al. 2007; Ter-Gabrielyan et al. 2019; Tofan et al. 2011]. Notably, Gotsman et al. [2013] handle memory reclamation by capturing *grace periods*, the time frame during which threads can safely access a given part of memory. Their proof method establishes that memory accesses occur only during a grace period and that deletions occur only after all threads have finished their grace period. However, they do not separate these two tasks. Our approach addresses the former task with the type system checking accessed pointers for guarantees \mathbb{A} , \mathbb{L} , \mathbb{S} and the latter task when verifying that the SMR implementation satisfies its SMR automaton. Furthermore, Gotsman et al. [2013] use temporal logic to reason about grace periods whereas our type system is syntactic. For a more detailed overview of manual techniques, we refer to the survey by Dongol and Derrick [2014].

The landscape of related work for automated linearizability proofs is surprisingly one-sided. Most approaches ignore memory reclamation, that is, assume a garbage collector [Abdulla et al. 2016; Amit et al. 2007; Berdine et al. 2008; Segalov et al. 2009; Sethi et al. 2013; Vafeiadis 2010a,b; Vechev et al. 2009; Zhu et al. 2015]. When reclamation is not considered, memory abstractions are simpler and more efficient, because they can exploit ownership guarantees [Bornat et al. 2005; Boyland 2003] and the resulting thread-local reasoning techniques [O’Hearn et al. 2001; Reynolds 2002]. Very few works [Abdulla et al. 2013; Holík et al. 2017] address the challenge of verifying non-blocking data structures under manual memory management. They assume that FL is used as an SMR algorithm and use hand-crafted semantics that allow for accessing deleted memory. The experimental results from Chapters 6 and 7 build upon the analysis by Abdulla et al. [2013]; at the time of writing, it is the most promising automated analysis that can handle reallocations when memory is managed manually.

9.3.5 Moverness

Movers were 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] and Flanagan and Qadeer [2003a] 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 the case of benign (strong) pointer races. Elmas et al. [2010] extend QED to establish linearizability for simple non-blocking 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 program that contains intermediary steps guiding the transformation [Kragl and Qadeer 2018]. Similarly, ANCHOR [Flanagan and Freund 2020] relies on annotations of mover types that guide the verifier. Movers were also applied in the context of relaxed memory [Bouajjani et al. 2018].

Future Work

We discuss possible directions of future work, some of which we have already hinted on.

Compositionality In Chapter 5 we proposed to verify data structures relative to an SMR automaton rather than an SMR implementation. The automaton abstracts away details of the SMR implementation that are irrelevant for verifying the data structure. In particular, it abstracts away potential starvation or blocking behavior [Tanenbaum and Bos 2014, Section 6.7]. When considering non-blocking code, we expect the data structure to be oblivious to whether or not an SMR function starves/blocks [Herlihy and Shavit 2008, Section 3.7]. Blocking data structures, however, may rely on such properties of the SMR implementation. This can lead to false alarms during verification. In order to rule out the spurious cases, one could modify the SMR program semantics (cf. Figure 5.9) to prevent the execution of `re` commands based on the SMR automaton. Recall that we currently use the SMR automaton only to prevent `free` commands that the SMR algorithm is guaranteed to defer. While the reduction and type check results should require little modification, a generalization of the invariant check is more involved. The reason for this is that the instrumentation from Section 8.5 would require to compile the SMR automaton into code such that the instrumented program mimics the starvation/blocking behavior of the invoked SMR functions. Since SMR automata reason about infinitely many variable valuations simultaneously, devising subclasses of SMR automata that allow for an effective instrumentation might be necessary.

The aforementioned generalization to blocking code is interesting as it allows our approach to verify data structures which use *Read-Copy-Update (RCU)* [McKenney 2004; Tanenbaum and Bos 2014, Section 2.3], a technique that is commonly used in the Linux Kernel [McKenney et al. 2020]. Intuitively, RCU lifts sequential data structures to the concurrent setting. Read accesses are unrestricted and thus allow for non-blocking implementations. Updates, on the other hand, are performed under mutual exclusion. Moreover, memory is reclaimed only after *RCU-barriers* which block until all concurrent readers have finished [Kuru and Gordon 2019]. Ignoring the blocking nature of barriers, our instrumentation from Section 8.5 might refute correct invariant annotations because the following reclamation (retirement) is performed *too soon*.

Along the same lines, it would be interesting to lift the SMR program syntax and semantics to support return values on SMR functions. In practice [Michael et al. 2021], SMR implementations provide functions that return an alias of a given pointer and guarantee that the referenced address is successfully protected. Recall that a successful protection requires to repeatedly read out the given pointer, issue a protection, and ensure that the given pointer still holds the same value (assuming that the given pointer is always active), like Lines 314 to 316 from Micheal&Scott's queue. Currently, without return values, such functions are not supported. To make their functionality available to the data structure nevertheless, the function needs to be copied into the data structure, for example, by replacing all

call-sites with the corresponding implementation. With return values, one could avoid spilling the implementation of the SMR function into the data structure, supporting a more natural separation as reflected in the code. Moreover, keeping small the size of the data structure likely speeds up verification—verifying SMR implementations is already efficient as demonstrated in Section 7.5.3.

It is worth pointing out that the above motivation of alias-generating SMR functions may require the data structure and the SMR implementation to share some parts of the memory, rather than a strict separation as assumed in Chapter 5. To do this, the SMR semantics would need additional environment steps that update the shared parts. The form of those updates could again be encoded by the SMR automaton, e.g., in the form of special events that are emitted by updates to the jointly used memory. As before, adapting the instrumentation from Section 8.5 to integrate the new environment steps might be challenging.

Pointer Races Recall that the free list technique, instead of reclaiming memory, makes previously retired memory immediately accessible for reuse. For verification purposes, Section 5.2 suggested to model this by immediately freeing retired addresses and allowing freed memory to be accessed (dereferenced). Doing so, however, jeopardizes the applicability of (strong) pointer race freedom. Freeing an address renders all pointers to that address invalid so that subsequent accesses raise a pointer race. To overcome this problem, one could devise a specialized version of the reduction results from Chapters 7 and 8 that allows for read accesses of freed memory. The value resulting from reading from an invalid pointer must be treated with care. The correspondences we have laid out do not guarantee that the obtained value coincides when mimicking the access in a smaller semantics (that elides memory reuse). Similarly to the harmful ABA freedom check, one needs to ensure that the obtained value does not influence the computation in a way that the simpler semantics cannot reproduce. Haziza et al. [2016] present a possible solution.¹ It is worth pointing out that the an integration of return values as suggested above settles the issue as well.

With the technique from Section 8.2 for avoiding frees, one could strengthen the reduction result from Chapter 7 and avoid frees of all addresses that are not available for reallocation. That is, an analysis of $\mathcal{O}[\![P]\!]_{one}^{one} = \bigcup_a \mathcal{O}[\![P]\!]_{\{a\}}^{\{a\}}$ rather than $\mathcal{O}[\![P]\!]_{Adr}^{one} = \bigcup_a \mathcal{O}[\![P]\!]_{Adr}^{\{a\}}$ would suffice.

Types The type system from Chapter 8 comes with the restriction that the underlying SMR automaton must not contain more than two variables. Lifting the restriction requires an adaptation of the soundness result. More specifically, it requires to adapt how the semantic information from computations τ is tied to the syntactic information of typings $x : T$ for threads t , denoted by $\tau, t \models x : T$ in Section 8.3.4. Currently, t and $m_\tau(x)$ uniquely define the valuation of the SMR automaton variables. With more than two variables, however, there no longer is a unique valuation. When considering all possible valuations that evaluate *some* variable to t and $m_\tau(x)$, types likely become too imprecise as they can no longer track a specific thread/address through the SMR automaton. When using distinguished variables z_t and z_a in the SMR automaton that are populated with t and $m_\tau(x)$, respectively, then types encode only a fragment of the SMR automata’s behavior. Since SMR automata are negative specifications, this means that the resulting abstraction via types becomes coarser. It has to be checked whether the introduced coarseness allows for successful verification.

¹ The author is a coauthor of [Haziza et al. 2016] some results of which are part of Chapter 6.

The evaluation from Section 8.8 demonstrated that checking invariants for correctness is oftentimes more time consuming than checking the actual correctness property. More specifically, our experiments suggest that the instrumentation from Section 8.5 introduces severe overheads for angels. Alternate instrumentations and more powerful constructs in the underlying GC verifier could address the performance bottleneck.

Beyond Data Structures This thesis has focused on high-performance data structures with manual memory management. We exploited their interaction to tame the mutual influence, leading to separate verification tasks. It would be interesting to apply the same methods to more systems across all sizes. Prime candidates are systems that exhibit similar interaction patterns. Examples are cloud computing, data bases, and hardware architectures. We elaborate.

The goal of cloud computing is to deliver to customers off-site computing resources as a service [Armbrust et al. 2009; Foster et al. 2008]. To provide such services with low latency and high availability, cloud computing infrastructures consist of a multitude of physically dispersed data centers. To maintain high availability, updates to the infrastructure need to be applied during operation. In the literature, this is referred to as dynamic reconfiguration of distributed systems [Barbacci et al. 1990; Kramer and Magee 1990]. Introducing new configurations is reminiscent of memory reclamation as we have studied it [Bidan et al. 1998; Gilbert et al. 2010]: out-dated configurations can be removed only if no part of the infrastructure is actively using it. Moreover, verifying the components of cloud infrastructures compositionally is likely to benefit verification [Sergey et al. 2018].

Data bases are responsible for serving large amounts of data, for example, within a single data center from the above cloud infrastructure [Silberschatz et al. 2020]. As such, they are similar to data structures which serve data within a single machine. Serving many rather than one machine, however, increases latencies undesirably [Brewer 2000; DeCandia et al. 2007; Gilbert and Lynch 2002]. To fight this problem, data bases trade consistency for latency. This results in intricate behaviors and makes verification challenging [Bouajjani et al. 2017b; Gotsman et al. 2016; Wilcox et al. 2015]. Semantic reductions can help to tame the intricate behaviors, i.e., avoid inconsistencies like we avoided memory reuse for data structures. There has already been some work into that direction [von Gleissenthall et al. 2019].

Hardware architectures integrate multiple levels of caching/buffering to speed up computations [Hennessy and Patterson 2012; Sewell et al. 2010; Stenström 1990]. Similar to data bases, semantic reductions could help to avoid the resulting inconsistencies and reason about the hardware as if it had less or no caches/buffers. A famous result along those lines is the data race freedom guarantee [Adve and Hill 1993]: if there are no unsynchronized concurrent reads and writes to a single location under sequential consistency [Lamport 1979], then one can ignore the caches/buffers of the actual hardware architecture and reason under sequential consistency. When consistency is traded for latencies, data races might occur and one needs intermediate results in the spirit of harmful ABA freedom. Reducing the number of addresses that are buffered/cached could ease verification, similar in spirit to existing robustness results [Bouajjani et al. 2015a, 2013, 2011; Calin et al. 2013; Owens 2010].

Conclusion

Throughout this thesis we have presented techniques that substantially simplify the verification of non-blocking data structures that manage manually their memory with the help of an SMR algorithm. Our results are based on a compositionality introduced in Chapter 5. It captures the influence the SMR algorithm has on the data structure in the form of an SMR automaton. This automaton abstracts from details of the implementation, encoding only the reclamation behavior of the SMR algorithm. Compositionality alone, however, was not enough to handle the intricacies of non-blocking data structures paired with memory reclamation in that automatic analyses remained imprecise and inefficient. In Chapter 6 we observed that the imprecision is introduced by the thread-modular abstraction which is necessary to verify concurrency libraries. To fight this imprecision, we introduced weak ownership. It is inspired by traditional ownership, i.e., access exclusivity, and tailored towards reclamation. Here, the key observation was that ownership may be broken by invalid (dangling) pointers only. While the new ownership technique yielded sufficient precision, the efficiency gains were too insignificant to handle memory management via SMR. Chapter 7 tackled the efficiency concerns with a semantic reduction. We showed that verification can be conducted in a much simpler semantics, namely one where only a single address can be reallocated. The reduction result came with two requirements: pointer race freedom and harmful ABA freedom. The former requires the absence of unsafe operations, like dereferencing deleted memory. The latter requires the absence of ABAs that could not be mimicked in the simpler semantics. Crucially, both properties can be established in the smaller semantics. This resulted in a tool capable of verifying non-blocking stacks and queues which use SMR. While successful, the approach required hand-crafted verification engines that support memory reclamation and integrate the ABA check. Finally, Chapter 8 strengthened the semantic reduction, showing that verification under garbage collection can answer the verification question for manual memory management. The reduction required strong pointer race freedom. Interestingly, the check for strong pointer races was automated with a type system. There, types attach to pointers the possible reclamation behavior that they are subject to. This behavior was elegantly encoded as a set of locations in the SMR automaton that specifies the SMR algorithm in use. The resulting tool, `SEAL`, proved to be highly efficient. To the best of our knowledge, `SEAL` is the first tool to fully automatically prove correct (linearizable) non-blocking data structures with state-of-the-art SMR algorithms. Altogether, our reductions eradicated the need for verification under a semantics other than garbage collection.

Bibliography

- [**Abdulla et al. 2013**] 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. [✂ DOI:10.1007/978-3-642-36742-7_23](https://doi.org/10.1007/978-3-642-36742-7_23) (cit. on pp. 11, 12, 15, 42, 47, 57–59, 71, 102, 106).
- [**Abdulla et al. 2017**] Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. 2017. *An Integrated Specification and Verification Technique for Highly Concurrent Data Structures*. In: STTT 19 (5). [✂ DOI:10.1007/s10009-016-0415-4](https://doi.org/10.1007/s10009-016-0415-4) (cit. on pp. 42, 47, 57, 58, 71).
- [**Abdulla et al. 2016**] Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. 2016. *Automated Verification of Linearization Policies*. In: SAS, LNCS vol. 9837. Springer. [✂ DOI:10.1007/978-3-662-53413-7_4](https://doi.org/10.1007/978-3-662-53413-7_4) (cit. on pp. 9, 102, 106).
- [**Adve and Hill 1993**] Sarita V. Adve and Mark D. Hill. 1993. *A Unified Formalization of Four Shared-Memory Models*. In: IEEE Trans. Parallel Distributed Syst. 4 (6). [✂ DOI:10.1109/71.242161](https://doi.org/10.1109/71.242161) (cit. on p. 110).
- [**Agesen et al. 2000**] Ole Agesen, David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Alan Martin, Nir Shavit, and Guy L. Steele Jr.. 2000. *DCAS-based concurrent dequeues*. In: SPAA, ACM. [✂ DOI:10.1145/341800.341817](https://doi.org/10.1145/341800.341817) (cit. on pp. 15, 102).
- [**Aghazadeh et al. 2014**] Zahra Aghazadeh, Wojciech M. Golab, and Philipp Woelfel. 2014. *Making objects writable*. In: PODC, ACM. [✂ DOI:10.1145/2611462.2611483](https://doi.org/10.1145/2611462.2611483) (cit. on p. 103).
- [**Alistarh et al. 2014**] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. 2014. *StackTrack: an automated transactional approach to concurrent memory reclamation*. In: EuroSys, ACM. [✂ DOI:10.1145/2592798.2592808](https://doi.org/10.1145/2592798.2592808) (cit. on p. 103).
- [**Alistarh et al. 2015**] Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. 2015. *ThreadScan: Automatic and Scalable Memory Reclamation*. In: SPAA, ACM. [✂ DOI:10.1145/2755573.2755600](https://doi.org/10.1145/2755573.2755600) (cit. on p. 103).
- [**Amit et al. 2007**] 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. [✂ DOI:10.1007/978-3-540-73368-3_49](https://doi.org/10.1007/978-3-540-73368-3_49) (cit. on p. 106).
- [**Ancona et al. 2016**] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniérou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. *Behavioral Types in Programming Languages*. In: Foundations and Trends in Programming Languages 3 (2-3). [✂ DOI:10.1561/25000000031](https://doi.org/10.1561/25000000031) (cit. on p. 104).
- [**Anderson and Moir 1995**] James H. Anderson and Mark Moir. 1995. *Universal Constructions for Multi-Object Operations*. In: PODC, ACM. [✂ DOI:10.1145/224964.224985](https://doi.org/10.1145/224964.224985) (cit. on p. 17).
- [**Arm Limited 2020**] Arm Limited. 2020. *Arm Architecture Reference Manual: Armv8, for Armv8-A architecture profile*. Version F.c. [✂ https://developer.arm.com/documentation/ddi0487/fc/](https://developer.arm.com/documentation/ddi0487/fc/) (cit. on pp. 16, 17).

- [Armbrust et al. 2009] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2009. *Above the Clouds: A Berkeley View of Cloud Computing*. Tech. rep. UCB/EECS-2009-28. EECS Department, University of California, Berkeley. [✂ https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html](https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html) (cit. on p. 110).
- [Arora et al. 1998] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. *Thread Scheduling for Multiprogrammed Multiprocessors*. In: *SPAA*, ACM. [✂ DOI:10.1145/277651.277678](https://doi.org/10.1145/277651.277678) (cit. on p. 102).
- [Back 1989] Ralph-Johan Back. 1989. *A Method for Refining Atomicity in Parallel Algorithms*. In: *PARLE*, LNCS vol. 366. Springer. [✂ DOI:10.1007/3-540-51285-3_42](https://doi.org/10.1007/3-540-51285-3_42) (cit. on p. 106).
- [Baier and Katoen 2008] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT Press. [✂ ISBN:978-0-262-02649-9](https://doi.org/10.1007/978-0-262-02649-9) (cit. on p. 45).
- [Balmau et al. 2016] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. 2016. *Fast and Robust Memory Reclamation for Concurrent Data Structures*. In: *SPAA*, ACM. [✂ DOI:10.1145/2935764.2935790](https://doi.org/10.1145/2935764.2935790) (cit. on pp. 23, 103).
- [Bar-Nissan et al. 2011] Gal Bar-Nissan, Danny Hendler, and Adi Suissa. 2011. *A Dynamic Elimination-Combining Stack Algorithm*. In: *OPODIS*, LNCS vol. 7109. Springer. [✂ DOI:10.1007/978-3-642-25873-2_37](https://doi.org/10.1007/978-3-642-25873-2_37) (cit. on p. 102).
- [Barbacci et al. 1990] Mario Barbacci, Dennis L. Doubleday, and Charles B. Weinstock. 1990. *Application-Level Programming*. In: *ICDCS*, IEEE Computer Society. [✂ DOI:10.1109/ICDCS.1990.89315](https://doi.org/10.1109/ICDCS.1990.89315) (cit. on p. 110).
- [Barnes 1992] Greg Barnes. 1992. *Wait-Free Algorithms for Heaps*. Tech. rep. [✂ https://dada.cs.washington.edu/research/tr/1994/12/UW-CSE-94-12-07.pdf](https://dada.cs.washington.edu/research/tr/1994/12/UW-CSE-94-12-07.pdf) (cit. on p. 102).
- [Barnes 1993] Greg Barnes. 1993. *A Method for Implementing Lock-Free Shared-Data Structures*. In: *SPAA*, ACM. [✂ DOI:10.1145/165231.165265](https://doi.org/10.1145/165231.165265) (cit. on p. 15).
- [Barr 2013] Michael Barr. 2013. *An Update on Toyota and Unintended Acceleration*. [✂ https://embeddedgurus.com/barr-code/2013/10/an-update-on-toyota-and-unintended-acceleration/](https://embeddedgurus.com/barr-code/2013/10/an-update-on-toyota-and-unintended-acceleration/) (cit. on p. 8).
- [Bäumler et al. 2011] Simon Bäumler, Gerhard Schellhorn, Bogdan Tofan, and Wolfgang Reif. 2011. *Proving linearizability with temporal logic*. In: *Formal Asp. Comput.* 23 (1). [✂ DOI:10.1007/s00165-009-0130-y](https://doi.org/10.1007/s00165-009-0130-y) (cit. on p. 106).
- [Bayer and Schkolnick 1977] Rudolf Bayer and Mario Schkolnick. 1977. *Concurrency of Operations on B-Trees*. In: *Acta Informatica* 9. [✂ DOI:10.1007/BF00263762](https://doi.org/10.1007/BF00263762) (cit. on p. 31).
- [Berdine et al. 2011] Josh Berdine, Byron Cook, and Samin Ishtiaq. 2011. *SLayer: Memory Safety for Systems-Level Code*. In: *CAV*, LNCS vol. 6806. Springer. [✂ DOI:10.1007/978-3-642-22110-1_15](https://doi.org/10.1007/978-3-642-22110-1_15) (cit. on p. 103).
- [Berdine et al. 2008] 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. [✂ DOI:10.1007/978-3-540-70545-1_37](https://doi.org/10.1007/978-3-540-70545-1_37) (cit. on pp. 11, 42, 51, 106).
- [Beyer et al. 2005] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2005. *Checking Memory Safety with Blast*. In: *FASE*, LNCS vol. 3442. Springer. [✂ DOI:10.1007/978-3-540-31984-9_2](https://doi.org/10.1007/978-3-540-31984-9_2) (cit. on pp. 103, 104).
- [Bidan et al. 1998] Christophe Bidan, Valérie Issarny, Titos Saridakis, and Apostolos V. Zarras. 1998. *A dynamic reconfiguration service for CORBA*. In: *CDS*, IEEE Computer Society. [✂ DOI:10.1109/CDS.1998.675756](https://doi.org/10.1109/CDS.1998.675756) (cit. on p. 110).
- [Bierhoff and Aldrich 2007] Kevin Bierhoff and Jonathan Aldrich. 2007. *Modular Typestate Checking of Aliased Objects*. In: *OOPSLA*, ACM. [✂ DOI:10.1145/1297027.1297050](https://doi.org/10.1145/1297027.1297050) (cit. on p. 104).

- [Birkhoff 1948] Garrett Birkhoff. 1948. *Lattice Theory (revised edition)*. American Mathematical Society.  ISBN: 9780821889534 (cit. on p. 94).
- [Bizjak et al. 2019] Ales Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. *Iron: Managing Obligations in Higher-order Concurrent Separation Logic*. In: *PACMPL* 3 (POPL).  DOI:10.1145/3290378 (cit. on p. 105).
- [Blechmann 2011] Tim Blechmann. 2011. *Boost C++ Libraries Documentation: lockfree*.  https://www.boost.org/doc/libs/1_74_0/doc/html/lockfree/rationale.html (cit. on p. 28).
- [Bornat et al. 2005] Richard Bornat, Cristiano Calcagno, Peter W. O’Hearn, and Matthew J. Parkinson. 2005. *Permission Accounting in Separation Logic*. In: *POPL*, ACM.  DOI:10.1145/1040305.1040327 (cit. on p. 106).
- [Bouajjani et al. 2015a] Ahmed Bouajjani, Georgel Calin, Egor Derevenetc, and Roland Meyer. 2015. *Lazy TSO Reachability*. In: *FASE*, LNCS vol. 9033. Springer.  DOI:10.1007/978-3-662-46675-9_18 (cit. on p. 110).
- [Bouajjani et al. 2013] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. *Checking and Enforcing Robustness against TSO*. In: *ESOP*, LNCS vol. 7792. Springer.  DOI:10.1007/978-3-642-37036-6_29 (cit. on p. 110).
- [Bouajjani et al. 2015b] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015. *On Reducing Linearizability to State Reachability*. In: *ICALP (2)*, LNCS vol. 9135. Springer.  DOI:10.1007/978-3-662-47666-6_8 (cit. on p. 49).
- [Bouajjani et al. 2017a] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. 2017. *Proving Linearizability Using Forward Simulations*. In: *CAV (2)*, LNCS vol. 10427. Springer.  DOI:10.1007/978-3-319-63390-9_28 (cit. on p. 106).
- [Bouajjani et al. 2017b] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. *On verifying causal consistency*. In: *POPL*, ACM.  DOI:10.1145/3093333.3009888 (cit. on p. 110).
- [Bouajjani et al. 2018] 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.  DOI:10.1007/978-3-319-96142-2_21 (cit. on p. 107).
- [Bouajjani et al. 2011] Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. 2011. *Deciding Robustness against Total Store Ordering*. In: *ICALP (2)*, LNCS vol. 6756. Springer.  DOI:10.1007/978-3-642-22012-8_34 (cit. on p. 110).
- [Boyland 2003] John Boyland. 2003. *Checking Interference with Fractional Permissions*. In: *SAS*, LNCS vol. 2694. Springer.  DOI:10.1007/3-540-44898-5_4 (cit. on p. 106).
- [Braginsky et al. 2013] Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2013. *Drop the anchor: lightweight memory management for non-blocking data structures*. In: *SPAA*, ACM.  DOI:10.1145/2486159.2486184 (cit. on p. 103).
- [Braginsky and Petrank 2012] Anastasia Braginsky and Erez Petrank. 2012. *A lock-free B+tree*. In: *SPAA*, ACM.  DOI:10.1145/2312005.2312016 (cit. on p. 102).
- [Brewer 2000] Eric A. Brewer. 2000. *Towards robust distributed systems (abstract)*. In: *PODC*, ACM.  DOI:10.1145/343477.343502 (cit. on p. 110).
- [Brookes 2004] Stephen D. Brookes. 2004. *A Semantics for Concurrent Separation Logic*. In: *CONCUR*, LNCS vol. 3170. Springer.  DOI:10.1007/978-3-540-28644-8_2 (cit. on p. 105).
- [Brown 2015] Trevor Alexander Brown. 2015. *Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way*. In: *PODC*, ACM.  DOI:10.1145/2767386.2767436 (cit. on pp. 9, 17, 18, 23, 24, 75, 102, 103).

- [Burckhardt et al. 2010] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. *Line-up: a complete and automatic linearizability checker*. In: *PLDI*, ACM. ✎ DOI:10.1145/1806596.1806634 (cit. on p. 105).
- [Calcagno and Distefano 2011] 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. ✎ DOI:10.1007/978-3-642-20398-5_33 (cit. on p. 103).
- [Calin et al. 2013] Georgel Calin, Egor Derevenetc, Rupak Majumdar, and Roland Meyer. 2013. *A Theory of Partitioned Global Address Spaces*. In: *FSTTCS*, LIPIcs vol. 24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. ✎ DOI:10.4230/LIPIcs.FSTTCS.2013.127 (cit. on p. 110).
- [Cao et al. 2017] Qinxiang Cao, Santiago Cuellar, and Andrew W. Appel. 2017. *Bringing Order to the Separation Logic Jungle*. In: *APLAS*, LNCS vol. 10695. Springer. ✎ DOI:10.1007/978-3-319-71237-6_10 (cit. on p. 9).
- [Carson 2019] Biz Carson. 2019. *Lime Scooter Software Glitch Causes Random Braking, Dozens Of Rider Injuries*. ✎ <https://www.forbes.com/sites/bizcarson/2019/02/22/lime-scooter-software-glitch-causes-random-braking-dozens-of-rider-injuries/> (cit. on p. 8).
- [Castegren and Wrigstad 2017] 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. ✎ DOI:10.4230/LIPIcs.ECOOP.2017.6 (cit. on pp. 11, 52, 104).
- [CBS News 2010] CBS News. 2010. *Toyota "Unintended Acceleration" Has Killed 89*. ✎ <https://www.cbsnews.com/news/toyota-unintended-acceleration-has-killed-89/> (cit. on p. 8).
- [Cerný et al. 2010] Pavol Cerný, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. 2010. *Model Checking of Linearizability of Concurrent List Implementations*. In: *CAV*, LNCS vol. 6174. Springer. ✎ DOI:10.1007/978-3-642-14295-6_41 (cit. on p. 105).
- [Chang et al. 2020] Bor-Yuh Evan Chang, Cezara Dragoi, Roman Manevich, Noam Rinetzkzy, and Xavier Rival. 2020. *Shape Analysis*. In: *Found. Trends Program. Lang.* 6 (1-2). ✎ DOI:10.1561/25000000037 (cit. on p. 54).
- [Charette 2005] R. N. Charette. 2005. *Why software fails [software failure]*. In: *IEEE Spectrum* 42 (9). ✎ DOI:10.1109/MSPEC.2005.1502528 (cit. on p. 8).
- [Clarke 2008] Edmund M. Clarke. 2008. *The Birth of Model Checking*. In: *25 Years of Model Checking*, LNCS vol. 5000. Springer. ✎ DOI:10.1007/978-3-540-69850-0_1 (cit. on p. 8).
- [Clarke and Emerson 1981] Edmund M. Clarke and E. Allen Emerson. 1981. *Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic*. In: *Logic of Programs*, LNCS vol. 131. Springer. ✎ DOI:10.1007/BFb0025774 (cit. on p. 9).
- [Cleaveland and Steffen 1991] Rance Cleaveland and Bernhard Steffen. 1991. *Computing Behavioural Relations, Logically*. In: *ICALP*, LNCS vol. 510. Springer. ✎ DOI:10.1007/3-540-54233-7_129 (cit. on p. 45).
- [Cohen 2018] Nachshon Cohen. 2018. *Every data structure deserves lock-free memory reclamation*. In: *PACMPL* 2 (OOPSLA). ✎ DOI:10.1145/3276513 (cit. on pp. 9, 19, 103).
- [Cohen and Petrank 2015a] Nachshon Cohen and Erez Petrank. 2015. *Automatic memory reclamation for lock-free data structures*. In: *OOPSLA*, ACM. ✎ DOI:10.1145/2814270.2814298 (cit. on pp. 15, 18, 103).
- [Cohen and Petrank 2015b] Nachshon Cohen and Erez Petrank. 2015. *Efficient Memory Management for Lock-Free Data Structures with Optimistic Access*. In: *SPAA*, ACM. ✎ DOI:10.1145/2755573.2755579 (cit. on p. 103).

- [Colvin et al. 2005] Robert Colvin, Simon Doherty, and Lindsay Groves. 2005. *Verifying Concurrent Data Structures by Simulation*. In: *Electr. Notes Theor. Comput. Sci.* 137 (2). [✂ DOI:10.1016/j.entcs.2005.04.026](#) (cit. on p. 106).
- [Colvin et al. 2006] 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. [✂ DOI:10.1007/11817963_44](#) (cit. on p. 106).
- [Coppo and Dezani-Ciancaglini 1978] Mario Coppo and Mariangiola Dezani-Ciancaglini. 1978. *A New Type Assignment for λ -Terms*. In: *Arch. Math. Log.* 19 (1). [✂ DOI:10.1007/BF02011875](#) (cit. on p. 80).
- [Crary et al. 1999] Karl Crary, David Walker, and J. Gregory Morrisett. 1999. *Typed Memory Management in a Calculus of Capabilities*. In: *POPL, ACM*. [✂ DOI:10.1145/292540.292564](#) (cit. on p. 81).
- [da Rocha Pinto et al. 2014] 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. [✂ DOI:10.1007/978-3-662-44202-9_9](#) (cit. on p. 105).
- [Das et al. 2002] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. *ESP: Path-Sensitive Program Verification in Polynomial Time*. In: *PLDI, ACM*. [✂ DOI:10.1145/512529.512538](#) (cit. on p. 104).
- [DeCandia et al. 2007] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. *Dynamo: amazon's highly available key-value store*. In: *SOSP, ACM*. [✂ DOI:10.1145/1294261.1294281](#) (cit. on p. 110).
- [deGrasse Tyson 2018] Neil deGrasse Tyson. 2018. *The Future of Colonizing Space*. Timestamp: 39min15s. YouTube. [✂ http s://youtu.be/X_m1mPtYzTk?t=2355](#) (cit. on p. 2).
- [Delbianco et al. 2017] 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. [✂ DOI:10.4230/LIPIcs.ECOOP.2017.8](#) (cit. on p. 106).
- [DeLine and Fähndrich 2004] Robert DeLine and Manuel Fähndrich. 2004. *Typestates for Objects*. In: *ECOOP, LNCS* vol. 3086. Springer. [✂ DOI:10.1007/978-3-540-24851-4_21](#) (cit. on p. 104).
- [de Roever et al. 2001] Willem P. de Roever, Frank S. de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. 2001. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge Tracts in Theoretical Computer Science vol. 54. Cambridge University Press. [✂ ISBN:0-521-80608-9](#) (cit. on p. 10).
- [Derrick et al. 2011] John Derrick, Gerhard Schellhorn, and Heike Wehrheim. 2011. *Mechanically verified proof obligations for linearizability*. In: *ACM Trans. Program. Lang. Syst.* 33 (1). [✂ DOI:10.1145/1889997.1890001](#) (cit. on p. 106).
- [Detlefs et al. 2001] David Detlefs, Paul Alan Martin, Mark Moir, and Guy L. Steele Jr.. 2001. *Lock-free reference counting*. In: *PODC, ACM*. [✂ DOI:10.1145/383962.384016](#) (cit. on p. 103).
- [Dice et al. 2016] Dave Dice, Maurice Herlihy, and Alex Kogan. 2016. *Fast non-intrusive memory reclamation for highly-concurrent data structures*. In: *ISMM, ACM*. [✂ DOI:10.1145/2926697.2926699](#) (cit. on p. 103).
- [Dietl and Müller 2013] Werner Dietl and Peter Müller. 2013. *Object Ownership in Program Verification*. In: *Aliasing in Object-Oriented Programming, LNCS* vol. 7850. Springer. [✂ DOI:10.1007/978-3-642-36946-9_11](#) (cit. on pp. 11, 52).
- [Dijkstra 1982] Edsger W. Dijkstra. 1982. *On Making Solutions More and More Fine-Grained*. In: *Selected Writings on Computing: A personal Perspective*, New York, NY: Springer New York. [✂ DOI:10.1007/978-1-4612-5695-3_53](#) (cit. on p. 105).

- [Dinsdale-Young et al. 2010] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. *Concurrent Abstract Predicates*. In: *ECOOP*, LNCS vol. 6183. Springer. [✎ DOI:10.1007/978-3-642-14107-2_24](#) (cit. on p. 105).
- [Dodds et al. 2009] Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. *Deny-Guarantee Reasoning*. In: *ESOP*, LNCS vol. 5502. Springer. [✎ DOI:10.1007/978-3-642-00590-9_26](#) (cit. on p. 105).
- [Dodds et al. 2015] Mike Dodds, Andreas Haas, and Christoph M. Kirsch. 2015. *A Scalable, Correct Time-Stamped Stack*. In: *POPL*, ACM. [✎ DOI:10.1145/2676726.2676963](#) (cit. on pp. 102, 106).
- [Doeppner 1977] Thomas W. Doeppner Jr.. 1977. *Parallel Program Correctness Through Refinement*. In: *POPL*, ACM. [✎ DOI:10.1145/512950.512965](#) (cit. on p. 106).
- [Doherty et al. 2004a] Simon Doherty, David Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul Alan Martin, Mark Moir, Nir Shavit, and Guy L. Steele Jr.. 2004. *DCAS is not a silver bullet for nonblocking algorithm design*. In: *SPAA*, ACM. [✎ DOI:10.1145/1007912.1007945](#) (cit. on p. 8).
- [Doherty et al. 2004b] 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. [✎ DOI:10.1007/978-3-540-30232-2_7](#) (cit. on pp. 8, 29, 30, 106).
- [Doherty and Moir 2009] Simon Doherty and Mark Moir. 2009. *Nonblocking Algorithms and Backward Simulation*. In: *DISC*, LNCS vol. 5805. Springer. [✎ DOI:10.1007/978-3-642-04355-0_28](#) (cit. on p. 106).
- [Dongol and Derrick 2014] Brijesh Dongol and John Derrick. 2014. *Verifying linearizability: A comparative survey*. In: *CoRR* abs/1410.6268. [✎ https://arxiv.org/abs/1410.6268](https://arxiv.org/abs/1410.6268) (cit. on p. 106).
- [Dragojevic et al. 2011] Aleksandar Dragojevic, Maurice Herlihy, Yossi Lev, and Mark Moir. 2011. *On the power of hardware transactional memory to simplify memory management*. In: *PODC*, ACM. [✎ DOI:10.1145/1993806.1993821](#) (cit. on p. 103).
- [Dudka et al. 2013] Kamil Dudka, Petr Peringer, and Tomás Vojnar. 2013. *Byte-Precise Verification of Low-Level List Manipulation*. In: *SAS*, LNCS vol. 7935. Springer. [✎ DOI:10.1007/978-3-642-38856-9_13](#) (cit. on p. 103).
- [Elmas et al. 2010] 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. [✎ DOI:10.1007/978-3-642-12002-2_25](#) (cit. on pp. 106, 107).
- [Elmas et al. 2009] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. *A Calculus of Atomic Actions*. In: *POPL*, ACM. [✎ DOI:10.1145/1480881.1480885](#) (cit. on pp. 98, 99, 106).
- [Emmi and Enea 2018] Michael Emmi and Constantin Enea. 2018. *Sound, complete, and tractable linearizability monitoring for concurrent collections*. In: *PACMPL 2 (POPL)*. [✎ DOI:10.1145/3158113](#) (cit. on p. 105).
- [Emmi et al. 2015] Michael Emmi, Constantin Enea, and Jad Hamza. 2015. *Monitoring refinement via symbolic reasoning*. In: *PLDI*, ACM. [✎ DOI:10.1145/2737924.2737983](#) (cit. on p. 105).
- [Fähndrich and DeLine 2002] Manuel Fähndrich and Robert DeLine. 2002. *Adoption and Focus: Practical Linear Types for Imperative Programming*. In: *PLDI*, ACM. [✎ DOI:10.1145/512529.512532](#) (cit. on p. 104).
- [Feldman et al. 2018] Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. 2018. *Order out of Chaos: Proving Linearizability Using Local Views*. In: *DISC, LIPIcs* vol. 121. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. [✎ DOI:10.4230/LIPIcs.DISC.2018.23](#) (cit. on p. 106).

- [Feng 2009] Xinyu Feng. 2009. *Local Rely-guarantee Reasoning*. In: *POPL*, ACM. [DOI:10.1145/1480881.1480922](#) (cit. on p. 105).
- [Feng et al. 2007] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. *On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning*. In: *ESOP*, LNCS vol. 4421. Springer. [DOI:10.1007/978-3-540-71316-6_13](#) (cit. on p. 105).
- [Fink et al. 2006] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. *Effective Typestate Verification in the Presence of Aliasing*. In: *ISSTA*, ACM. [DOI:10.1145/1146238.1146254](#) (cit. on p. 104).
- [Flanagan and Freund 2020] Cormac Flanagan and Stephen N. Freund. 2020. *The Anchor Verifier for Blocking and Non-Blocking Concurrent Software*. In: *PACMPL 4 (OOPSLA)*. [DOI:10.1145/3428224](#) (cit. on pp. 97, 107).
- [Flanagan et al. 2008] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. 2008. *Types for Atomicity: Static Checking and Inference for Java*. In: *ToPLaS 30 (4)*. [DOI:10.1145/1377492.1377495](#) (cit. on p. 106).
- [Flanagan et al. 2005] Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia. 2005. *Modular verification of multithreaded programs*. In: *Theor. Comput. Sci.* 338 (1-3). [DOI:10.1016/j.tcs.2004.12.006](#) (cit. on p. 106).
- [Flanagan and Leino 2001] Cormac Flanagan and K. Rustan M. Leino. 2001. *Houdini, an Annotation Assistant for ESC/Java*. In: *FME*, LNCS vol. 2021. Springer. [DOI:10.1007/3-540-45251-6_29](#) (cit. on pp. 13, 76).
- [Flanagan and Qadeer 2003a] Cormac Flanagan and Shaz Qadeer. 2003. *A Type and Effect System for Atomicity*. In: *PLDI*, ACM. [DOI:10.1145/781131.781169](#) (cit. on pp. 98, 106).
- [Flanagan and Qadeer 2003b] Cormac Flanagan and Shaz Qadeer. 2003. *Thread-Modular Model Checking*. In: *SPIN*, LNCS vol. 2648. Springer. [DOI:10.1007/3-540-44829-2_14](#) (cit. on pp. 11, 42).
- [Flanagan et al. 2002] Cormac Flanagan, Shaz Qadeer, and Sanjit A. Seshia. 2002. *A Modular Checker for Multithreaded Programs*. In: *CAV*, LNCS vol. 2404. Springer. [DOI:10.1007/3-540-45657-0_14](#) (cit. on p. 106).
- [Fomitchev and Ruppert 2004] Mikhail Fomitchev and Eric Ruppert. 2004. *Lock-free linked lists and skip lists*. In: *PODC*, ACM. [DOI:10.1145/1011767.1011776](#) (cit. on p. 102).
- [Foster et al. 2008] I. Foster, Y. Zhao, I. Raicu, and S. Lu. 2008. *Cloud Computing and Grid Computing 360-Degree Compared*. In: *2008 Grid Computing Environments Workshop*, [DOI:10.1109/GCE.2008.4738445](#) (cit. on p. 110).
- [Foster et al. 2002] Jeffrey S. Foster, Tachio Terauchi, and Alexander Aiken. 2002. *Flow-Sensitive Type Qualifiers*. In: *PLDI*, ACM. [DOI:10.1145/512529.512531](#) (cit. on pp. 81, 104).
- [Fraser 2004] Keir Fraser. 2004. *Practical lock-freedom*. PhD thesis. University of Cambridge, UK. <https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.599193> (cit. on pp. 18, 21, 102).
- [Freund and Qadeer 2004] Stephen N. Freund and Shaz Qadeer. 2004. *Checking Concise Specifications for Multithreaded Software*. In: *Journal of Object Technology* 3 (6). [DOI:10.5381/jot.2004.3.6.a4](#) (cit. on p. 106).
- [Fu et al. 2010] 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. [DOI:10.1007/978-3-642-15375-4_27](#) (cit. on pp. 105, 106).
- [Gidenstam et al. 2005] Anders Gidenstam, Marina Papatriantafyllou, Håkan Sundell, and Philippas Tsigas. 2005. *Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting*. In: *ISPAN*, IEEE Computer Society. [DOI:10.1109/ISPAN.2005.42](#) (cit. on p. 103).

- [**Gilbert and Lynch 2002**] Seth Gilbert and Nancy A. Lynch. 2002. *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. In: *SIGACT News* 33 (2). [✎ DOI:10.1145/564585.564601](#) (cit. on p. 110).
- [**Gilbert et al. 2010**] Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. 2010. *Rambo: a robust, reconfigurable atomic memory service for dynamic networks*. In: *Distributed Comput.* 23 (4). [✎ DOI:10.1007/s00446-010-0117-1](#) (cit. on p. 110).
- [**Gong and Wing 1990**] Chun Gong and Jeannette M. Wing. 1990. *A Library of Concurrent Objects and Their Proofs of Correctness*. In: [✎ DOI:10.1184/R1/6587534.v1](#) (cit. on p. 102).
- [**Gotsman et al. 2007**] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. 2007. *Thread-modular shape analysis*. In: *PLDI*, ACM. [✎ DOI:10.1145/1250734.1250765](#) (cit. on pp. 11, 52).
- [**Gotsman et al. 2013**] Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. 2013. *Verifying Concurrent Memory Reclamation Algorithms with Grace*. In: *ESOP*, LNCS vol. 7792. Springer. [✎ DOI:10.1007/978-3-642-37036-6_15](#) (cit. on pp. 10, 23, 46, 105, 106).
- [**Gotsman et al. 2016**] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. *'Cause I'm strong enough: reasoning about consistency choices in distributed systems*. In: *POPL*, ACM. [✎ DOI:10.1145/2837614.2837625](#) (cit. on p. 110).
- [**Grädel et al. 2002**] Erich Grädel, Wolfgang Thomas, and Thomas Wilke. 2002. *Automata, Logics, and Infinite Games: A Guide to Current Research*. LNCS vol. 2500. Springer. [✎ DOI:10.1007/3-540-36387-4](#) (cit. on p. 93).
- [**Greenwald 1999**] Michael Greenwald. 1999. *Non-Blocking Synchronization and System Design*. PhD thesis. [✎ http://i.stanford.edu/pub/cstr/reports/cs/tr/99/1624/CS-TR-99-1624.pdf](#) (cit. on pp. 15, 102).
- [**Groves 2007**] Lindsay Groves. 2007. *Reasoning about Nonblocking Concurrency using Reduction*. In: *ICECCS*, IEEE Computer Society. [✎ DOI:10.1109/ICECCS.2007.39](#) (cit. on p. 106).
- [**Groves 2008**] Lindsay Groves. 2008. *Verifying Michael and Scott's Lock-Free Queue Algorithm using Trace Reduction*. In: *CATS*, CRPIT vol. 77. Australian Computer Society. [✎ https://crpit.com/abstracts/CRPITV77Groves.html](#) (cit. on p. 106).
- [**Haas 2015**] Andreas Haas. 2015. *Fast Concurrent Data Structures Through Timestamping*. PhD thesis. [✎ http://www.cs.uni-salzburg.at/~ahaas/papers/thesis.pdf](#) (cit. on p. 102).
- [**Haas et al. 2015**] Andreas Haas, Thomas Hütter, Christoph M. Kirsch, Michael Lippautz, Mario Preishuber, and Ana Sokolova. 2015. *Scal: A Benchmarking Suite for Concurrent Data Structures*. In: *NETYS*, LNCS vol. 9466. Springer. [✎ DOI:10.1007/978-3-319-26850-7_1](#) (cit. on p. 102).
- [**Harris 2001**] Timothy L. Harris. 2001. *A Pragmatic Implementation of Non-blocking Linked-Lists*. In: *DISC*, LNCS vol. 2180. Springer. [✎ DOI:10.1007/3-540-45414-4_21](#) (cit. on pp. 8, 21, 35, 103).
- [**Harris et al. 2002**] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. *A Practical Multi-word Compare-and-Swap Operation*. In: *DISC*, LNCS vol. 2508. Springer. [✎ DOI:10.1007/3-540-36108-1_18](#) (cit. on p. 17).
- [**Hart et al. 2007**] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. *Performance of memory reclamation for lockless synchronization*. In: *J. Parallel Distrib. Comput.* 67 (12). [✎ DOI:10.1016/j.jpdc.2007.04.010](#) (cit. on pp. 23, 103).

- [**Hawblitzel et al. 2015**] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. *Automated and Modular Refinement Reasoning for Concurrent Programs*. In: CAV, LNCS vol. 9207. Springer. ⚡ DOI:10.1007/978-3-319-21668-3_26 (cit. on pp. 97, 107).
- [**Haziza et al. 2016**] Frédéric Haziza, Lukás Holík, Roland Meyer, and Sebastian Wolff. 2016. *Pointer Race Freedom*. In: VMCAI, LNCS vol. 9583. Springer. ⚡ DOI:10.1007/978-3-662-49122-5_19 (cit. on p. 109).
- [**Hemed et al. 2015**] Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. 2015. *Modular Verification of Concurrency-Aware Linearizability*. In: DISC, LNCS vol. 9363. Springer. ⚡ DOI:10.1007/978-3-662-48653-5_25 (cit. on p. 106).
- [**Hendler et al. 2010**] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. *Flat combining and the synchronization-parallelism tradeoff*. In: SPAA, ACM. ⚡ DOI:10.1145/1810479.1810540 (cit. on p. 102).
- [**Hendler et al. 2004**] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. *A scalable lock-free stack algorithm*. In: SPAA, ACM. ⚡ DOI:10.1145/1007912.1007944 (cit. on p. 102).
- [**Hennessy and Patterson 2012**] John L. Hennessy and David A. Patterson. 2012. *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann. ⚡ ISBN:978-0-12-383872-8 (cit. on p. 110).
- [**Henzinger et al. 1995**] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. 1995. *Computing Simulations on Finite and Infinite Graphs*. In: FOCS, IEEE Computer Society. ⚡ DOI:10.1109/SFCS.1995.492576 (cit. on p. 45).
- [**Henzinger et al. 2003**] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2003. *Software Verification with BLAST*. In: SPIN, LNCS vol. 2648. Springer. ⚡ DOI:10.1007/3-540-44829-2_17 (cit. on p. 103).
- [**Henzinger et al. 2013a**] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. 2013. *Quantitative relaxation of concurrent data structures*. In: POPL, ACM. ⚡ DOI:10.1145/2429069.2429109 (cit. on pp. 8, 102).
- [**Henzinger et al. 2013b**] Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2013. *Aspect-Oriented Linearizability Proofs*. In: CONCUR, LNCS vol. 8052. Springer. ⚡ DOI:10.1007/978-3-642-40184-8_18 (cit. on p. 106).
- [**Herlihy et al. 2005**] Maurice Herlihy, Victor Luchangco, Paul A. Martin, and Mark Moir. 2005. *Nonblocking memory management support for dynamic-sized data structures*. In: ACM Trans. Comput. Syst. 23 (2). ⚡ DOI:10.1145/1062247.1062249 (cit. on p. 103).
- [**Herlihy et al. 2002**] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2002. *The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures*. In: DISC, LNCS vol. 2508. Springer. ⚡ DOI:10.1007/3-540-36108-1_23 (cit. on p. 103).
- [**Herlihy and Shavit 2008**] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann. ⚡ ISBN:978-0-12-370591-4 (cit. on pp. 15, 17, 20, 79, 102, 108).
- [**Herlihy and Wing 1990**] Maurice Herlihy and Jeannette M. Wing. 1990. *Linearizability: A Correctness Condition for Concurrent Objects*. In: ACM Trans. Program. Lang. Syst. 12 (3). ⚡ DOI:10.1145/78969.78972 (cit. on pp. 15, 16).
- [**Holíík et al. 2016**] Lukás Holík, Michal Kotoun, Petr Peringer, Veronika Soková, Marek Trtík, and Tomáš Vojnar. 2016. *Predator Shape Analysis Tool Suite*. In: HVC, LNCS vol. 10028. Springer. ⚡ DOI:10.1007/978-3-319-49052-6_13 (cit. on p. 103).
- [**Holíík et al. 2013**] Lukás Holík, Ondrej Lengál, Adam Rogalewicz, Jirí Simáček, and Tomáš Vojnar. 2013. *Fully Automated Shape Analysis Based on Forest Automata*. In: CAV, LNCS vol. 8044. Springer. ⚡ DOI:10.1007/978-3-642-39799-8_52 (cit. on p. 103).

- [**Holik et al. 2017**] Lukás Holík, Roland Meyer, Tomáš Vojnar, and Sebastian Wolff. 2017. *Effect Summaries for Thread-Modular Analysis - Sound Analysis Despite an Unsound Heuristic*. In: SAS, LNCS vol. 10422. Springer. [✂ DOI:10.1007/978-3-319-66706-5_9](#) (cit. on pp. 12, 59, 106).
- [**Horn and Kroening 2015**] Alex Horn and Daniel Kroening. 2015. *Faster Linearizability Checking via P-Compositionality*. In: FORTE, LNCS vol. 9039. Springer. [✂ DOI:10.1007/978-3-319-19195-9_4](#) (cit. on p. 105).
- [**Hunt and Sands 2006**] Sebastian Hunt and David Sands. 2006. *On Flow-sensitive Security Types*. In: POPL, ACM. [✂ DOI:10.1145/1111037.1111045](#) (cit. on p. 81).
- [**IBM 1983**] IBM. 1983. *IBM System/370 Extended Architecture: Principles of Operation*. Version IBM Publication No. SA22-7085. [✂ https://bitsavers.informatik.uni-stuttgart.de/pdf/ibm/370/princOps/SA22-7085-0_370-XA_Principles_of_Operation_Mar83.pdf](#) (cit. on pp. 16, 19, 20).
- [**Intel Corporation 2016**] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Version 325383-072US. Volume 2A: Instruction Set Reference, A-L. [✂ https://www.intel.de/content/www/de/de/architecture-and-technology/64-ia-32-architectures-software-developer-vol-2a-manual.html](#) (cit. on pp. 16, 17).
- [**ISO 2011**] ISO. 2011. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Standard ISO/IEC 14882:2011. Geneva, CH: International Organization for Standardization. [✂ https://www.iso.org/standard/50372.html](#) (cit. on pp. 9, 18).
- [**Israeli and Rappoport 1993**] Amos Israeli and Lihu Rappoport. 1993. *Efficient Wait-Free Implementation of a Concurrent Priority Queue*. In: WDAG, LNCS vol. 725. Springer. [✂ DOI:10.1007/3-540-57271-6_23](#) (cit. on p. 102).
- [**Jensen et al. 1987**] Eric H. Jensen, Jeffrey M. Broughton, and Gary W. Hagensen. 1987. *A New Approach to Exclusive Data Access in Shared Memory Multiprocessors*. Tech. rep. [✂ https://llnl.primo.exlibrisgroup.com/permalink/01LLNL_INST/1g1o79t/alma991001081569706316](#) (cit. on p. 17).
- [**Jones 1983**] Cliff B. Jones. 1983. *Tentative Steps Toward a Development Method for Interfering Programs*. In: ACM Trans. Program. Lang. Syst. 5 (4). [✂ DOI:10.1145/69575.69577](#) (cit. on pp. 11, 42, 51, 105).
- [**Jones and Muchnick 1979**] Neil D. Jones and Steven S. Muchnick. 1979. *Flow Analysis and Optimization of Lisp-Like Structures*. In: POPL, ACM Press. [✂ DOI:10.1145/567752.567776](#) (cit. on p. 13).
- [**Jonsson 2012**] Bengt Jonsson. 2012. *Using refinement calculus techniques to prove linearizability*. In: Formal Asp. Comput. 24 (4-6). [✂ DOI:10.1007/s00165-012-0250-7](#) (cit. on p. 106).
- [**Jung et al. 2018**] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*. In: J. Funct. Program. 28. [✂ DOI:10.1017/S0956796818000151](#) (cit. on p. 105).
- [**Jung et al. 2015**] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. *Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning*. In: POPL, ACM. [✂ DOI:10.1145/2676726.2676980](#) (cit. on p. 105).
- [**Kang and Jung 2020**] Jeehoon Kang and Jaehwang Jung. 2020. *A marriage of pointer- and epoch-based reclamation*. In: PLDI, ACM. [✂ DOI:10.1145/3385412.3385978](#) (cit. on p. 103).
- [**Khyzha et al. 2017**] Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. 2017. *Proving Linearizability Using Partial Orders*. In: ESOP, LNCS vol. 10201. Springer. [✂ DOI:10.1007/978-3-662-54434-1_24](#) (cit. on p. 106).

- [Kogan and Petrank 2011] Alex Kogan and Erez Petrank. 2011. *Wait-free queues with multiple enqueueers and dequeuers*. In: *PPOPP*, ACM. ✎ DOI:10.1145/1941553.1941585 (cit. on p. 102).
- [Kogan and Petrank 2012] Alex Kogan and Erez Petrank. 2012. *A methodology for creating fast wait-free data structures*. In: *PPOPP*, ACM. ✎ DOI:10.1145/2145816.2145835 (cit. on p. 102).
- [Kragl and Qadeer 2018] Bernhard Kragl and Shaz Qadeer. 2018. *Layered Concurrent Programs*. In: *CAV*, LNCS vol. 10981. Springer. ✎ DOI:10.1007/978-3-319-96145-3_5 (cit. on pp. 98, 107).
- [Kramer and Magee 1990] Jeff Kramer and Jeff Magee. 1990. *The Evolving Philosophers Problem: Dynamic Change Management*. In: *IEEE Trans. Software Eng.* 16 (11). ✎ DOI:10.1109/32.60317 (cit. on p. 110).
- [Krebbers et al. 2018] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. *MoSeL: a general, extensible modal framework for interactive proofs in separation logic*. In: *PACMPL* 2 (ICFP). ✎ DOI:10.1145/3236772 (cit. on p. 9).
- [Krishna et al. 2018] Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2018. *Go with the flow: compositional abstractions for concurrent data structures*. In: *PACMPL* 2 (POPL). ✎ DOI:10.1145/3158125 (cit. on p. 106).
- [Kuru and Gordon 2019] Ismail Kuru and Colin S. Gordon. 2019. *Safe Deferred Memory Reclamation with Types*. In: *ESOP*, LNCS vol. 11423. Springer. ✎ DOI:10.1007/978-3-030-17184-1_4 (cit. on pp. 104, 108).
- [Ladan-Mozes and Shavit 2004] Edya Ladan-Mozes and Nir Shavit. 2004. *An Optimistic Approach to Lock-Free FIFO Queues*. In: *DISC*, LNCS vol. 3274. Springer. ✎ DOI:10.1007/978-3-540-30186-8_9 (cit. on p. 8).
- [Lamport 1979] Leslie Lamport. 1979. *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*. In: *IEEE Trans. Computers* 28 (9). ✎ DOI:10.1109/TC.1979.1675439 (cit. on pp. 38, 110).
- [Lamport and Schneider 1989] Leslie Lamport and Fred B. Schneider. 1989. *Pretending Atomicity*. In: *SRC Research Report* 44. ✎ <https://www.microsoft.com/en-us/research/publication/pretending-atomicity/> (cit. on p. 106).
- [Laviron et al. 2010] Vincent Laviron, Bor-Yuh Evan Chang, and Xavier Rival. 2010. *Separating Shape Graphs*. In: *ESOP*, LNCS vol. 6012. Springer. ✎ DOI:10.1007/978-3-642-11957-6_21 (cit. on p. 103).
- [Levandoski et al. 2013] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. *The Bw-Tree: A B-tree for new hardware platforms*. In: *ICDE*, IEEE Computer Society. ✎ DOI:10.1109/ICDE.2013.6544834 (cit. on p. 102).
- [Leveson and Turner 1993] Nancy G. Leveson and Clark Savage Turner. 1993. *An Investigation of the Therac-25 Accidents*. In: *Computer* 26 (7). ✎ DOI:10.1109/MC.1993.274940 (cit. on p. 8).
- [Liang and Feng 2013] Hongjin Liang and Xinyu Feng. 2013. *Modular verification of linearizability with non-fixed linearization points*. In: *PLDI*, ACM. ✎ DOI:10.1145/2462156.2462189 (cit. on p. 106).
- [Liang et al. 2012] Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. *A rely-guarantee-based simulation for verifying concurrent program transformations*. In: *POPL*, ACM. ✎ DOI:10.1145/2103656.2103711 (cit. on p. 106).
- [Liang et al. 2014] Hongjin Liang, Xinyu Feng, and Ming Fu. 2014. *Rely-Guarantee-Based Simulation for Compositional Verification of Concurrent Program Transformations*. In: *ACM Trans. Program. Lang. Syst.* 36 (1). ✎ DOI:10.1145/2576235 (cit. on p. 106).
- [Lipton 1975] Richard J. Lipton. 1975. *Reduction: A Method of Proving Properties of Parallel Programs*. In: *CACM* 18 (12). ✎ DOI:10.1145/361227.361234 (cit. on pp. 75, 97, 98, 105, 106).

- [Liu et al. 2009] Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. 2009. *Model Checking Linearizability via Refinement*. In: *FM*, LNCS vol. 5850. Springer. [DOI:10.1007/978-3-642-05089-3_21](#) (cit. on p. 105).
- [Liu et al. 2013] Yang Liu, Wei Chen, Yanhong A. Liu, Jun Sun, Shao Jie Zhang, and Jin Song Dong. 2013. *Verifying Linearizability via Optimized Refinement Checking*. In: *IEEE Trans. Software Eng.* 39 (7). [DOI:10.1109/TSE.2012.82](#) (cit. on p. 105).
- [Lowe 2017] Gavin Lowe. 2017. *Testing for linearizability*. In: *Concurrency and Computation: Practice and Experience* 29 (4). [DOI:10.1002/cpe.3928](#) (cit. on p. 105).
- [Mckenney 2004] Paul E. Mckenney. 2004. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis. [DOI:10.6083/M4GH9FVB](#) (cit. on p. 108).
- [McKenney et al. 2020] Paul E. McKenney, Joel Fernandes, Silas Boyd-Wickizer, and Jonathan Walpole. 2020. *RCU Usage In the Linux Kernel: Eighteen Years Later*. In: *ACM SIGOPS Oper. Syst. Rev.* 54 (1). [DOI:10.1145/3421473.3421481](#) (cit. on p. 108).
- [McKenney and Slingwine 1998] Paul E. McKenney and John D. Slingwine. 1998. *Read-copy Update: Using Execution History to Solve Concurrency Problems*. In: <http://www.rdrop.com/~paulmck/scalability/paper/rclockpdcspproof.pdf> (cit. on p. 21).
- Dinesh P. Mehta and Sartaj Sahni, eds. (2004). *Handbook of Data Structures and Applications*. Chapman and Hall/CRC. [DOI:10.1201/9781420035179](#) (cit. on p. 8).
- [Michael et al. 2021] Maged Michael, Jay Feldblum, and Andres Suarez. 2021. *Facebook Folly*. <https://github.com/facebook/folly/blob/49926b98f5afb5667d0c06807da79d606a6d43c3/folly/synchronization/HazptrHolder.h#L143> (cit. on p. 108).
- [Michael 2002a] Maged M. Michael. 2002. *High performance dynamic lock-free hash tables and list-based sets*. In: *SPAA*, [DOI:10.1145/564870.564881](#) (cit. on pp. 8, 20, 25, 34, 132).
- [Michael 2002b] Maged M. Michael. 2002. *Safe memory reclamation for dynamic lock-free objects using atomic reads and writes*. In: *PODC*, ACM. [DOI:10.1145/571825.571829](#) (cit. on pp. 9, 18, 20, 23, 24, 26–29, 36).
- [Michael 2003] Maged M. Michael. 2003. *CAS-Based Lock-Free Algorithm for Shared Deques*. In: *Euro-Par*, LNCS vol. 2790. Springer. [DOI:10.1007/978-3-540-45209-6_92](#) (cit. on p. 102).
- [Michael 2004] Maged M. Michael. 2004. *Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects*. In: *IEEE Trans. Parallel Distrib. Syst.* 15 (6). [DOI:10.1109/TPDS.2004.8](#) (cit. on pp. 23, 24, 29).
- [Michael and Scott 1995] Maged M. Michael and Michael L. Scott. 1995. *Correction of a Memory Management Method for Lock-Free Data Structures*. Tech. rep. https://www.cs.rochester.edu/u/scott/papers/1995_TR599.pdf (cit. on p. 8).
- [Michael and Scott 1996] Maged M. Michael and Michael L. Scott. 1996. *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*. In: *PODC*, ACM. [DOI:10.1145/248052.248106](#) (cit. on pp. 11, 15, 20, 28, 66).
- [Milner 1971] Robin Milner. 1971. *An Algebraic Definition of Simulation Between Programs*. In: *IJCAI*, William Kaufmann. <https://ijcai.org/Proceedings/71/Papers/044.pdf> (cit. on p. 45).
- [Moir and Shavit 2004] Mark Moir and Nir Shavit. 2004. *Concurrent Data Structures*. In: *Handbook of Data Structures and Applications*, Chapman and Hall/CRC. [DOI:10.1201/9781420035179.ch47](#) (cit. on p. 18).
- [Morrison and Afek 2013] Adam Morrison and Yehuda Afek. 2013. *Fast concurrent queues for x86 processors*. In: *PPOPP*, ACM. [DOI:10.1145/2442516.2442527](#) (cit. on p. 102).

- [Nanevski et al. 2014] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. *Communicating State Transition Systems for Fine-Grained Concurrent Resources*. In: *ESOP*, LNCS vol. 8410. Springer. ✎ DOI:10.1007/978-3-642-54833-8_16 (cit. on p. 105).
- [Natarajan and Mittal 2014] Aravind Natarajan and Neeraj Mittal. 2014. *Fast concurrent lock-free binary search trees*. In: *PPOPP*, ACM. ✎ DOI:10.1145/2555243.2555256 (cit. on p. 102).
- [Natarajan et al. 2020] Aravind Natarajan, Arunmozhi Ramachandran, and Neeraj Mittal. 2020. *FEAST: A Lightweight Lock-free Concurrent Binary Search Tree*. In: *ACM Trans. Parallel Comput.* 7 (2). ✎ DOI:10.1145/3391438 (cit. on p. 102).
- [Necula et al. 2002] George C. Necula, Scott McPeak, and Westley Weimer. 2002. *CCured: Type-safe Retrofitting of Legacy Code*. In: *POPL*, ACM. ✎ DOI:10.1145/503272.503286 (cit. on p. 103).
- [Nikolaev and Ravindran 2019] Ruslan Nikolaev and Binoy Ravindran. 2019. *Hyaline: Fast and Transparent Lock-Free Memory Reclamation*. In: *PODC*, ACM. ✎ DOI:10.1145/3293611.3331575 (cit. on p. 103).
- [Nikolaev and Ravindran 2020] Ruslan Nikolaev and Binoy Ravindran. 2020. *Universal wait-free memory reclamation*. In: *PPoPP*, ACM. ✎ DOI:10.1145/3332466.3374540 (cit. on pp. 9, 17, 102, 103).
- [O’Hearn 2004] Peter W. O’Hearn. 2004. *Resources, Concurrency and Local Reasoning*. In: *CONCUR*, LNCS vol. 3170. Springer. ✎ DOI:10.1007/978-3-540-28644-8_4 (cit. on pp. 11, 52, 105).
- [O’Hearn et al. 2001] Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. 2001. *Local Reasoning about Programs that Alter Data Structures*. In: *CSL*, LNCS vol. 2142. Springer. ✎ DOI:10.1007/3-540-44802-0_1 (cit. on pp. 105, 106).
- [O’Hearn et al. 2010] Peter W. O’Hearn, Noam Rinetzkky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. *Verifying linearizability with hindsight*. In: *PODC*, ACM. ✎ DOI:10.1145/1835698.1835722 (cit. on pp. 32, 106).
- [Oracle 2020] Oracle. 2020. *Java® Platform, Standard Edition & Java Development Kit*. Version Version 15 API Specification. ✎ <https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/concurrent/ConcurrentLinkedQueue.html> (cit. on p. 28).
- [Owens 2010] Scott Owens. 2010. *Reasoning about the Implementation of Concurrency Abstractions on x86-TSO*. In: *ECOOP*, LNCS vol. 6183. Springer. ✎ DOI:10.1007/978-3-642-14107-2_23 (cit. on p. 110).
- [Owicki and Gries 1976] Susan S. Owicki and David Gries. 1976. *An Axiomatic Proof Technique for Parallel Programs I*. In: *Acta Inf.* 6. ✎ DOI:10.1007/BF00268134 (cit. on pp. 11, 42, 51, 76, 81).
- [Parkinson et al. 2007] Matthew J. Parkinson, Richard Bornat, and Peter W. O’Hearn. 2007. *Modular verification of a non-blocking stack*. In: *POPL*, ACM. ✎ DOI:10.1145/1190216.1190261 (cit. on p. 106).
- [Pierce 2002] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press. ✎ ISBN:0-262-16209-1 (cit. on pp. 80, 94).
- [Plotkin 1981] Gordon D. Plotkin. 1981. *A structural approach to operational semantics*. DAIMI Report FN-19. Computer Science Department, Aarhus University. ✎ <https://web.eecs.umich.edu/~weimerw/2008-615/reading/plotkin81structural.pdf> (cit. on p. 39).
- [Prakash et al. 1994] Sundeep Prakash, Yann-Hang Lee, and Theodore Johnson. 1994. *A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap*. In: *IEEE Trans. Computers* 43 (5). ✎ DOI:10.1109/12.280802 (cit. on p. 32).
- [Ramachandran and Mittal 2015] Arunmozhi Ramachandran and Neeraj Mittal. 2015. *A Fast Lock-Free Internal Binary Search Tree*. In: *ICDCN*, ACM. ✎ DOI:10.1145/2684464.2684472 (cit. on p. 102).

- [**Ramalhete and Correia 2017**] Pedro Ramalhete and Andreia Correia. 2017. *Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation*. In: *SPAA*, ACM. ✎ DOI:10.1145/3087556.3087588 (cit. on p. 103).
- [**Reynolds 2002**] John C. Reynolds. 2002. *Separation Logic: A Logic for Shared Mutable Data Structures*. In: *LICS*, IEEE. ✎ DOI:10.1109/LICS.2002.1029817 (cit. on pp. 105, 106).
- [**Schellhorn et al. 2012**] Gerhard Schellhorn, Heike Wehrheim, and John Derrick. 2012. *How to Prove Algorithms Linearisable*. In: *CAV*, LNCS vol. 7358. Springer. ✎ DOI:10.1007/978-3-642-31424-7_21 (cit. on p. 106).
- [**Segalov et al. 2009**] 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. ✎ DOI:10.1007/978-3-642-10672-9_5 (cit. on p. 106).
- [**Sergey et al. 2015a**] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. *Mechanized verification of fine-grained concurrent programs*. In: *PLDI*, ACM. ✎ DOI:10.1145/2737924.2737964 (cit. on p. 106).
- [**Sergey et al. 2015b**] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. *Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity*. In: *ESOP*, LNCS vol. 9032. Springer. ✎ DOI:10.1007/978-3-662-46669-8_14 (cit. on p. 106).
- [**Sergey et al. 2018**] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. *Programming and proving with distributed protocols*. In: *Proc. ACM Program. Lang.* 2 (POPL). ✎ DOI:10.1145/3158116 (cit. on p. 110).
- [**Sethi et al. 2013**] Divyot Sethi, Muralidhar Talupur, and Sharad Malik. 2013. *Model Checking Unbounded Concurrent Lists*. In: *SPIN*, LNCS vol. 7976. Springer. ✎ DOI:10.1007/978-3-642-39176-7_20 (cit. on p. 106).
- [**Sewell et al. 2010**] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. *x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors*. In: *Commun. ACM* 53 (7). ✎ DOI:10.1145/1785414.1785443 (cit. on p. 110).
- [**Shafiei 2015**] Niloufar Shafiei. 2015. *Non-Blocking Doubly-Linked Lists with Good Amortized Complexity*. In: *OPODIS*, LIPIcs vol. 46. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. ✎ DOI:10.4230/LIPIcs.OPODIS.2015.35 (cit. on p. 102).
- [**Shann et al. 2000**] Chien-Hua Shann, Ting-Lu Huang, and Cheng Chen. 2000. *A Practical Nonblocking Queue Algorithm Using Compare-and-Swap*. In: *ICPADS*, IEEE Computer Society. ✎ DOI:10.1109/ICPADS.2000.857731 (cit. on p. 102).
- [**Silberschatz et al. 2020**] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. 2020. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company. ✎ ISBN:9780078022159 (cit. on p. 110).
- [**Stallings 2013**] William Stallings. 2013. *Computer Organization and Architecture - Designing for Performance (9. ed.)* Pearson / Prentice Hall. ✎ ISBN:9789332518704 (cit. on p. 16).
- [**Stenström 1990**] Per Stenström. 1990. *A Survey of Cache Coherence Schemes for Multiprocessors*. In: *Computer* 23 (6). ✎ DOI:10.1109/2.55497 (cit. on p. 110).
- [**Ströder et al. 2017**] 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*. In: *J. Autom. Reasoning* 58 (1). ✎ DOI:10.1007/s10817-016-9389-x (cit. on p. 103).
- [**Sundell and Tsigas 2003**] Håkan Sundell and Philippas Tsigas. 2003. *Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems*. In: *IPDPS*, IEEE Computer Society. ✎ DOI:10.1109/IPDPS.2003.1213189 (cit. on p. 102).

- [Sundell and Tsigas 2004] Håkan Sundell and Philippas Tsigas. 2004. *Lock-Free and Practical Doubly Linked List-Based Deques Using Single-Word Compare-and-Swap*. In: *OPODIS*, LNCS vol. 3544. Springer. [✎ DOI:10.1007/11516798_18](#) (cit. on p. 102).
- [Svendsen and Birkedal 2014] Kasper Svendsen and Lars Birkedal. 2014. *Impredicative Concurrent Abstract Predicates*. In: *ESOP*, LNCS vol. 8410. Springer. [✎ DOI:10.1007/978-3-642-54833-8_9](#) (cit. on p. 105).
- [Tanenbaum and Bos 2014] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems*. 4th. USA: Prentice Hall Press. [✎ ISBN:013359162X](#) (cit. on p. 108).
- [Ter-Gabrielyan et al. 2019] Arshavir Ter-Gabrielyan, Alexander J. Summers, and Peter Müller. 2019. *Modular verification of heap reachability properties in separation logic*. In: *PACMPL 3 (OOPSLA)*. [✎ DOI:10.1145/3360547](#) (cit. on p. 106).
- [Tofan et al. 2011] Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif. 2011. *Formal Verification of a Lock-Free Stack with Hazard Pointers*. In: *ICTAC*, LNCS vol. 6916. Springer. [✎ DOI:10.1007/978-3-642-23283-1_16](#) (cit. on p. 106).
- [Travkin et al. 2013] Oleg Travkin, Annika Mütze, and Heike Wehrheim. 2013. *SPIN as a Linearizability Checker under Weak Memory Models*. In: *Haifa Verification Conference*, LNCS vol. 8244. Springer. [✎ DOI:10.1007/978-3-319-03077-7_21](#) (cit. on p. 105).
- [Treiber 1986] R. Kent Treiber. 1986. *Systems programming: coping with parallelism*. Tech. rep. RJ 5118. IBM. [✎ https://do.minoweb.draco.res.ibm.com/reports/rj5118.pdf](#) (cit. on pp. 15, 19, 20, 26, 27).
- [Tsigas and Zhang 2001] Philippas Tsigas and Yi Zhang. 2001. *A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems*. In: *SPAA*, ACM. [✎ DOI:10.1145/378580.378611](#) (cit. on p. 102).
- [Turon et al. 2013] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. *Unifying Refinement and Hoare-style Reasoning in a Logic for Higher-order Concurrency*. In: *ICFP*, ACM. [✎ DOI:10.1145/2544174.2500600](#) (cit. on p. 105).
- [Turon et al. 2014] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. *GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation*. In: *OOPSLA*, ACM. [✎ DOI:10.1145/2660193.2660243](#) (cit. on p. 105).
- [Vafeiadis 2009] Viktor Vafeiadis. 2009. *Shape-Value Abstraction for Verifying Linearizability*. In: *VMCAI*, LNCS vol. 5403. Springer. [✎ DOI:10.1007/978-3-540-93900-9_27](#) (cit. on pp. 92, 99).
- [Vafeiadis 2010a] Viktor Vafeiadis. 2010. *RGSep Action Inference*. In: *VMCAI*, LNCS vol. 5944. Springer. [✎ DOI:10.1007/978-3-642-11319-2_25](#) (cit. on pp. 9, 11, 92, 99, 102, 104–106).
- [Vafeiadis 2010b] Viktor Vafeiadis. 2010. *Automatically Proving Linearizability*. In: *CAV*, LNCS vol. 6174. Springer. [✎ DOI:10.1007/978-3-642-14295-6_40](#) (cit. on pp. 9, 11, 15, 92, 99, 102, 104, 106).
- [Vafeiadis and Parkinson 2007] Viktor Vafeiadis and Matthew J. Parkinson. 2007. *A Marriage of Rely/Guarantee and Separation Logic*. In: *CONCUR*, LNCS vol. 4703. Springer. [✎ DOI:10.1007/978-3-540-74407-8_18](#) (cit. on pp. 11, 52, 105).
- [Vardi 1987] Moshe Y. Vardi. 1987. *Verification of Concurrent Programs: The Automata-Theoretic Framework*. In: *LICS*, IEEE Computer Society (cit. on p. 49).
- [Vechev and Yahav 2008] Martin T. Vechev and Eran Yahav. 2008. *Deriving linearizable fine-grained concurrent objects*. In: *PLDI*, ACM. [✎ DOI:10.1145/1375581.1375598](#) (cit. on pp. 30–33, 105).
- [Vechev et al. 2009] Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2009. *Experience with Model Checking Linearizability*. In: *SPIN*, LNCS vol. 5578. Springer. [✎ DOI:10.1007/978-3-642-02652-2_21](#) (cit. on p. 106).

- [**von Gleissenthall et al. 2019**] Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. *Pretend synchrony: synchronous verification of asynchronous distributed programs*. In: *Proc. ACM Program. Lang.* 3 (POPL). ✎ DOI:10.1145/3290372 (cit. on p. 110).
- [**Wen et al. 2018**] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. *Interval-based memory reclamation*. In: *PPOPP*, ACM. ✎ DOI:10.1145/3178487.3178488 (cit. on p. 103).
- [**Wilcox et al. 2015**] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. *Verdi: a framework for implementing and formally verifying distributed systems*. In: *PLDI*, ACM. ✎ DOI:10.1145/2737924.2737958 (cit. on p. 110).
- [**Wirth 1978**] Niklaus Wirth. 1978. *Algorithms + Data Structures = Programs*. USA: Prentice Hall PTR. ✎ ISBN:0130224189 (cit. on p. 8).
- [**Wu et al. 2016**] Hao Wu, Xiaoxiao Yang, and Joost-Pieter Katoen. 2016. *Performance Evaluation of Concurrent Data Structures*. In: *SETTA*, LNCS vol. 9984. ✎ DOI:10.1007/978-3-319-47677-3_3 (cit. on p. 8).
- [**Wulf et al. 2006**] 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. ✎ DOI:10.1007/11817963_5 (cit. on p. 94).
- [**Yang and Wrigstad 2017**] Albert Mingkun Yang and Tobias Wrigstad. 2017. *Type-assisted automatic garbage collection for lock-free data structures*. In: *ISMM*, ACM. ✎ DOI:10.1145/3092255.3092274 (cit. on p. 103).
- [**Yang et al. 2008**] 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. ✎ DOI:10.1007/978-3-540-70545-1_36 (cit. on p. 103).
- [**Yang et al. 2017**] Xiaoxiao Yang, Joost-Pieter Katoen, Huimin Lin, and Hao Wu. 2017. *Verifying Concurrent Stacks by Divergence-Sensitive Bisimulation*. In: *CoRR* abs/1701.06104. ✎ <https://arxiv.org/abs/1701.06104> (cit. on p. 105).
- [**Yoshida 2013a**] Junko Yoshida. 2013. *Toyota Case: Single Bit Flip That Killed*. ✎ <https://www.eetimes.com/toyota-case-single-bit-flip-that-killed/> (cit. on p. 8).
- [**Yoshida 2013b**] Junko Yoshida. 2013. *Toyota Case: Vehicle Testing Confirms Fatal Flaws*. ✎ <https://www.eetimes.com/toyota-case-vehicle-testing-confirms-fatal-flaws/> (cit. on p. 8).
- [**Zhang 2011**] Shao Jie Zhang. 2011. *Scalable automatic linearizability checking*. In: *ICSE*, ACM. ✎ DOI:10.1145/1985793.1986037 (cit. on p. 105).
- [**Zhu et al. 2015**] He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. *Poling: SMT Aided Linearizability Proofs*. In: *CAV (2)*, LNCS vol. 9207. Springer. ✎ DOI:10.1007/978-3-319-21668-3_1 (cit. on pp. 15, 106).

Acknowledgements

First and foremost, I would like to thank my supervisor Roland Meyer for accepting me as his PhD student and guiding me through the sometimes rough and bewildering waters that are research. I am grateful for his interest in the topic, his support of my research, and his constant indomitable will to push our results beyond practicability to theoretic elegance and simplicity.

I sincerely thank Rupak Majumdar and Constantin Enea for accepting to review this thesis.

I am deeply indebted to my parents, Evelin and Martin, for their constant support. Without them, I would have had none of the opportunities that lead to this work.

Last but not least, I thank all my partners in crime when it came to extracurricular activities, in order of appearance: Thomas Lottermann, Manuel Dossinger, Sebastian Henningsen, Frederik Walk, Simon Birnbach, Jana Lampe, Adrian Leva, Sarah Dossinger, Michael Hohenstein, Marvin Huber, Sebastian Schumb, Stefan Templin, Phillip Schon, Sven Kautz, Peter Chini, Sebastian Muskalla, Florian Furbach, Emanuele D'Ossualdo, Prakash Saivasan, Elisabeth Neumann, Sören van der Wall, Mike Becker, Thomas Haas, Elaine Anklam, and Johannes Mohr.

Appendices

Additional Material

We extend selected parts of our development. Appendix A.1 gives a formal account of the compositionality result from Chapter 5. Appendix A.2 presents a specification for the hazard pointer technique that supports transferring protections. As such, the specification expands on the discussions from Sections 5.2 and 8.4. Appendix A.3 discusses a relaxation of strong pointer races that allows for the same reduction results while avoiding false alarms in practice. The relaxation allows constants, like `NULL`, to be compared to invalid pointers in assumptions. For the full meta theory developed during this thesis, refer to Appendix B (and the proofs in Appendix C).

A.1 Compositionality

We revisit the compositional verification results from Chapter 5. Consider some $P(R)$, a data structure P using an SMR implementation R . As stated informally in Chapter 5, we require that the only influence that P is subject to are the free commands that R performs. More precisely, we require a separation of the memory such that P does not access the memory belonging to R , and vice versa. The memory separation is induced by a partitioning of

- (i) the program variables $Var = PVar \cup DVar$ into the variables of P and R , $Var = Var^P \uplus Var^R$, and
- (ii) the pointer selectors $Sel = PSel \cup DSel$ into the selectors of P and R , $Sel = Sel^P \uplus Sel^R$, such that we have $a.sel \in Sel^Q$ iff $b.sel \in Sel^Q$ for all $a, b \in Addr$ and $Q \in \{P, R\}$.

We may write $sel \in Sel^Q$ instead of $a.sel \in Sel^Q$ as membership is independent of the address a . Hereafter, we assume a fixed partitioning the precise form of which does not matter. The induced memory separation is $m_\tau = m_\tau^P \uplus m_\tau^R$ defined by

$$m_\tau^P = m_\tau \downarrow_{Var^P \cup Sel^P} \quad \text{and} \quad m_\tau^R = m_\tau \downarrow_{Var^R \cup Sel^R}.$$

Then, a *separation violation* is an action that does not respect the memory separation, i.e., an action of P that accesses variables or selectors of R , or vice versa. In order to know whether an action stems from executing P or R , we extend the SOS transition relation and write $\vdash_{Q,t}$ to indicate that thread t is taking a step due to $Q \in \{P, R\}$. Technically, we have $\vdash_{R,t}$ if Rule (SOS-STD-SMR) is involved in the derivation of the program step and $\vdash_{P,t}$ otherwise. To simplify our development when it comes to function calls, we relax separation violations and allow R to read out the variables passed to invocations. We introduce a set of *interface variables* $IVar \subseteq Var^P \setminus shared$ and assume that invocations $in:func(r_1, \dots, r_n)$ contain interface variables only, $r_i \in IVar$ for all $1 \leq i \leq n$.

Definition A.1. Program step $(pc, \tau) \rightarrow_{Q,t} (pc', \tau.act)$ with $act = \langle t, com, up \rangle$ is a separation violation if (i) com assigns to variable $x \notin Var^Q$, (ii) com contains variable $x \notin Var^Q \cup IVar$, or (iii) com contains selector $x.sel$ with $sel \notin Sel^Q$. We say τ contains a separation violation. \square

Separation violations in $P(R)$ can be found by a simple syntactic analysis of the program code. The data structure and SMR implementations from Chapter 2 are free from separation violations. We believe that our results can be lifted to more involved memory separations.

Hereafter, it will be convenient to access the control locations of P and R separately. To that end, we define $pc_1 \circ pc_2$ as well as $ctrl^P(\tau)$ and $ctrl^R(\tau)$ as follows:

$$\begin{aligned} pc_1 \circ pc_2 &:= \lambda t. pc_1(t) \circ pc_2(t) \\ \text{and} \quad ctrl^P(\tau) &:= \{ pc_1 \mid \exists pc_2. pc_1 \circ pc_2 \in ctrl(\tau) \} \\ \text{and} \quad ctrl^R(\tau) &:= \{ pc_2 \mid \exists pc_1. pc_1 \circ pc_2 \in ctrl(\tau) \} \end{aligned}$$

Then, the set of program locations of thread t are $ctrl^Q(\tau)(t) := \{ pc(t) \mid pc \in ctrl^Q(\tau) \}$.

We formulate the requirements about the most general client MGC . We require that the MGC can mimic invocations, assignments to interface variables, and allocations performed by P . More specifically, we assume that for all threads t and $\sigma \in \llbracket MGC(R) \rrbracket_{Adr}^{Adr}$ there is $act = \langle t, com, up \rangle$ such that $\sigma.act \in \llbracket MGC(R) \rrbracket_{Adr}^{Adr}$, provided one of the following applies:

- (i) $com \equiv \text{in:func}(r_1, \dots, r_n)$ and $r_i \in IVar$ and $m_\sigma(r_i) \neq \text{seg}$ and $\text{skip} \in ctrl^R(\sigma)(t)$,
- (ii) $com \equiv \text{re:func}$ and $\text{await re:func} \in ctrl^R(\sigma)(t)$,
- (iii) $com \equiv p := \text{malloc}$, or
- (iv) $up = [p \mapsto a]$ and $p \in IVar$ and $a \in \text{used}(\sigma)$.

The first two properties allow us to mimic function invocations and responses. The third property allows us to mimic the allocations performed by P . The last property allows us to update the interface variables using the addresses that the MGC has already allocated and thus holds a pointer to. Here, we assume that the MGC takes its variables from Var^P and define:

$$\text{used}(\sigma) := \{ m_{\sigma_1.act}(p) \mid \exists \sigma_2. \sigma = \sigma_1.act.\sigma_2 \wedge act = \langle \bullet, p := \text{malloc}, \bullet \rangle \wedge p \in Var^P \}$$

We use a single action act for simplicity; a generalization to sequences of actions that mimic act on the interface variables is straight forward. We now show that the simpler MGC over-approximates P 's usage of R . Hence, $R \models \mathcal{O}$ guarantees that R adheres to the specification \mathcal{O} when used by P .

Theorem A.2 (\Leftarrow Proof C.5). Let $\tau \in \llbracket P(R) \rrbracket_{Adr}^{Adr}$ be free from separation violations. Then, there is $\sigma \in \llbracket MGC(R) \rrbracket_{Adr}^{Adr}$ such that (i) $ctrl^R(\tau) \subseteq ctrl^R(\sigma)$, (ii) $m_\tau^R = m_\sigma^R$, (iii) $m_\tau \downarrow_{IVar} = m_\sigma \downarrow_{IVar}$, (iv) $\mathcal{H}(\tau) = \mathcal{H}(\sigma)$, (v) $\text{fresh}_\tau \subseteq \text{fresh}_\sigma$, (vi) $\text{freed}_\tau \subseteq \text{freed}_\sigma$, and (vii) $\text{used}(\tau) \subseteq \text{used}(\sigma)$. \square

Corollary A.3 (\Leftarrow Proof C.6). We have $\mathcal{H}(\llbracket P(R) \rrbracket_{Adr}^{Adr}) \subseteq \mathcal{H}(\llbracket MGC(R) \rrbracket_{Adr}^{Adr})$ provided $\llbracket P(R) \rrbracket_{Adr}^{Adr}$ is free from separation violations. \square

Now, we are ready to show that P can be verified against \mathcal{O} rather than R , provided that $P(R)$ is free from separation violations and that $R \models \mathcal{O}$ holds. More specifically, we show that $P(R)$ and $P(\mathcal{O})$ reach the same control locations.

Theorem A.4 (cf. Proof C.7). Let $\tau \in \llbracket P(R) \rrbracket_{Adr}^{Adr}$ be free from separation violations. If $R \models \mathcal{O}$, then there is some $\sigma \in \mathcal{O} \llbracket P \rrbracket_{Adr}^{Adr}$ with (i) $ctrl^P(\tau) \subseteq ctrl(\sigma)$, (ii) $m_\tau^P = m_\sigma^P$, (iii) $\mathcal{H}(\tau) = \mathcal{H}(\sigma)$, (iv) $fresh_\tau \subseteq fresh_\sigma$, (v) $freed_\tau \subseteq freed_\sigma$, and (vi) $retired_\tau \subseteq retired_\sigma$. \square

In Chapter 5 we introduced the predicate $good(\tau)$ that tests whether or not a computation τ reaches a bad program location which we assumed to be in P . Formally, we let $Fault$ be the bad program locations and define the $good(\tau)$ predicate by $good(\tau) : \iff ctrl^P(\tau) \cap Fault = \emptyset$. Then, Theorems 5.10 and 5.11 are consequences of the above Theorem A.4.

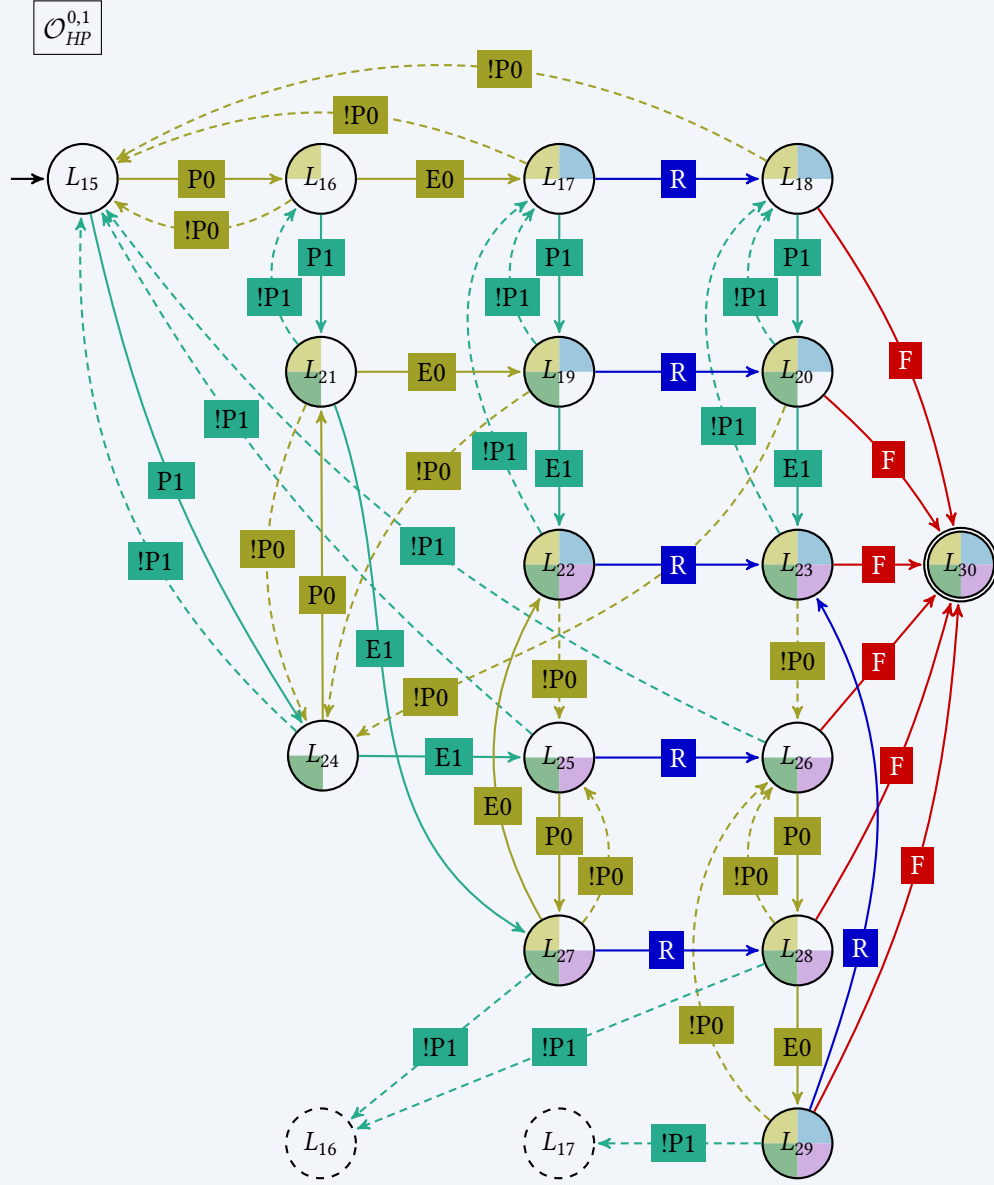
A.2 Hazard Pointer Specification





Recall from Section 5.2 that there are two SMR automata specifying the hazard pointer technique with two hazard pointers per thread: $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$ and $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^{0,1}$. We used the former in the example from Figure 8.20 for typing Micheal&Scott's queue. However, $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$ does not support the transfer of protections among hazard pointers (cf. Section 2.3.3). This feature is crucial for more complicated data structures, like Micheal's set [Michael 2002a]. To that end, we use the SMR automaton $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^{0,1}$. The definition of $\mathcal{O}_{HP}^{0,1}$ is given Figure A.5. Intuitively, $\mathcal{O}_{HP}^{0,1}$ is the cross-product $\mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$ with additional transitions for tracking the correlation among protections and for transferring the protection of the 0-th hazard pointer into the 1-st hazard pointer. More precisely, locations L_{16} to L_{23} deal with protections of the 0-th hazard pointer. Taking just locations L_{15} to L_{18} and L_{30} corresponds to \mathcal{O}_{HP}^0 . If the same address is additionally protected with the 1-st hazard pointer, then $\mathcal{O}_{HP}^{0,1}$ moves to locations L_{19} to L_{21} after $protect_1$ is invoked and to locations L_{22} and L_{23} after $protect_1$ returns. When the protection of the 0-th hazard pointer is revoked, the protection is transferred to the 1-st hazard pointer. In $\mathcal{O}_{HP}^{0,1}$ this is encoded by the transitions from L_{22} to L_{25} and L_{23} to L_{26} . Locations L_{24} to L_{29} are the dual of L_{16} to L_{23} : they track the protections of the 1-st hazard pointer, similarly to \mathcal{O}_{HP}^1 . In particular, location L_{26} prevents frees of the tracked address. That is, the transition from L_{23} to L_{26} transfers the protection indeed. However, when the 1-st hazard pointer is revoked, no transfer takes place. This is reflected by location L_{29} transitioning to L_{17} instead of L_{18} . Indeed, L_{17} allows the tracked address to be freed while L_{18} does not. Hence, $\mathcal{O}_{HP}^{0,1}$ faithfully specifies the transfer of hazard pointers.

When using $\mathcal{O}_{HP}^{0,1}$ with the type system from Chapter 8, we use four HP-specific guarantees $\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3, \mathbb{E}_4$ the locations of which are marked in Figure A.5. Their meaning is similar to what we have seen in the aforementioned typing of Michael&Scott's queue from Figure 8.20. Guarantee \mathbb{E}_1 encodes that an invocation of $protect_0$ has been issued. Guarantee \mathbb{E}_2 encodes that the invocation has returned. Formally, we have:

$$\emptyset, x, in:protect_0(x) \rightsquigarrow \mathbb{E}_1 \quad \text{and} \quad \mathbb{E}_1, x, re:protect_0 \rightsquigarrow \mathbb{E}_2.$$

Figure A.5: Hazard pointer specification $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^{0,1}$ for two hazard pointer per thread supporting the transfer of protections. The HP-specific guarantees $\mathbb{E}_0, \mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3$ are needed when typing.



 $\in Loc(\mathbb{E}_1)$	 $\in Loc(\mathbb{E}_2)$	 $\in Loc(\mathbb{E}_3)$	 $\in Loc(\mathbb{E}_4)$
$P0 \coloneqq \text{in:protect}_0(t, a), t = z_t \wedge a = z_a$	$P1 \coloneqq \text{in:protect}_1(t, a), t = z_t \wedge a = z_a$	$!P0 \coloneqq \text{in:protect}_0(t, a), t = z_t \wedge a \neq z_a$	$!P1 \coloneqq \text{in:protect}_1(t, a), t = z_t \wedge a \neq z_a$
$!P0 \coloneqq \text{in:protect}_0(t, a), t = z_t \wedge a \neq z_a$	$!P1 \coloneqq \text{in:protect}_1(t, a), t = z_t \wedge a \neq z_a$	$\text{in:unprotect}_0(t), t = z_t$	$\text{in:unprotect}_1(t), t = z_t$
$E0 \coloneqq \text{re:protect}(t), t = z_t$	$E1 \coloneqq \text{re:protect}(t), t = z_t$	$R \coloneqq \text{in:retire}(t, a), a = z_a$	$F \coloneqq \text{free}(a), a = z_a$

Similarly, guarantees $\mathbb{E}_3, \mathbb{E}_4$ deal with protect_1 . The protection is successful if it is issued while the address is active:

$$\mathbb{E}_2 \wedge \mathbb{A} \rightsquigarrow \mathbb{S} \quad \text{and} \quad \mathbb{E}_4 \wedge \mathbb{A} \rightsquigarrow \mathbb{S}.$$

We omit an explicit construction of the cross-product $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^{0,1}$ and a typing example. It is analogous to the example from Figure 8.20.

A.3 Relaxation of Strong Pointer Races

Chapter 8 introduced the notion of strong pointer races. Strong pointer races extend ordinary pointer races with unsafe assumptions, Definition 8.3. Recall that an assumption $\text{assume } p = q$ is deemed unsafe if p or q is invalid. In practice, this restriction leads to false alarms during verification. The reason for this is that data structures may compare invalid pointers with constants like `NULL`; for an example consider Line 439 in Vechev&Yahav’s 2CAS set. Since the addresses held by such constants, unlike ordinary pointers, do not undergo allocation-free-reallocation cycles, the problematic comparisons are, in fact, not ABA prone. That is, the strong pointer race is spurious and verification need not fail.

In the following, we present a lift of the reduction from Chapter 8 such that assumptions involving invalid pointers and constants do not hinder verification. To that end, we introduce a set of constant pointer variables $CVar \subseteq PVar \cap \text{shared}$ that may be compared to any other pointer, valid or not.

Definition A.6 (Relaxed Unsafe Assumption). A computation τ is a relaxed unsafe assumption if there is some $\text{assume } p = q \in \text{next-com}(\tau)$ such that (i) $p \notin \text{valid}_\tau \wedge q \notin CExp$ or (ii) $p \notin CExp \wedge q \notin \text{valid}_\tau$. \square

Then, a moderate pointer race is either an ordinary pointer race or a relaxed unsafe assumption. Note that strong pointer race freedom implies moderate pointer race freedom.

Definition A.7 (Moderate Pointer Race). A computation $\tau.act$ is a moderate pointer race if (i) act is an ordinary pointer race, or (ii) $\tau.act$ is a relaxed unsafe assumption. \square

For our results to apply, we require that constants point to addresses that have never been freed before. This guarantees that constants are never equal to invalid pointers. That is, the assumptions the above relaxation allows for are not enabled. The following definition formulates the requirements.

Definition A.8 (Constant Violation). A computation τ is a constant violation if there is a constant $C \in CVar$ such that $m_\tau(C) \in \text{frees}_\tau \cup \text{retired}_\tau$. \square

For simplicity, we assume that $\mathcal{O}[\![P]\!]_{Adr}^{Adr}$ is free from constant violations. The requirement can be established easily with a syntactic check. For example, by ensuring that (i) constants are not updated after the first retirement, and (ii) no address held by a constant is retired. This aligns with our intuition of constants: they are written once at the very beginning of a computation.

Assumption A.9 (No Constant Violations). There are no constant violations in $\mathcal{O}[[P]]_{Adr}^{Adr}$. \square

Provided that a program is free from moderate pointer races, we obtain the same reduction as for strong pointer races in Chapter 8.

Theorem A.10 (✎ Proof C.58). If \mathcal{O} supports elision and $\mathcal{O}[[P]]_{Adr}^{\emptyset}$ is free from moderate pointer races and double retires, then we have $good(\mathcal{O}[[P]]_{Adr}^{Adr}) \iff good(\llbracket P \rrbracket_{\emptyset}^{\emptyset})$ and $\mathcal{O}[[P]]_{Adr}^{Adr}$ is free from double retires. \square

The type system from Chapter 8 can be adapted to moderate pointer races by adding the following rules which types the newly allowed scenario:

$$\frac{\text{(ASSUME1-CONSTANT)} \quad T'' = (T \wedge T') \setminus \{\mathbb{L}\} \quad \{p, q\} \cap CVar \neq \emptyset}{\{\Gamma, p : T, q : T'\} \text{ assume } p = q \quad \{\Gamma, p : T'', q : T''\}}$$

The extend type rules are sound and check for moderate pointer races.

Theorem A.11 (✎ Proof C.84). If $inv(\llbracket P \rrbracket_{\emptyset}^{\emptyset})$ and $\vdash P$, then $inv(\mathcal{O}[[P]]_{Adr}^{\emptyset})$ and $\mathcal{O}[[P]]_{Adr}^{\emptyset}$ is free from moderate pointer races and double retires. \square

For the type check, observe that constants are always active. Hence, corresponding annotations can be added without the need to check them.

Proposition A.12 (✎ Proof C.85). If $\tau \in \mathcal{O}[[P]]_{Adr}^{Adr}$, then $m_{\tau}(CVar) \subseteq active(\tau)$. \square

Meta Theory

We present the full meta theory developed for this thesis. The proofs can be found in Appendix C. Hereafter, we use the following abbreviations for computations τ :

$\tau \text{ DRF} : \iff \tau$ is free from double retires, Definition 5.5,

$\tau \text{ UAF} : \iff \tau$ is free from unsafe accesses, Definition 7.10,

$\tau \text{ PRF} : \iff \tau$ is free from (ordinary) pointer races, Definition 7.13,

$\tau \text{ SPRF} : \iff \tau$ is free from strong pointer races, Definition 8.4,

$\tau \text{ MPRF} : \iff \tau$ is free from moderate pointer races, Definition A.7,

and similarly for sets of computations.

Repeated Assumption 5.2. SMR automata \mathcal{O} satisfy: if $s_1 \xrightarrow{h} s_2$ and $s_1 \xrightarrow{h} s_3$, then $s_2 = s_3$. □

Repeated Assumption 5.6. SMR automata \mathcal{O} are of the form $\mathcal{O} = \mathcal{O}_{Base} \times \mathcal{O}_{SMR}$. □

B.1 Formal Definitions

We make formal the definitions that are missing in the main part. For convenience, we repeat key definitions that play a central role in our development and are frequently invoked in Appendices B and C. We write $com(act) = com$ to access the command com executed by action $act = \langle t, com, up \rangle$.

B.1.1 Computations

Definition B.1. We define $var \cap Adr := \emptyset$ for $var \in PVar \cup DVar$ and $a.sel \cap Adr := \{a\}$. □

Definition B.2. The addresses in use in m are: $adr(m) := (range(m) \cup dom(m)) \cap Adr$. □

Definition B.3. Memories m valuate sets M by $m(M) := \{m(exp) \mid exp \in M\} \setminus \{seg\}$. □

Definition B.4. The set of thread-local variables of t is $local_t = \{p_t \mid p \notin shared\}$ of non-shared variables indexed by t . □

Definition B.5. The fresh addresses after a computation τ , denoted $fresh_\tau$, are defined by:

$$\begin{aligned} fresh_\epsilon &:= Adr \\ fresh_{\tau.act} &:= fresh_\tau \setminus \{a\} && \text{if } com(act) \equiv free(a) \\ fresh_{\tau.act} &:= fresh_\tau \setminus \{a\} && \text{if } com(act) \equiv p := malloc \wedge m_{\tau.act}(p) = a \\ fresh_{\tau.act} &:= fresh_\tau && \text{otherwise} \end{aligned}$$

□

Definition B.6. The freed addresses after a computation τ , denoted $freed_\tau$, are defined by:

$$\begin{aligned} freed_\epsilon &:= \emptyset \\ freed_{\tau.act} &:= freed_\tau \cup \{a\} && \text{if } com(act) \equiv free(a) \\ freed_{\tau.act} &:= freed_\tau \setminus \{a\} && \text{if } com(act) \equiv p := malloc \wedge m_{\tau.act}(p) = a \\ freed_{\tau.act} &:= freed_\tau && \text{otherwise} \end{aligned}$$

□

Definition B.7. The addresses freed in τ are:

$$frees_\tau := \{a \mid \exists \tau_1, \tau_2. \tau = \tau_1.\tau_2 \wedge a \in freed_{\tau_1}\}.$$

□

Definition B.8. The retired addresses after a computation τ , denoted $retired_\tau$, are defined by:

$$\begin{aligned} retired_\epsilon &:= \emptyset \\ retired_{\tau.act} &:= retired_\tau \cup \{a\} && \text{if } com(act) \equiv in:retire(p) \wedge m_\tau(p) = a \\ retired_{\tau.act} &:= retired_\tau \setminus \{a\} && \text{if } com(act) \equiv free(a) \\ retired_{\tau.act} &:= retired_\tau && \text{otherwise} \end{aligned}$$

□

Definition B.9. The active addresses after τ are:

$$active(\tau) := Adr \setminus (freed_\tau \cup retired_\tau).$$

□

Definition B.10. The set of control-flow-enabled commands after τ are:

$$next-com(\tau) := \{com \mid \exists t, pc, stmt. pc \in ctrl(\tau) \wedge pc(t) \xrightarrow{com} stmt\}.$$

□

Definition B.11. We define:

$$VExp(\tau) := PVar \cup \{a.next \mid a \in m_\tau(valid_\tau)\}.$$

□

Definition B.12. Indexing a (pointer/angel/data) variable var by a thread t yields a new variable var_t . Indexing a function $func$ by a thread t yields a new function $func_t$, which we assume to exist in the used SMR implementation. Indexing all non-shared variables $var \notin shared$ and all functions $func$ by a thread t in P gives a new program $P^{[t]}$. □

Definition B.13. The initial program counter pc_{init} is $pc_{init} = \lambda t. P^{[t]} \circ skip$ for the standard semantics and $pc_{init} = \lambda t. P^{[t]}$ for the SMR semantics. □

B.1.2 SMR Automata

Definition B.14. The fresh addresses after a history h , denoted by $fresh_h$, are defined by:

$$\begin{aligned} fresh_\epsilon &:= \text{Adr} \\ fresh_{h.\text{free}(a)} &:= fresh_h \setminus \{a\} \\ fresh_{h.\text{in:func}(t,v_1,\dots,v_n)} &:= fresh_h \setminus \{v_1, \dots, v_n\} \\ fresh_{h.\text{re:func}(t)} &:= fresh_h \end{aligned}$$

□

Definition B.15. The addresses freed in h are:

$$frees_h := \{a \mid \exists h_1, h_2. h = h_1.\text{free}(a).h_2\}.$$

□

Definition B.16. A set of locations L in \mathcal{O} is closed under interference if:

$$\begin{aligned} \forall l, l', f, t, \bar{r}, g. \quad & (l \in L \wedge l \xrightarrow{f(t, \bar{r}), g} l' \wedge l' \notin L) \implies g \models t = z_t \\ & \wedge (l \in L \wedge l \xrightarrow{\text{free}(\bullet), g} l') \implies l' \in L \end{aligned}$$

where \models denotes entailment among logic formulas.

□

Definition B.17. The locations in \mathcal{O} that grant safe dereferences, denoted $\text{SafeLoc}(\mathcal{O})$, are:

$$\begin{aligned} L_{\text{safe}} &:= \{l \mid \forall t, a, g \exists l' \in L_{\text{acc}}. l \xrightarrow{\text{free}(a), g} l' \wedge \text{SAT}(g \wedge t = z_t \wedge a = z_a)\} \\ \text{SafeLoc}(\mathcal{O}) &:= \bigcup \{L \subseteq L_{\text{safe}} \mid L \text{ closed under interference}\} \end{aligned}$$

where L_{acc} are the accepting locations in \mathcal{O} and $\text{SAT}(\bullet)$ tests logic formulas for satisfiability.

□

Definition B.18. The post locations of L under com wrt. x , denoted by $\text{post}_{x, \text{com}}(L)$, are:

$$\begin{aligned} \text{post}_{x, \text{com}}(L) &:= \{l' \mid \exists l \in L \exists t, g. l \xrightarrow{\text{in:func}(t, \bar{r}), g} l' \wedge \text{SAT}(g \wedge t = z_t) \\ &\quad \wedge (x \in \bar{r} \implies \text{SAT}(g \wedge x = z_a))\} \quad \text{if } \text{com} \equiv \text{in:func}(\bar{r}) \\ \text{post}_{x, \text{com}}(L) &:= \{l' \mid \exists l \in L \exists t, g. l \xrightarrow{\text{re:func}(t), g} l' \wedge \text{SAT}(g \wedge t = z_t)\} \quad \text{if } \text{com} \equiv \text{re:func} \\ \text{post}_{x, \text{com}}(L) &:= L \quad \text{otherwise.} \end{aligned}$$

where $\text{SAT}(\bullet)$ tests logic formulas for satisfiability.

□

Definition B.19. The locations reached by SMR automaton \mathcal{O} after history h wrt. thread t and address a , denoted by $\text{reach}_{t,a}^\mathcal{O}(h)$, are defined by:

$$\text{reach}_{t,a}^\mathcal{O}(h) := \{l \mid \exists \varphi. (l_{\text{init}}, \varphi) \xrightarrow{h} (l, \varphi) \wedge \varphi(z_t) = t \wedge \varphi(z_a) = a\}$$

where l_{init} is the initial location in \mathcal{O} . For seg we define $\text{reach}_{t, \text{seg}}^\mathcal{O}(h) := \text{Loc}(\mathcal{O})$ to be all locations of \mathcal{O} . The definition extends naturally to (sets of) computations and histories.

□

B.1.3 Elision

Definition B.20. An address mapping is a bijection $swap_{adr} : Adr \rightarrow Adr$. For convenience, we extend this function by $swap_{adr}(\perp) = \perp$ and $swap_{adr}(seg) = seg$ as well as $swap_{adr}(d) = d$ for any $d \in Dom$. The address mapping induces an expression mapping $swap_{exp}$ with

$$\begin{aligned} swap_{exp}(p) &= p & swap_{exp}(a.next) &= swap_{adr}(a).next \\ swap_{exp}(u) &= u & swap_{exp}(a.data) &= swap_{adr}(a).data \end{aligned}$$

and a history mapping $swap_{hist}$ with

$$\begin{aligned} swap_{hist}(\epsilon) &= \epsilon \\ swap_{hist}(h.free(a)) &= swap_{hist}(h).free(swap_{adr}(a)) \\ swap_{hist}(h.in:func(t, \bar{v})) &= swap_{hist}(h).func(t, swap_{adr}(\bar{v})) \\ swap_{hist}(h.re:func(t)) &= swap_{hist}(h).re:func(t) \end{aligned}$$

for any SMR function $func$. For $\bar{v} = v_1, \dots, v_k$ we use $swap_{adr}(\bar{v}) = swap_{adr}(v_1), \dots, swap_{adr}(v_k)$.

If $swap_{adr}$ is an address mapping, then we write $swap_{exp}^{-1}$ and $swap_{hist}^{-1}$ for the expression and history mapping induced by the inverse address mapping $swap_{adr}^{-1}$. \square

Definition B.21. The swap of addresses a and b in history h is $h[a/b] := swap_{hist}(h)$ where the history mapping $swap_{hist}$ is induced by the address mapping $swap_{adr}$ such that $swap_{adr}(a) = b$ and $swap_{adr}(b) = a$ and $swap_{adr}(c) = c$ in all other cases. \square

Repeated Definition 7.14 (Elision Support). SMR automaton $\mathcal{O} = \mathcal{O}_{Base} \times \mathcal{O}_{SMR}$ supports elision of memory reuse if for all histories $h, h' \in \mathcal{S}(\mathcal{O}_{Base})$ and for all addresses $a, b, c \in Adr$ the following conditions are met:

- (i) $a \neq c \neq b$ implies $\mathcal{F}_{\mathcal{O}_{SMR}}(h, c) = \mathcal{F}_{\mathcal{O}_{SMR}}(h[a/b], c)$,
- (ii) $\mathcal{F}_{\mathcal{O}}(h, a) \subseteq \mathcal{F}_{\mathcal{O}}(h', a)$ and $b \in fresh_{h'}$ implies $\mathcal{F}_{\mathcal{O}_{SMR}}(h, b) \subseteq \mathcal{F}_{\mathcal{O}_{SMR}}(h', b)$, and
- (iii) $a \neq b$ and $h.free(a) \in \mathcal{S}(\mathcal{O})$ implies $\mathcal{F}_{\mathcal{O}_{SMR}}(h, b) = \mathcal{F}_{\mathcal{O}_{SMR}}(h.free(a), b)$. \square

B.1.4 Races

Repeated Definition 6.5 (Valid Expressions). The valid pointer expressions in $\tau \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$, denoted by $\text{valid}_\tau \subseteq \text{PExp}$, are defined by:

$$\begin{aligned}
\text{valid}_\tau &:= \text{PVar} \\
\text{valid}_{\tau.\langle t, p := q, up \rangle} &:= \text{valid}_\tau \cup \{p\} && \text{if } q \in \text{valid}_\tau \\
\text{valid}_{\tau.\langle t, p := q, up \rangle} &:= \text{valid}_\tau \setminus \{p\} && \text{if } q \notin \text{valid}_\tau \\
\text{valid}_{\tau.\langle t, p.\text{next} := q, up \rangle} &:= \text{valid}_\tau \cup \{a.\text{next}\} && \text{if } m_\tau(p) = a \in \text{Adr} \wedge q \in \text{valid}_\tau \\
\text{valid}_{\tau.\langle t, p.\text{next} := q, up \rangle} &:= \text{valid}_\tau \setminus \{a.\text{next}\} && \text{if } m_\tau(p) = a \in \text{Adr} \wedge q \notin \text{valid}_\tau \\
\text{valid}_{\tau.\langle t, p := q.\text{next}, up \rangle} &:= \text{valid}_\tau \cup \{p\} && \text{if } q \in \text{valid}_\tau \wedge m_\tau(q).\text{next} \in \text{valid}_\tau \\
\text{valid}_{\tau.\langle t, p := q.\text{next}, up \rangle} &:= \text{valid}_\tau \setminus \{p\} && \text{if } q \notin \text{valid}_\tau \vee m_\tau(q).\text{next} \notin \text{valid}_\tau \\
\text{valid}_{\tau.\langle t, \text{free}(a), up \rangle} &:= \text{valid}_\tau \setminus \text{invalid}_a \\
\text{valid}_{\tau.\langle t, p := \text{malloc}, up \rangle} &:= \text{valid}_\tau \cup \{p, a.\text{next}\} && \text{if } [p \mapsto a] \in up \\
\text{valid}_{\tau.\langle t, \text{assume } p=q, up \rangle} &:= \text{valid}_\tau \cup \{p, q\} && \text{if } \{p, q\} \cap \text{valid}_\tau \neq \emptyset \\
\text{valid}_{\tau.\text{act}} &:= \text{valid}_\tau && \text{otherwise}
\end{aligned}$$

with $\text{invalid}_a := \{p \mid m_\tau(p) = a\} \cup \{b.\text{next} \mid m_\tau(b.\text{next}) = a\} \cup \{a.\text{next}\}$. \square

Repeated Definition 7.10 (Unsafe Access). A computation $\tau.\langle t, \text{com}, up \rangle$ performs an unsafe access if com contains $p.\text{data}$ or $p.\text{next}$ with $p \notin \text{valid}_\tau$. \square

Repeated Definition 7.12 (Racy SMR Calls). A computation $\tau.\langle t, \text{in}:\text{func}(\bar{r}), \emptyset \rangle$ performs a racy call if for $\mathcal{H}(\tau) = h$ and $m_\tau(\bar{r}) = \bar{v}$ we have:

$$\begin{aligned}
&\exists a \exists \bar{w}. \quad (\forall i. (v_i = a \vee r_i \in \text{valid}_\tau \vee r_i \in \text{DExp}) \implies v_i = w_i) \\
&\wedge \mathcal{F}_\mathcal{O}(h.\text{in}:\text{func}(t, \bar{w}), a) \notin \mathcal{F}_\mathcal{O}(h.\text{in}:\text{func}(t, \bar{v}), a)
\end{aligned}$$

Repeated Definition 7.13 (Pointer Race). A computation $\tau.\text{act}$ is a pointer race if act performs (i) an unsafe access, or (ii) a racy SMR call. \square

Repeated Definition 8.3 (Unsafe Assumption). A computation τ is prone to an unsafe assumption if there is $\text{assume } p = q \in \text{next-com}(\tau)$ with $p \notin \text{valid}_\tau$ or $q \notin \text{valid}_\tau$. \square

Repeated Definition 8.4 (Strong Pointer Race). A computation $\tau.\text{act}$ is a strong pointer race if act performs (i) an ordinary pointer race, or (ii) an unsafe assumption. \square

Repeated Definition A.6 (Relaxed Unsafe Assumption). Computation τ is prone to a relaxed unsafe assumption if there is $\text{assume } p = q \in \text{next-com}(\tau)$ with (i) $p, q \notin \text{valid}_\tau$, (ii) $p \notin \text{valid}_\tau$ and $q \notin \text{CExp}$, or (iii) $p \notin \text{CExp}$ and $q \notin \text{valid}_\tau$. \square

Repeated Definition A.7 (Moderate Pointer Race). A computation $\tau.\text{act}$ is a moderate pointer race if (i) act is an ordinary pointer race, or (ii) $\tau.\text{act}$ is a relaxed unsafe assumption. \square

B.1.5 Correspondences

Repeated Definition 7.3 (Restrictions). A restriction of memory m to a set $P \subseteq PExp$, denoted by $m|_P$, is a new memory m' such that $dom(m') := P \cup DVar \cup \{ a.data \in DExp \mid a \in m(P) \}$ and $m(e) = m'(e)$ for all $e \in dom(m')$. \square

Repeated Definition 7.4 (Computation Similarity). Two computations τ and σ are similar, denoted by $\tau \sim \sigma$, if $ctrl(\tau) = ctrl(\sigma)$ and $m_\tau|_{valid_\tau} = m_\sigma|_{valid_\sigma}$. \square

Repeated Definition 7.7 (SMR Behavior). The behavior allowed by automaton \mathcal{O} on address a after history h is the set $\mathcal{F}_\mathcal{O}(h, a) := \{ h' \mid h.h' \in S(\mathcal{O}) \wedge frees_{h'} \subseteq a \}$. \square

Repeated Definition 7.8 (SMR Behavior Inclusion). Computation σ includes the SMR behavior of τ , denoted by $\tau \leq \sigma$, if $\mathcal{F}_\mathcal{O}(\tau, a) \subseteq \mathcal{F}_\mathcal{O}(\sigma, a)$ holds for all $a \in adr(m_\tau|_{valid_\tau})$. \square

Repeated Definition 7.17 (Address Alignment). Two computations τ and σ are a -aligned, denoted by $\tau \leq_a \sigma$, if

$$\begin{aligned} & \forall p \in PVar. m_\tau(p) = a \iff m_\sigma(p) = a \\ & \text{and } \forall b \in m_\tau(valid_\tau). m_\tau(b.next) = a \iff m_\sigma(b.next) = a \\ & \text{and } a \in fresh_\tau \cup freed_\tau \iff a \in fresh_\sigma \cup freed_\sigma \\ & \text{and } \mathcal{F}_\mathcal{O}(\tau, a) \subseteq \mathcal{F}_\mathcal{O}(\sigma, a) \\ & \text{and } a \in retired_\tau \iff a \in retired_\sigma. \end{aligned} \quad \square$$

Repeated Definition 7.18 (Harmful ABA). $\mathcal{O}[[P]]_{Adr}^{one}$ is free from harmful ABAs if:

$$\begin{aligned} & \forall \sigma_a.act \in \mathcal{O}[[P]]_{Adr}^{\{a\}} \forall \sigma_b \in \mathcal{O}[[P]]_{Adr}^{\{b\}} \exists \sigma'_b \in \mathcal{O}[[P]]_{Adr}^{\{b\}}. \\ & \sigma_a \sim \sigma_b \wedge act = \langle \bullet, \text{assume } \bullet, \bullet \rangle \implies \sigma_a.act \sim \sigma'_b \wedge \sigma_b \leq_b \sigma'_b \wedge \sigma_a.act \leq \sigma'_b. \end{aligned} \quad \square$$

B.1.6 Types

Definition B.22. The initial type environment for P is Γ_{init} . For $P^{[t]}$ it is $\Gamma_{init}^{[t]}$. Formally:

$$\begin{aligned} \Gamma_{init} &:= \{ x : \emptyset \mid x \in PVar \cup AVar \} \\ \Gamma_{init}^{[t]} &:= \{ p : \emptyset \mid p \in PVar \cap shared \} \cup \{ p_t : \emptyset \mid p \in PVar \setminus shared \} \\ &\quad \cup \{ r_t : \emptyset \mid r \in AVar \} \end{aligned} \quad \square$$

Definition B.23. A type T is valid, denoted by $isValid(T)$, if:

$$isValid(T) : \iff T \cap \{ \mathbb{S}, \mathbb{A}, \mathbb{L} \} \neq \emptyset. \quad \square$$

Definition B.24. A computation τ induces a straight-line program $stmt(\tau, t)$ for thread t by:

$$\begin{aligned} stmt(\epsilon, t) &:= \text{skip} \\ stmt(\tau.act, t) &:= stmt(\tau, t); com && \text{if } act = \langle t, com, up \rangle \\ stmt(\tau.act, t) &:= stmt(\tau, t) && \text{if } act = \langle t', com, up \rangle \wedge t \neq t' \end{aligned} \quad \square$$

Definition B.25. A pointer p has no valid alias in a computation τ , denoted by $noalias_\tau(p)$, if the following holds:
 $seg \neq m_\tau(p) \notin m_\tau(valid_\tau \setminus \{p\})$. \square

Definition B.26. Let $\tau \in \mathcal{O}[[P]]_{Addr}^{Addr}$. Let $inv(\tau)$ have the prenex normal form $\exists r_1 \dots \exists r_n. \phi$, 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

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

We define $m_\tau(r) = repr_\tau(r)$ \square

Definition B.27. An invocation $in:func(\bar{r})$ is approximatively race free in type environment Γ , denoted by $SafeCall(\Gamma, func(\bar{r}))$, if:

$$\begin{aligned} & SafeCall(\Gamma, func(\bar{r})) \\ : \iff & \nexists h \exists t, a, \bar{v}, \bar{w}. \quad (\forall i. r_i \notin DExp \implies reach_{t, v_i}^O(h) \subseteq Loc(\Gamma(r_i))) \\ & \wedge (\forall i. (v_i = a \vee isValid(\Gamma(r_i)) \vee r_i \in DExp) \implies v_i = w_i) \\ & \wedge \mathcal{F}_O(h.in:func(t, \bar{w}), a) \notin \mathcal{F}_O(h.in:func(t, \bar{v}), a). \end{aligned} \quad \square$$

Repeated Definition 8.9 (Meaning of Types). The locations associated with types are:

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

where $Loc(\mathcal{O})$ and $Loc(\mathcal{O}_{SMR})$ refer to all locations of \mathcal{O} and \mathcal{O}_{SMR} respectively. \square

Repeated Definition 8.10 (Type Transformer). The type transformer relation $T, x, com \rightsquigarrow T'$ is defined by:

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

Moreover, we define the following abbreviations:

$$\begin{aligned} \Gamma, com \rightsquigarrow \Gamma' : \iff & \forall x. \Gamma(x), x, com \rightsquigarrow \Gamma'(x) \\ \text{and} \quad \Gamma \rightsquigarrow \Gamma' : \iff & \Gamma, skip \rightsquigarrow \Gamma'. \end{aligned} \quad \square$$

B.2 Compositionality

We present the meta theory for the compositionality result from Chapter 5.

Lemma B.28 (\hookrightarrow Proof C.1). The simulation relation $\leq_{\mathcal{O}_{EBR}}$ for SMR automaton \mathcal{O}_{EBR} is:

$$L_7 \leq_{\mathcal{O}_{EBR}} L_6 \leq_{\mathcal{O}_{EBR}} L_5 \leq_{\mathcal{O}_{EBR}} L_4 \quad \square$$

Lemma B.29 (⚡ Proof C.2). The simulation relation $\leq_{\mathcal{O}_{HP}^k}$ for SMR automaton \mathcal{O}_{HP}^k is:

$$L_{12} \leq_{\mathcal{O}_{HP}^k} L_{11} \leq_{\mathcal{O}_{HP}^k} L_{10} \leq_{\mathcal{O}_{HP}^k} L_9 \leq_{\mathcal{O}_{HP}^k} L_8$$

□

Lemma B.30 (⚡ Proof C.3). The simulation relation $\leq_{\mathcal{O}_{HP}^{0,1}}$ for SMR automaton $\mathcal{O}_{HP}^{0,1}$ is:

$$\begin{aligned} L_{15} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15} & L_{16} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{16} & L_{17} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{16}, L_{17} & L_{18} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{16}, L_{17}, L_{18} \\ L_{19} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{16}, L_{17}, L_{21}, L_{19}, L_{24} & L_{20} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{16}, L_{17}, L_{18}, L_{21}, L_{19}, L_{20}, L_{24} \\ L_{21} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{16}, L_{21}, L_{24} & L_{22} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{16}, L_{17}, L_{21}, L_{19}, L_{22}, L_{24}, L_{25}, L_{27} \\ L_{23} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{16}, L_{17}, L_{18}, L_{21}, L_{19}, L_{20}, L_{22}, L_{23}, L_{24}, L_{25}, L_{26}, L_{27}, L_{28}, L_{29} & L_{24} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{24} \\ L_{25} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{24}, L_{25} & L_{26} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{24}, L_{25}, L_{26} & L_{27} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{16}, L_{21}, L_{24}, L_{25}, L_{27} \\ L_{28} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{16}, L_{21}, L_{24}, L_{25}, L_{26}, L_{27}, L_{28} & L_{29} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{16}, L_{17}, L_{21}, L_{19}, L_{22}, L_{24}, L_{25}, L_{26}, L_{27}, L_{28}, L_{29} \\ L_{30} &\leq_{\mathcal{O}_{HP}^{0,1}} L_{15}, L_{16}, L_{17}, L_{18}, L_{21}, L_{19}, L_{20}, L_{22}, L_{23}, L_{24}, L_{25}, L_{26}, L_{27}, L_{28}, L_{29}, L_{30} \end{aligned}$$

□

Repeated Proposition 5.3 (⚡ Proof C.4). If $l \leq_{\mathcal{O}} l'$, then $\mathcal{S}((l, \varphi)) \subseteq \mathcal{S}((l', \varphi))$ for all φ . □

Repeated Theorem A.2 (⚡ Proof C.5). Let $\tau \in \llbracket P(R) \rrbracket_{Adr}^{Adr}$ be free from separation violations. Then, there is some $\sigma \in \llbracket MGC(R) \rrbracket_{Adr}^{Adr}$ such that (i) $ctrl^R(\tau) \subseteq ctrl^R(\sigma)$, (ii) $m_\tau^R = m_\sigma^R$, (iii) $m_\tau \downarrow_{IVar} = m_\sigma \downarrow_{IVar}$, (iv) $\mathcal{H}(\tau) = \mathcal{H}(\sigma)$, (v) $fresh_\tau \subseteq fresh_\sigma$, (vi) $freed_\tau \subseteq freed_\sigma$, and (vii) $used(\tau) \subseteq used(\sigma)$. □

Repeated Theorem A.4 (⚡ Proof C.7). Consider $\tau \in \llbracket P(R) \rrbracket_{Adr}^{Adr}$ which is free from separation violations. If $R \models \mathcal{O}$, then there is $\sigma \in \mathcal{O} \llbracket P \rrbracket_{Adr}^{Adr}$ with (i) $ctrl^P(\tau) \subseteq ctrl(\sigma)$, (ii) $m_\tau^P = m_\sigma^P$, (iii) $\mathcal{H}(\tau) = \mathcal{H}(\sigma)$, (iv) $fresh_\tau \subseteq fresh_\sigma$, (v) $freed_\tau \subseteq freed_\sigma$, and (vi) $retired_\tau \subseteq retired_\sigma$. □

Repeated Theorem 5.10 (⚡ Proof C.8). If $R \models \mathcal{O}$ and $good(\mathcal{O} \llbracket P \rrbracket_{Adr}^{Adr})$, then $good(\llbracket P(R) \rrbracket_{Adr}^{Adr})$. □

Repeated Theorem 5.11 (⚡ Proof C.9). If $R \models \mathcal{O}$, then $\llbracket P(R) \rrbracket_{Adr}^{Adr}$ is DRF if $\mathcal{O} \llbracket P \rrbracket_{Adr}^{Adr}$ is. □

B.3 Ownership

We present the meta theory for the ownership result from Chapter 6.

Repeated Theorem 6.7 (⚡ Proof C.10). Consider some $\tau \in \mathcal{O} \llbracket P \rrbracket_{Adr}^{Adr}$ with $m_\tau(p) \in owned_\tau(t)$. Then, we have: $p \in valid_\tau$ implies $p \in local_t$. □

B.4 Reductions

We present the meta theory for the reduction results from Chapters 7 and 8.

B.4.1 Useful Observations

We provide insights useful for the proofs. To simplify the presentation, all symbols that are not explicitly quantified are implicitly universally quantified. Furthermore, we implicitly assume computations $\tau, \tau_1, \tau_2, \tau_3$ to be drawn from $\mathcal{O}[\![P]\!]_{Adr}^{Adr}$ unless specified otherwise.

Lemma B.31 (↻ Proof C.11). If $\tau_1 \sim \tau_2$, then $\tau_2 \sim \tau_1$. □

Lemma B.32 (↻ Proof C.12). If $\tau_1 \sim \tau_2 \sim \tau_3$, then $\tau_1 \sim \tau_3$. □

Lemma B.33 (↻ Proof C.13). If $m_{\tau_1}(\text{valid}_{\tau_1}) \subseteq m_{\tau_2}(\text{valid}_{\tau_2})$ and $\tau_1 \preceq_A \tau_2 \preceq_B \tau_3$ holds, then we have $\tau_1 \preceq_{A \cap B} \tau_3$. □

Lemma B.34 (↻ Proof C.14). If $\text{adr}(m_{\tau_1} \upharpoonright \text{valid}_{\tau_1}) \subseteq \text{adr}(m_{\tau_2} \upharpoonright \text{valid}_{\tau_2})$ and $\tau_1 \prec \tau_2 \prec \tau_3$ holds, then we have $\tau_1 \prec \tau_3$. □

Lemma B.35 (↻ Proof C.15). We have $\text{adr}(m_{\tau} \upharpoonright \text{valid}_{\tau}) = (\text{valid}_{\tau} \cap \text{Adr}) \cup m_{\tau}(\text{valid}_{\tau})$. □

Lemma B.36 (↻ Proof C.16). We have $\text{valid}_{\tau} \subseteq \text{dom}(m_{\tau})$. □

Lemma B.37 (↻ Proof C.17). If $\tau \sim \sigma$, then $\text{valid}_{\tau} = \text{valid}_{\sigma}$. □

Lemma B.38 (↻ Proof C.18). If $\tau \sim \sigma$, then $\text{adr}(m_{\tau} \upharpoonright \text{valid}_{\tau}) = \text{adr}(m_{\sigma} \upharpoonright \text{valid}_{\sigma})$. □

Lemma B.39 (↻ Proof C.19). If $\tau \sim \sigma$, then $\text{next-com}(\tau) = \text{next-com}(\sigma)$. □

Lemma B.40 (↻ Proof C.20). If $\tau.\langle t, \text{com}, \text{up} \rangle \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$, $t \neq \perp$, then $\text{com} \in \text{next-com}(\tau)$. □

Lemma B.41 (↻ Proof C.21). If $\text{frees}_{h_2} \subseteq \{a\}$, then we have:

$$h_3 \in \mathcal{F}_{\mathcal{O}}(h_1.h_2, a) \iff h_2.h_3 \in \mathcal{F}_{\mathcal{O}}(h_1, a) . \quad \square$$

Lemma B.42 (↻ Proof C.22). If $a \in \text{fresh}_{\tau}$, then $a \notin \text{range}(m_{\tau})$. □

Lemma B.43 (↻ Proof C.23). If $a \in \text{fresh}_{\tau}$, then $a \notin m_{\tau}(\text{valid}_{\tau})$ and $a.\text{next} \notin \text{valid}_{\tau}$. □

Lemma B.44 (↻ Proof C.24). We have $\text{fresh}_{\tau} \cap \text{freed}_{\tau} = \emptyset$ and $\text{fresh}_{\tau} \cap \text{retired}_{\tau} = \emptyset$. □

Lemma B.45 (↻ Proof C.25). If $\varphi = \{z_a \mapsto a\}$, then we have $(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau)} (L_3, \varphi)$ iff $a \in \text{retired}_{\tau}$ as well as $(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau)} (L_2, \varphi)$ iff $a \notin \text{retired}_{\tau}$. □

Lemma B.46 (↻ Proof C.26). If $\tau.\langle t, \text{free}(a), \text{up} \rangle \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$, then $a \in \text{retired}_{\tau}$. □

Lemma B.47 (↻ Proof C.27). Let $\tau.act \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$ with $act = \langle t, com, up \rangle$. We have:

$$\begin{array}{ll}
m_{\tau.act}(valid_{\tau.act}) \subseteq m_{\tau}(valid_{\tau}) \cup \{m_{\tau.act}(p)\} & \text{if } com \equiv p := \text{malloc} \\
m_{\tau.act}(valid_{\tau.act}) \subseteq m_{\tau}(valid_{\tau}) \setminus \{a\} & \text{if } com \equiv \text{free}(a) \\
m_{\tau.act}(valid_{\tau.act}) \subseteq m_{\tau}(valid_{\tau}) & \text{if } com \equiv \text{env}(a) \\
m_{\tau.act}(valid_{\tau.act}) \subseteq m_{\tau}(valid_{\tau}) & \text{if } com \equiv exp := exp^! \\
m_{\tau.act}(valid_{\tau.act}) = m_{\tau}(valid_{\tau}) & \text{otherwise} \\
\text{and } adr(m_{\tau.act}|_{valid_{\tau.act}}) \subseteq adr(m_{\tau}|_{valid_{\tau}}) \cup \{m_{\tau.act}(p)\} & \text{if } com \equiv p := \text{malloc} \\
adr(m_{\tau.act}|_{valid_{\tau.act}}) \subseteq adr(m_{\tau}|_{valid_{\tau}}) \setminus \{a\} & \text{if } com \equiv \text{free}(a) \\
adr(m_{\tau.act}|_{valid_{\tau.act}}) \subseteq adr(m_{\tau}|_{valid_{\tau}}) & \text{if } com \equiv \text{env}(a) \\
adr(m_{\tau.act}|_{valid_{\tau.act}}) \subseteq adr(m_{\tau}|_{valid_{\tau}}) & \text{if } com \equiv exp := exp^! \\
adr(m_{\tau.act}|_{valid_{\tau.act}}) = adr(m_{\tau}|_{valid_{\tau}}) & \text{otherwise .}
\end{array}$$

The next set of lemmas provides insights about computations free from unsafe accesses.

Lemma B.48 (↻ Proof C.28). If τ UAF and $a \in freed_{\tau}$, so $a \notin m_{\tau}(valid_{\tau})$ and $a.next \notin valid_{\tau}$. □

Lemma B.49 (↻ Proof C.29). If τ UAF, $pexp \in VExp(\tau)$, and $m_{\tau}(pexp) = \text{seg}$, so $pexp \in valid_{\tau}$. □

Lemma B.50 (↻ Proof C.30). If τ UAF, then $VExp(\tau) \subseteq dom(m_{\tau})$. □

Lemma B.51 (↻ Proof C.31). If $\tau \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$ UAF, then $m_{\tau}(VExp(\tau) \setminus valid_{\tau}) \subseteq frees_{\tau}$. □

Lemma B.52 (↻ Proof C.32). If $\tau \in \mathcal{O}[\![P]\!]_{Adr}^{\emptyset}$ UAF, then $m_{\tau}(VExp(\tau) \setminus valid_{\tau}) \subseteq freed_{\tau}$. □

Lemma B.53 (↻ Proof C.33). If $\tau \in \mathcal{O}[\![P]\!]_{Adr}^A$ UAF, then we have:

$$adr(m_{\tau}|_{valid_{\tau}}) \cap m_{\tau}(VExp(\tau) \setminus valid_{\tau}) \subseteq A .$$
□

Lemma B.54 (↻ Proof C.34). If $\tau \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$ UAF, then $m_{\tau}(CVar) \cap m_{\tau}(PVar \setminus valid_{\tau}) = \emptyset$. □

Consider an SMR automaton \mathcal{O} . By Assumption 5.6, it is of the form $\mathcal{O} = \mathcal{O}_{Base} \times \mathcal{O}_{SMR}$. Recall that elision support for \mathcal{O} , Definition 7.14, provides properties for \mathcal{O}_{SMR} only. We generalize them to full \mathcal{O} .

Lemma B.55 (↻ Proof C.35). Let \mathcal{O} support elision. For all $\tau \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$ and $a, b, c \in Adr$ such that $\mathcal{H}(\tau) = h$ and $a \neq c \neq b$, we have $\mathcal{F}_{\mathcal{O}}(h, c) = \mathcal{F}_{\mathcal{O}}(h[a/b], c)$. □

Lemma B.56 (↻ Proof C.36). Let \mathcal{O} supports elision. For all $\tau \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$ and $a, b \in Adr$ such that $a \neq b$ and $\mathcal{H}(\tau) = h$ and $h.free(a) \in \mathcal{S}(\mathcal{O})$, we have $\mathcal{F}_{\mathcal{O}}(h.free(a), b) = \mathcal{F}_{\mathcal{O}}(h, b)$. □

Lemma B.57 (↻ Proof C.37). Let \mathcal{O} supports elision. For all $\tau, \sigma \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$ and $a, b \in Adr$ such that we have $\mathcal{F}_{\mathcal{O}}(\tau, a) \subseteq \mathcal{F}_{\mathcal{O}}(\sigma, a)$ and $b \notin retired_{\tau}$ and $b \in fresh_{\sigma}$, we also have $\mathcal{F}_{\mathcal{O}}(\tau, b) \subseteq \mathcal{F}_{\mathcal{O}}(\sigma, b)$. □

B.4.2 Elision Technique

We present the elision technique which forms the backbone of our reduction results. Intuitively, the techniques allows us to rename addresses in a computation.

Lemma B.58 (⚡ Proof C.38). If $swap_{adr}$ is an address mapping, so is $swap_{adr}^{-1}$. □

Lemma B.59 (⚡ Proof C.39). If $swap_{hist}(h_1) = swap_{hist}(h_2)$, then $h_1 = h_2$. □

Lemma B.60 (⚡ Proof C.40). We have $swap_{hist}(h) \in swap_{hist}(H) \iff h \in H$. □

Lemma B.61 (⚡ Proof C.41). For every $\otimes \in \{ \setminus, \cup, \cap \}$ we have:

$$swap_{adr}(A_1) \otimes swap_{adr}(A_2) = swap_{adr}(A_1 \otimes A_2)$$

$$swap_{exp}(B_1) \otimes swap_{exp}(B_2) = swap_{exp}(B_1 \otimes B_2)$$

$$swap_{hist}(C_1) \otimes swap_{hist}(C_2) = swap_{hist}(C_1 \otimes C_2)$$
□

Lemma B.62 (⚡ Proof C.42). For all h we have $swap_{hist}^{-1}(swap_{hist}(h)) = h$. □

Lemma B.63 (⚡ Proof C.43). For all h we have $h \in \mathcal{S}(\mathcal{O}) \iff swap_{hist}(h) \in \mathcal{S}(\mathcal{O})$. □

Lemma B.64 (⚡ Proof C.44). We have $swap_{hist}(\mathcal{F}_{\mathcal{O}}(h, a)) = \mathcal{F}_{\mathcal{O}}(swap_{hist}(h), swap_{adr}(a))$ for all addresses $a \in \text{Adr}$. □

Theorem B.65 (⚡ Proof C.45). For all $\tau \in \mathcal{O}[\![P]\!]_{\text{Adr}}^A$ and all address mappings $swap_{adr}$, there is some $\sigma \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{swap_{adr}(A)}$ such that:

$$\begin{array}{lll} m_{\sigma} \circ swap_{exp} = swap_{adr} \circ m_{\tau} & \mathcal{H}(\sigma) = swap_{hist}(\mathcal{H}(\tau)) & freed_{\sigma} = swap_{adr}(freed_{\tau}) \\ valid_{\sigma} = swap_{exp}(valid_{\tau}) & ctrl(\sigma) = ctrl(\tau) & fresh_{\sigma} = swap_{adr}(fresh_{\tau}) \end{array}$$
□

Lemma B.66 (⚡ Proof C.46). Let \mathcal{O} support elision. If $\tau \in \mathcal{O}[\![P]\!]_{\text{Adr}}^A$ and $a \notin \text{adr}(m_{\tau}|_{\text{valid}_{\tau}}) \cup A$, then there is some $\sigma \in \mathcal{O}[\![P]\!]_{\text{Adr}}^A$ such that (i) $\tau \sim \sigma$, (ii) $\tau \leq_A \sigma$, (iii) $\tau \leq \sigma$, (iv) $\text{retired}_{\tau} \subseteq \text{retired}_{\sigma} \cup \{a\}$, (v) $a \in \text{fresh}_{\sigma}$, (vi) if $a \notin \text{fresh}_{\tau}$, then $\mathcal{F}_{\mathcal{O}}(\tau, c) = \mathcal{F}_{\mathcal{O}}(\sigma, c)$ for all $c \in \text{fresh}_{\sigma} \setminus \{a\}$, and (vii) if we have $pexp, qexp \in \text{VExp}(\tau)$, then $m_{\tau}(exp) \neq m_{\tau}(exp')$ implies $m_{\sigma}(exp) \neq m_{\sigma}(exp')$. □

B.4.3 Reduction Results

Towards the reduction results, we establish the following auxiliary lemmas which deal with the less interesting cases of the reduction. Furthermore, we make an observation on indicators for double retires.

Lemma B.67 (⚡ Proof C.47). Consider $\tau.act \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{A_{\text{dr}}}$ and $\sigma \in \mathcal{O}[\![P]\!]_{\text{Adr}}^A$ with $\tau \sim \sigma$, $\tau \leq_A \sigma$, and $\tau \leq \sigma$. Let $act = \langle t, com, up \rangle$. If one of the following cases applies

- (i) $com \equiv x := y$ with $x, y \in \text{Var}$,
- (ii) $com \equiv x := q.sel$ with $x \in \text{Var}$ and $q \in \text{valid}_{\tau}$, or
- (iii) $com \equiv p.sel := y$ with $y \in \text{Var}$ and $p \in \text{valid}_{\tau}$,

then there is $act' = \langle t, com, up' \rangle$ with $\sigma.act' \in \mathcal{O}[\![P]\!]_{\text{Adr}}^A$, $\tau.act \sim \sigma.act'$, $\tau.act \leq_A \sigma.act'$, and $\tau.act \leq \sigma.act'$. □

Lemma B.68 (✎ Proof C.48). Consider $\tau.act \in \mathcal{O}[[P]]_{Adr}^{Adr}$ and $\sigma \in \mathcal{O}[[P]]_{Adr}^A$ with $\tau \sim \sigma$, $\tau \preceq_A \sigma$, and $\tau \prec_A \sigma$. Let $act = \langle t, com, up \rangle$. If $\sigma.act \in \mathcal{O}[[P]]_{Adr}^A$ and one of the following cases applies:

- (i) $com \equiv \text{in:func}(r_1, \dots, r_n)$ with $m_\tau(r_i) = m_\sigma(r_i)$ for all $1 \leq i \leq n$,
- (ii) $com \equiv \text{re:func}$,
- (iii) $com \equiv \text{assume } \bullet$,
- (iv) $com \equiv \text{free}(a)$ with $\mathcal{F}_\mathcal{O}(h.\text{free}(a), b) = \mathcal{F}_\mathcal{O}(h, b)$ for all $b \neq a$ and $h \in \{\mathcal{H}(\tau), \mathcal{H}(\sigma)\}$,
- (v) $com \equiv p := \text{malloc}$ with $m_{\tau.act}(p) = a$ and $a \notin A \implies \mathcal{F}_\mathcal{O}(\tau, a) \subseteq \mathcal{F}_\mathcal{O}(\sigma, a)$, or
- (vi) $com \equiv \text{env}(a)$.

then $\tau.act \sim \sigma.act$, $\tau.act \preceq_A \sigma.act$, and $\tau.act \prec \sigma.act$. \square

Lemma B.69 (✎ Proof C.49). Let $\tau = \tau_1.\langle \perp, \text{env}(a), up \rangle.\tau_2 \in \mathcal{O}[[P]]_X^Y$ UAF and $\sigma = \tau_1.\tau_2$. Then: (i) $\sigma \in \mathcal{O}[[P]]_X^Y$, (ii) $\text{ctrl}(\tau) = \text{ctrl}(\sigma)$, (iii) $\mathcal{H}(\tau) = \mathcal{H}(\sigma)$, (iv) $\text{fresh}_\tau = \text{fresh}_\sigma$, (v) $\text{freed}_\tau = \text{freed}_\sigma$, (vi) $\text{retired}_\tau = \text{retired}_\sigma$, and (vii) $(\text{exp} \cap \text{Adr}) \cap (\text{fresh}_\tau \cup \text{freed}_\tau) \neq \emptyset$ if $m_\tau(\text{exp}) \neq m_\sigma(\text{exp})$. \square

Lemma B.70 (✎ Proof C.50). Let \mathcal{O} supports elision. If $\tau \in \mathcal{O}[[P]]_X^Y$ UAF and $a \in \text{retired}_\tau \cap \text{freed}_\tau$, then $\mathcal{O}[[P]]_X^Y$ contains a double retire. \square

We turn to the reduction result from Chapter 7 which relies on pointer race freedom to verify $\mathcal{O}[[P]]_{Adr}^{Adr}$ under the much smaller semantics $\mathcal{O}[[P]]_{Adr}^\emptyset$.

Repeated Theorem 7.20 (✎ Proof C.51). Let \mathcal{O} support elision. Let $\mathcal{O}[[P]]_{Adr}^{one}$ be PRF, DRF, and free from harmful ABAs. Then, for all $\tau \in \mathcal{O}[[P]]_{Adr}^{Adr}$ and all $a \in \text{Adr}$ there is $\sigma \in \mathcal{O}[[P]]_{Adr}^{\{a\}}$ such that $\tau \sim \sigma$, $\tau \prec \sigma$, and $\tau \preceq_a \sigma$. \square

Repeated Theorem 7.21 (✎ Proof C.52). Let \mathcal{O} support elision. Let $\mathcal{O}[[P]]_{Adr}^{one}$ be PRF, DRF, and free from harmful ABAs. Then, $\text{good}(\mathcal{O}[[P]]_{Adr}^{Adr}) \iff \text{good}(\mathcal{O}[[P]]_{Adr}^{one})$. \square

Repeated Theorem 7.22 (✎ Proof C.53). Let \mathcal{O} support elision. Let $\mathcal{O}[[P]]_{Adr}^{one}$ be PRF, DRF, and free from harmful ABAs. Then, $\mathcal{O}[[P]]_{Adr}^{Adr}$ is DRF. \square

Repeated Proposition 7.15 (✎ Proof C.54). The SMR automata $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$ and $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^{0,1}$ and $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$ support elision. \square

Repeated Proposition 7.23 (✎ Proof C.55). If a call is racy wrt. $\mathcal{O}_{Base} \times \mathcal{O}_{EBR}$ or $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^{0,1}$ or $\mathcal{O}_{Base} \times \mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$, then it is a call to retire with an invalid pointer as its argument. \square

Next, we turn to the reduction result from Chapter 8 which relies on strong pointer race freedom to verify $\mathcal{O}[[P]]_{Adr}^{Adr}$ under $[[P]]_\emptyset^\emptyset$. Technically, we establish the reduction for the generalization from Appendix A.3 which relies on moderate pointer race freedom.

Theorem B.71 (✎ Proof C.56). Let \mathcal{O} support elision and let $\mathcal{O}[[P]]_{Adr}^\emptyset$ be MPRF and DRF. Then: for all $\tau \in \mathcal{O}[[P]]_{Adr}^{Adr}$ there is $\sigma \in \mathcal{O}[[P]]_{Adr}^\emptyset$ with: (i) $\tau \sim \sigma$, (ii) $\tau \prec \sigma$, (iii) $\mathcal{F}_\mathcal{O}(\tau, a) \subseteq \mathcal{F}_\mathcal{O}(\sigma, a)$ for all $a \in \text{fresh}_\sigma$, (iv) $\text{retired}_\tau \subseteq \text{retired}_\sigma$, and (v) $m_\tau(\text{pexp}) \neq m_\tau(\text{qexp}) \implies m_\sigma(\text{pexp}) \neq m_\sigma(\text{qexp})$ for all $\text{pexp}, \text{qexp} \in \text{VExp}(\tau)$. Moreover, we have (vi) τ UAF, (vii) $\text{freed}_\tau \cap \text{adr}(m_\tau|_{\text{valid}_\tau}) = \emptyset$, and (viii) $\text{freed}_\tau \cap \text{retired}_\tau = \emptyset$. \square

Theorem B.72 (⚡ Proof C.57). For all $\tau \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\emptyset}$ UAF there is $\sigma \in \llbracket P \rrbracket_{\emptyset}^{\emptyset}$ such that we have: (i) $\text{ctrl}(\tau) = \text{ctrl}(\sigma)$, (ii) $\text{fresh}_{\tau} = \text{fresh}_{\sigma}$, (iii) $\text{freed}_{\tau} = \text{retired}_{\sigma}$, (iv) $\text{retired}_{\tau} = \text{retired}_{\sigma}$, (v) $\text{inv}(\sigma)$ implies $\text{inv}(\tau)$, (vi) $m_{\tau}(\text{exp}) \neq m_{\sigma}(\text{exp})$ implies $(\text{exp} \cap \text{Adr}) \cap (\text{fresh}_{\tau} \cup \text{freed}_{\tau}) \neq \emptyset$. \square

Repeated Theorem A.10 (⚡ Proof C.58). If \mathcal{O} supports elision and $\mathcal{O}[\![P]\!]_{\text{Adr}}^{\emptyset}$ is MPRF and DRF, then we have $\text{good}(\mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}) \iff \text{good}(\llbracket P \rrbracket_{\emptyset}^{\emptyset})$ and $\mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$ is DRF. \square

Repeated Theorem 8.5 (⚡ Proof C.59). Let \mathcal{O} support elision and let $\mathcal{O}[\![P]\!]_{\text{Adr}}^{\emptyset}$ be SPRF and DRF. For all $\tau \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$ there is $\sigma \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\emptyset}$ with $\tau \sim \sigma$, $\tau \leq \sigma$, and $\text{retired}_{\tau} \subseteq \text{retired}_{\sigma}$. \square

Repeated Theorem 8.6 (⚡ Proof C.60). If $\tau \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\emptyset}$ is SPRF, then there is $\sigma \in \llbracket P \rrbracket_{\emptyset}^{\emptyset}$ such that $\text{ctrl}(\tau) = \text{ctrl}(\sigma)$, $m_{\tau}|_{\text{valid}_{\tau}} = m_{\sigma}|_{\text{valid}_{\sigma}}$, and $\text{inv}(\sigma) \implies \text{inv}(\tau)$. \square

Repeated Theorem 8.7 (⚡ Proof C.61). If \mathcal{O} supports elision and $\mathcal{O}[\![P]\!]_{\text{Adr}}^{\emptyset}$ is SPRF and DRF, then we have $\text{good}(\mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}) \iff \text{good}(\llbracket P \rrbracket_{\emptyset}^{\emptyset})$ and $\mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$ is DRF. \square

B.5 Type System

We present the meta theory for the type system from Chapter 8.

Repeated Assumption 8.8. SMR automata have two variables z_t resp. z_a tracking a thread resp. an address. \square

Repeated Assumption 8.11. Programs adhere to the following restricted syntax:

$$\begin{aligned} \text{stmt} ::= & \text{stmt}; \text{stmt} \mid \text{stmt} \oplus \text{stmt} \mid \text{stmt}^* \mid \text{beginAtomic}; \text{stmt}; \text{endAtomic} \\ & \mid \text{beginAtomic}; \text{com}; \text{endAtomic} . \end{aligned}$$

Repeated Assumption A.9. There are no constant violations in $\mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$. \square

Lemma B.73 (⚡ Proof C.62). If $\Gamma_1 \rightsquigarrow \Gamma_2$ and $\Gamma_2 \rightsquigarrow \Gamma_3$, then $\Gamma_1 \rightsquigarrow \Gamma_3$. \square

Lemma B.74 (⚡ Proof C.63). If $\vdash \{ \Gamma_1 \} \text{stmt} \{ \Gamma_2 \}$ and $\text{stmt} \xrightarrow{\text{com}} \text{stmt}'$, then there is an intermediate environment Γ such that $\vdash \{ \Gamma_1 \} \text{com} \{ \Gamma \}$ and $\vdash \{ \Gamma \} \text{stmt}' \{ \Gamma_2 \}$. \square

Lemma B.75 (⚡ Proof C.64). Let $\vdash \{ \Gamma_{\text{init}} \} P \{ \Gamma \}$. Consider $(pc_{\text{init}}, \epsilon) \rightarrow^* (pc, \tau)$ and some thread t . Then there is Γ_1, Γ_2 with $\vdash \{ \Gamma_{\text{init}}^{[t]} \} \text{stmt}(\tau, t) \{ \Gamma_1 \}$ and $\vdash \{ \Gamma_1 \} pc(t) \{ \Gamma_2 \}$. \square

Lemma B.76 (⚡ Proof C.65). Consider some $\tau.act \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\emptyset}$ and $t \neq \text{thrd}(\text{act}) \neq \perp$. Then, we have either $\text{stmt}(\tau, t) = \text{skip}$ or $\text{stmt}(\tau, t) = \text{stmt}; \text{endAtomic}$. \square

Lemma B.77 (⚡ Proof C.66). Let $\tau.act \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\emptyset}$ and $t \neq \text{thrd}(\text{act}) \neq \perp$ and $x \in \text{PVar} \cup \text{AVar}$. Assume $\vdash \{ \Gamma_{\text{init}}^{[t]} \} \text{stmt}(\tau, t) \{ \Gamma \}$. Then, $\mathbb{A} \notin \Gamma(x)$ and $x \notin \text{local}_t \implies \Gamma(x) \cap \{ \mathbb{L}, \mathbb{S} \} = \emptyset$. \square

Lemma B.78 (⚡ Proof C.67). Let $\tau.act \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\emptyset}$ and $t \neq \text{thrd}(\text{act}) \neq \perp$ and $p \in \text{PVar} \cap \text{local}_t$. Then, $p \in \text{valid}_{\tau}$ implies $p \in \text{valid}_{\tau.act}$. Moreover, $\text{noalias}_{\tau}(p)$ implies $\text{noalias}_{\tau.act}(p)$. \square

Lemma B.79 (⚡ Proof C.68). Consider $\tau.act \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\emptyset}$ UAF such that $\text{act} = \langle t, @\text{inv } p = q, \text{up} \rangle$ and $\text{inv}(\tau.act)$. Then, $\{ p, q \} \cap \text{valid}_{\tau} \neq \emptyset$ implies $\{ p, q \} \subseteq \text{valid}_{\tau}$. \square

Lemma B.80 (✎ Proof C.69). Let $\tau.act \in \mathcal{O}[\![P]\!]_{Adr}^\emptyset$ UAF such that $act = \langle t, @inv \text{ active}(p), up \rangle$ and $inv(\tau.act)$. Then, $p \in valid_{\tau.act}$ and $reach_{t,a}^\mathcal{O}(\tau.act) \subseteq Loc(\mathbb{A})$ for $a = m_{\tau.act}(p)$. \square

Lemma B.81 (✎ Proof C.70). Consider $\tau.act \in \mathcal{O}[\![P]\!]_{Adr}^\emptyset$ such that $act = \langle t, @inv \text{ active}(r), up \rangle$ and $inv(\tau.act)$. Then, we have for all $a \in repr_{\tau.act}(r)$:

$$repr_{\tau.act}(r) \cap freed_{\tau.act} = \emptyset \quad \text{and} \quad reach_{t,a}^\mathcal{O}(\tau.act) \subseteq Loc(\mathbb{A}). \quad \square$$

Lemma B.82 (✎ Proof C.71). Consider $\tau \in \mathcal{O}[\![P]\!]_{Adr}^\emptyset$. Let Γ, Γ' such that $\Gamma \rightsquigarrow \Gamma'$. Let t be some thread. Let $p \in PVar$ and $a = m_\tau(p)$. Let $r \in AVar$ and $b \in repr_\tau(r)$. Then:

$$\begin{aligned} isValid(\Gamma(p)) \implies p \in valid_\tau & \quad \text{implies} \quad isValid(\Gamma'(p)) \implies p \in valid_\tau \\ isValid(\Gamma(r)) \implies b \notin freed_\tau & \quad \text{implies} \quad isValid(\Gamma'(r)) \implies b \notin freed_\tau \\ \mathbb{L} \in \Gamma(p) \implies noalias_\tau(p) & \quad \text{implies} \quad \mathbb{L} \in \Gamma'(p) \implies noalias_\tau(p) \\ reach_{t,a}^\mathcal{O}(\tau) \subseteq Loc(\Gamma(p)) & \quad \text{implies} \quad reach_{t,a}^\mathcal{O}(\tau) \subseteq Loc(\Gamma'(p)) \\ reach_{t,b}^\mathcal{O}(\tau) \subseteq Loc(\Gamma(r)) & \quad \text{implies} \quad reach_{t,b}^\mathcal{O}(\tau) \subseteq Loc(\Gamma'(r)) \end{aligned} \quad \square$$

Theorem B.83 (✎ Proof C.72). Consider thread t , environment Γ , and $\tau \in \mathcal{O}[\![P]\!]_{Adr}^\emptyset$ UAF such that $inv(\tau)$ and $\vdash \{ \Gamma_{init}^{[t]} \} stmt(\tau, t) \{ \Gamma \}$. Then, we have $reach_{t,m_\tau(p)}^\mathcal{O}(\tau) \subseteq Loc(\Gamma(p))$ and $isValid(\Gamma(p)) \implies p \in valid_\tau$, for all $p \in PVar$. \square

Corollary B.84 (✎ Proof C.73). Consider some thread t , type environment Γ , and $\tau \in \mathcal{O}[\![P]\!]_{Adr}^\emptyset$ such that $inv(\tau)$ and $\vdash \{ \Gamma_{init}^{[t]} \} stmt(\tau, t) \{ \Gamma \}$. Then, τ is PRF. \square

Theorem B.85 (✎ Proof C.74). If $\vdash P$ and $inv(\llbracket P \rrbracket_\emptyset^\emptyset)$, then $\mathcal{O}[\![P]\!]_{Adr}^\emptyset$ is MPRF. \square

Theorem B.86 (✎ Proof C.75). If $\vdash P$ and $inv(\llbracket P \rrbracket_\emptyset^\emptyset)$, then $\mathcal{O}[\![P]\!]_{Adr}^\emptyset$ is DRF. \square

Theorem B.87 (✎ Proof C.76). If $\vdash P$ and $inv(\llbracket P \rrbracket_\emptyset^\emptyset)$, then $inv(\mathcal{O}[\![P]\!]_{Adr}^\emptyset)$. \square

Repeated Theorem 8.14 (✎ Proof C.77). For all threads t and all $\tau \in \mathcal{O}[\![P]\!]_{Adr}^\emptyset$ with $inv(\tau)$ we have the following: $\vdash \{ \Gamma_{init} \} stmt(\tau, t) \{ \Gamma \}$ implies $\models \{ \Gamma_{init} \} stmt(\tau, t) \{ \Gamma \}$. \square

Repeated Theorem 8.15 (✎ Proof C.78). If $\vdash P$ and $inv(\mathcal{O}[\![P]\!]_{Adr}^\emptyset)$, then $\mathcal{O}[\![P]\!]_{Adr}^\emptyset$ is SPRF. \square

Repeated Theorem 8.16 (✎ Proof C.79). If $\vdash P$ and $inv(\mathcal{O}[\![P]\!]_{Adr}^\emptyset)$, then $\mathcal{O}[\![P]\!]_{Adr}^{Adr}$ is DRF. \square

Repeated Theorem 8.21 (✎ Proof C.80). If $\vdash P$ and $inv(\llbracket P \rrbracket_\emptyset^\emptyset)$, then $\mathcal{O}[\![P]\!]_{Adr}^\emptyset$ SPRF and we have $inv(\mathcal{O}[\![P]\!]_{Adr}^\emptyset)$. \square

Repeated Theorem 8.22 (✎ Proof C.81). We have $inv(\llbracket P \rrbracket_\emptyset^\emptyset)$ iff $safe(\llbracket inst(P) \rrbracket_\emptyset^\emptyset)$. The instrumentation is linear in size. \square

Repeated Theorem 8.24 (✎ Proof C.82). Type inference $\vdash P$ runs in time $\mathcal{O}(|P|^2)$. \square

Repeated Theorem 8.26 (✎ Proof C.83). For $\Phi(\Gamma_{init}, P, X)$ we have $Isol(X) = \bigcap \{ \vdash \{ \Gamma_{init} \} P \{ \Gamma \} \mid \Gamma \}$. Hence $Isol(X) \neq \top$ if and only if $\vdash P$. \square

Repeated Theorem A.11 (✎ Proof C.84). If $\vdash P$ and $inv(\llbracket P \rrbracket_\emptyset^\emptyset)$, then $\mathcal{O}[\![P]\!]_{Adr}^\emptyset$ is MPRF and DRF and we have $inv(\mathcal{O}[\![P]\!]_{Adr}^\emptyset)$. \square

Repeated Proposition A.12 (✎ Proof C.85). If $\tau \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$, then $m_\tau(CVar) \subseteq active(\tau)$. \square

Proof of Meta Theory

We give the proofs for the meta theory from Appendix B.

C.1 Compositionality

Proof C.1 (Lemma B.28). By definition of \leq_\bullet and \mathcal{O}_{EBR} . ■

Proof C.2 (Lemma B.29). By definition of \leq_\bullet and \mathcal{O}_{HP}^k . ■

Proof C.3 (Lemma B.30). By definition of \leq_\bullet and $\mathcal{O}_{HP}^{0,1}$. ■

Proof C.4 (Proposition 5.3). Consider some $h \in \mathcal{S}((l_1, \varphi))$. We show that $h \in \mathcal{S}((l_2, \varphi))$ holds. To that end, we proceed by induction over the length of h . In the base case, we have $h = \epsilon$. Then, location l_1 is not accepting by definition. By the simulation relation, l_2 is not accepting as well. Hence, $h \in \mathcal{S}((l_2, \varphi))$ follows as required. For the induction step, consider $f(\bar{v}).h \in \mathcal{S}((l_1, \varphi))$. By Assumption 5.2, there are steps $(l_1, \varphi) \xrightarrow{f(\bar{v})} (l'_1, \varphi)$ and $(l_2, \varphi) \xrightarrow{f(\bar{v})} (l'_2, \varphi)$. The former step is due to a transition $l_1 \xrightarrow{f(\bar{r}), g} l'_1$ such that $\varphi(g[\bar{r} \mapsto \bar{v}])$ evaluates to true. Similarly, the latter step is due to a transition $l_2 \xrightarrow{f(\bar{r}), g} l'_2$ such that $\varphi(g'[\bar{r} \mapsto \bar{v}])$ evaluates to true. This means φ is a model for g and g' . That is, $g \wedge g'$ is satisfiable. Then, $l_1 \leq_{\mathcal{O}} l_2$ yields $l'_1 \leq_{\mathcal{O}} l'_2$. Note that we have $h \in \mathcal{S}((l'_1, \varphi))$ by definition. By induction, we get $h \in \mathcal{S}((l'_2, \varphi))$. Because SMR automata are deterministic by Assumption 5.2, we conclude $f(\bar{v}).h \in \mathcal{S}((l_2, \varphi))$ as required. ■

Proof C.5 (Theorem A.2). We proceed by induction over the structure of τ . In the base case, we have the empty computation $\tau = \epsilon$. Then, the claim follows by definition for $\sigma = \epsilon$. For the induction step, consider $\tau \in \llbracket P(R) \rrbracket_{Adr}^{Adr}$ and the following step in the standard semantics:

$$(pc_1 \circ pc_3, \tau) \xrightarrow{\tau}_{Q,t} (pc_1' \circ pc_3', \tau.act) \quad \text{with} \quad pc_1 \circ pc_3 \in ctrl(\tau). \quad (1)$$

By definition, $\tau.act \in \llbracket P(R) \rrbracket_{Adr}^{Adr}$. Assume we already constructed for τ some $\sigma \in \llbracket MGC(R) \rrbracket_{Adr}^{Adr}$ with the desired properties. That is, there is pc_2 with $pc_2 \circ pc_3 \in ctrl(\sigma)$. Furthermore, we have the following: $m_\tau^R = m_\sigma^R$, $m_\tau \downarrow_{IVar} = m_\sigma \downarrow_{IVar}$, $\mathcal{H}(\tau) = \mathcal{H}(\sigma)$, $fresh_\tau \subseteq fresh_\sigma$, $freed_\tau \subseteq freed_\sigma$, and $used(\tau) \subseteq used(\sigma)$. We construct a computation $\sigma' \in \llbracket MGC(R) \rrbracket_{Adr}^{Adr}$ that mimics $\tau.act$. To that end, we show that there is a program step of the form:

$$(pc_2 \circ pc_3, \sigma) \xrightarrow{\tau}^* (pc_2' \circ pc_3', \sigma') \quad (2)$$

with the following: $m_{\tau.act}^R = m_{\sigma'}^R$, $m_{\tau.act} \downarrow_{IVar} = m_{\sigma'} \downarrow_{IVar}$, $fresh_{\tau.act} \subseteq fresh_{\sigma'}$, $freed_{\tau.act} \subseteq freed_{\sigma'}$, and $used(\tau.act) \subseteq used(\sigma')$, as well as $\mathcal{H}(\tau.act) = \mathcal{H}(\sigma')$. Let $act = \langle t, com, up \rangle$.

◇ **Case 1:** $Q = R$

Step (1) is due to Rule (SOS-STD-PAR) followed by Rule (SOS-STD-SMR). By definition, we have $pc_1' = pc_1$. Let $stmt_3 = pc_3(t)$. Then, $pc_3' = pc_3[t \mapsto stmt_3']$ with $stmt_3 \xrightarrow{com} stmt_3'$. By definition of Rule (SOS-STD-PAR), we the step $(stmt_2 \circ stmt_3, \sigma) \xrightarrow{R,t} (stmt_2 \circ stmt_3', \sigma.act)$ satisfies (2), provided we have $act \in Act(\sigma, t, com)$. We show that $act \in Act(\sigma, t, com)$ holds and that $\sigma' = \sigma.act$ satisfies the required properties. To do this, we rely on the following:

$$\forall exp. \text{ com contains } exp \implies exp \in IVar \cup Var^R \cup Sel^R \quad (3)$$

$$\forall exp. \text{ com assigns to } exp \implies exp \in Var^R \cup Sel^R \quad (4)$$

$$\forall exp. exp \in IVar \cup Var^R \cup Sel^R \implies m_\tau(exp) = m_\sigma(exp) \quad (5)$$

Implications (3) and (4) follow from $\tau.act$ being free from separation violations. The remaining implication (5) is due to $exp \in dom(m_\tau \downarrow_{IVar}) \cup dom(m_\tau^R)$ by definition together with both $m_\tau^R = m_\sigma^R$ and $m_\tau \downarrow_{IVar} = m_\sigma \downarrow_{IVar}$ by induction. Now, we do a case distinction over com .

◇ **Case 1.1:** $com \in \{in:func(\bar{r}), re:func, env(a)\}$

By definition of Rule (SOS-STD-SMR), the case does apply.

◇ **Case 1.2:** $com \in \{beginAtomic, endAtomic, com \equiv skip\}$

By definition, $act \in Act(\sigma, t, com)$ as required. We conclude by induction:

$$\begin{aligned} m_{\tau.act}^R &= m_\tau^R = m_\sigma^R = m_{\sigma.act}^R \\ m_{\tau.act} \downarrow_{IVar} &= m_\tau \downarrow_{IVar} = m_\sigma \downarrow_{IVar} = m_{\sigma.act} \downarrow_{IVar} \\ fresh_{\tau.act} &= fresh_\tau \subseteq fresh_\sigma = fresh_{\sigma.act} \\ freed_{\tau.act} &= freed_\tau \subseteq freed_\sigma = freed_{\sigma.act} \\ used(\tau.act) &= used(\tau) \subseteq used(\sigma) = used(\sigma.act) \\ \mathcal{H}(\tau.act) &= \mathcal{H}(\tau) = \mathcal{H}(\sigma) = \mathcal{H}(\sigma.act) \end{aligned}$$

◇ **Case 1.3:** $com \equiv exp := exp'$

First, we show $act \in Act(\sigma, t, com)$. To that end, we show that the update up is an appropriate update for com after σ . If $exp \in Var$, then the update is $up = [exp \mapsto m_\tau(exp')]$. From (5), we obtain $m_\tau(exp') = m_\sigma(exp')$. That is, up is appropriate. Otherwise, we have $exp \notin Var$. This means $exp \in Sel^R$ by (3). So exp must be of the form $exp \equiv p.sel$. Let $a = m_\tau(p)$. The update is $up = [a.sel \mapsto m_\tau(exp')]$. From (5), we get $m_\sigma(p) = a$ and $m_\tau(exp') = m_\sigma(exp')$. Again, up is appropriate. Altogether, $act \in Act(\sigma, t, com)$.

Observe that (4) yields $exp \in Var^R \cup Sel^R$. We have $m_\tau(exp') = m_\sigma(exp') = b$ for some address b , as argued above. So we get:

$$\begin{aligned} m_{\tau.act} \downarrow_{IVar} &= m_\tau \downarrow_{IVar} = m_\sigma \downarrow_{IVar} = m_{\sigma.act} \downarrow_{IVar} \\ m_{\tau.act}^R &= m_\tau^R[up] = m_\sigma^R[up] = m_{\sigma.act}^R \end{aligned}$$

The remaining properties follow by definition and induction as before.

◇ **Case 1.4:** $com \equiv \text{assume } cond$

Let exp be an expression in $cond$. Similarly to the previous cases, $m_\tau(exp) = m_\sigma(exp)$ follows (3) and (5). So $act \in Act(\sigma, t, com)$ follows because $cond$ has the same truth value after τ and σ . The remaining properties follow immediately since act does not modify the memory nor affects the fresh/free addresses.

◇ **Case 1.5:** $com \equiv p := \text{malloc}$

Let $a = m_{\tau.act}(p)$ be the allocated address. We have $a \in fresh_\tau \cup freed_\tau$. By induction, we get $a \in fresh_\sigma \cup freed_\sigma$. This yields $act \in Act(\sigma, t, com)$. Moreover, we get:

$$\begin{aligned} fresh_{\tau.act} &= fresh_\tau \setminus \{a\} \subseteq fresh_\sigma \setminus \{a\} = fresh_{\sigma.act} \\ freed_{\tau.act} &= freed_\tau \setminus \{a\} \subseteq freed_\sigma \setminus \{a\} = freed_{\sigma.act} \\ \mathcal{H}(\tau.act) &= \mathcal{H}(\tau) = \mathcal{H}(\sigma) = \mathcal{H}(\sigma.act) \end{aligned}$$

We turn to the remaining properties. We have $up = [p \mapsto a, a.next \mapsto seg, a.data \mapsto d]$ for some d . Observe that $p \in Var^R$ holds by (4). Since $Var^R \cap IVar = \emptyset$, we get:

$$\begin{aligned} m_{\tau.act} \downarrow_{IVar} &= m_\tau \downarrow_{IVar} = m_\sigma \downarrow_{IVar} = m_{\sigma.act} \downarrow_{IVar} \\ used(\tau.act) &= used(\tau) \subseteq used(\sigma) = used(\sigma.act) \end{aligned}$$

For p we have $m_{\tau.act}^R(p) = a = m_{\sigma.act}^R(p)$. For $a.next$ we have:

$$\begin{aligned} m_{\tau.act}^R(a.next) &= seg = m_{\sigma.act}^R(a.next) && \text{if } next \in Sel^R \\ m_{\tau.act}^R(a.next) &= \perp = m_{\sigma.act}^R(a.next) && \text{otherwise.} \end{aligned}$$

Similarly, we obtain $m_{\tau.act}^R(a.data) = m_{\sigma.act}^R(a.data)$. Altogether, this establishes the desired $m_{\tau.act}^R = m_{\sigma.act}^R$ as up does not modify expressions besides p , $a.next$, and $a.data$.

◇ **Case 1.6:** $com \equiv \text{free}(p)$

We have $act \in \text{Act}(\sigma, t, com)$ by definition. Let $m_\tau(p) = a$. As before, $m_\tau(p) = m_\sigma(p)$ follows from (3) and (5). Hence, we conclude as follows:

$$\begin{aligned} \text{fresh}_{\tau.act} &= \text{fresh}_\tau \setminus \{a\} \subseteq \text{fresh}_\sigma \setminus \{a\} = \text{fresh}_{\sigma.act} \\ \text{freed}_{\tau.act} &= \text{freed}_\tau \cup \{a\} \subseteq \text{freed}_\sigma \cup \{a\} = \text{freed}_{\sigma.act} \\ \mathcal{H}(\tau.act) &= \mathcal{H}(\tau).\text{free}(a) = \mathcal{H}(\sigma).\text{free}(a) = \mathcal{H}(\sigma.act) \\ \text{used}(\tau.act) &= \text{used}(\tau) \subseteq \text{used}(\sigma) = \text{used}(\sigma.act) \\ m_{\tau.act} \downarrow_{IVar} &= m_\tau \downarrow_{IVar} = m_\sigma \downarrow_{IVar} = m_{\sigma.act} \downarrow_{IVar} \\ m_{\tau.act}^R &= m_\tau^R = m_\sigma^R = m_{\sigma.act}^R \end{aligned}$$

where the last two equalities are due to $up = \emptyset$.

◇ **Case 2:** $Q = P$ and $com \neq \text{env}(a)$

Step (1) is due to Rule (SOS-STD-PAR). Let $stmt_1 = pc_1(t)$ and $stmt_3 = pc_3(t)$. By definition, $pc_1' = pc_1[t \mapsto stmt_1']$ and $pc_3' = pc_3[t \mapsto stmt_3']$ with $stmt_1 \circ stmt_3 \xrightarrow{com} stmt_1' \circ stmt_3'$.

◇ **Case 2.1:** $com \equiv \text{in}:func(\bar{r})$

Step (1) involves Rule (SOS-STD-CALL): $stmt_3 \equiv \text{skip}$ and $stmt_3' \equiv R.func; \text{await } func$. Assume for a moment we have $stmt_2 \xrightarrow{com} stmt_2'$ and $act \in \text{Act}(\sigma, t, com)$. Then, we satisfies (2) by step $(pc_2 \circ pc_3, \sigma) \vdash_{R,t} (pc_2' \circ pc_3', \sigma.act)$ due to Rule (SOS-STD-CALL) combined with Rule (SOS-STD-PAR). Now, we show that act is enabled after σ . By assumption, $r_i \in IVar$. By induction, $\text{seg} \neq m_\tau(r_i) = m_\sigma(r_i)$. So, $\sigma.act \in \mathcal{O}[P]_{Adr}^{Adr}$ by the assumptions on the MGC. This means we satisfy (2) because the step $stmt_2 \xrightarrow{com} stmt_2'$ exists and $act \in \text{Act}(\sigma, t, com)$ holds. We turn to the remaining properties and establish that $\sigma' = \sigma.act$ is an adequate choice. Let evt be an event such that $\mathcal{H}(\tau.act) = \mathcal{H}(\tau).evt$. Since we have established $m_\tau(r_i) = m_\sigma(r_i)$ already, we get $\mathcal{H}(\sigma.act) = \mathcal{H}(\sigma).evt$. By induction, $\mathcal{H}(\tau.act) = \mathcal{H}(\sigma.act)$. The remaining properties follow by definition together with induction as before since act does not affect the memory nor the fresh/freed/used addresses.

◇ **Case 2.2:** $com \equiv \text{re}:func$

The step is due to Rule (SOS-STD-RETURN): $stmt_3 \equiv \text{await } func$ and $stmt_3' \equiv \text{skip}$. We show that $\sigma' = \sigma.act$ is an appropriate choice. We get $\sigma.act \in \mathcal{O}[P]_{Adr}^{Adr}$ by the assumptions on the MGC. We find a step $(pc_2 \circ pc_3, \sigma) \vdash_{R,t} (pc_2' \circ pc_3', \sigma.act)$ that satisfies (2), as in the previous case. Further, we get:

$$\mathcal{H}(\tau.act) = \mathcal{H}(\tau).\text{re}:func(t) = \mathcal{H}(\sigma).\text{re}:func(t) = \mathcal{H}(\sigma.act)$$

and conclude the remaining properties by definition together with induction as before.

◇ **Case 2.3:** otherwise

The step is due to Rule (SOS-STD-DS). By definition, $stmt_3 \equiv stmt_3$. Observe that we satisfy (2) with $\sigma' = \sigma$ and $\sigma' = \sigma.act'$ for any $act' \in \text{Act}(\sigma, t, com)$. Depending on com , we decide whether or not to append an action and show the remaining properties.

◇ **Case 2.3.1:** $com \equiv p := \text{malloc}$ and $p \notin IVar$

Let $m_{\tau.act}(p) = a$. The update up is $up = [p \mapsto a, a.next \mapsto \text{seg}, a.data \mapsto d]$ with some d and $a \in \text{fresh}_\tau \cup \text{freed}_\tau$. By induction, we have $a \in \text{fresh}_\sigma \cup \text{freed}_\sigma$. Let q be some pointer variable of the MGC such that $q \notin IVar$ and $q \notin \text{Var}^R$. We show that $\sigma' = \sigma.act'$ is an appropriate choice for $act' = \langle t, q := \text{malloc}, up' \rangle$ with update $up' = [q \mapsto a, a.next \mapsto \text{seg}, a.data \mapsto d]$. By the assumptions on the MGC, we have $\sigma.act' \in \mathcal{O}[P]_{Adr}^{Adr}$. That is, $act' \in \text{Act}(\sigma, t, com)$ holds. As stated before, this satisfies (2). By induction and definition:

$$\begin{aligned} m_{\tau.act} \downarrow IVar &= m_\tau \downarrow IVar = m_\sigma \downarrow IVar = m_{\sigma.act} \downarrow IVar \\ \mathcal{H}(\tau.act) &= \mathcal{H}(\tau) = \mathcal{H}(\sigma) = \mathcal{H}(\sigma.act') \\ \text{fresh}_{\tau.act} &= \text{fresh}_\tau \subseteq \{a\} \subseteq \text{fresh}_\sigma \subseteq \{a\} = \text{fresh}_{\sigma.act'} \\ \text{freed}_{\tau.act} &= \text{freed}_\tau \setminus \{a\} \subseteq \text{freed}_\sigma \setminus \{a\} = \text{freed}_{\sigma.act'} \end{aligned}$$

Next, observe that $p \in \text{Var}^P$ since $\tau.act$ is free from separation violations. Hence, we obtain:

$$\begin{aligned} \text{used}(\tau.act) &= \text{used}(\tau) \cup \{m_{\tau.act}(p)\} = \text{used}(\tau) \cup \{m_{\sigma.act'}(q)\} \\ &\subseteq \text{used}(\sigma) \cup \{m_{\sigma.act'}(q)\} = \text{used}(\sigma.act') \end{aligned}$$

It remains to show $m_{\tau.act}^R = m_{\sigma.act'}^R$. It suffices to show $m_{\tau.act}^R(a.sel) = m_{\sigma.act'}^R(a.sel)$ for all selectors $sel \in \text{Sel}^R$ since $p \notin IVar$ (the selectors of all other addresses remain unchanged). This follows from the fact that up and up' agree on the updates they perform on selectors. Formally, we have $m_{\tau.act}^R(a.sel) = v = m_{\sigma.act'}^R(a.sel)$ where we use $v = a$ if $sel = \text{next}$ and $v = d$ if $sel = \text{data}$.

◇ **Case 2.3.2:** $com \equiv p := \text{malloc}$ and $p \in IVar$

Follows analogously to the previous case. Here, we can simply choose $\sigma' = \sigma.act$ and observe that $m_{\tau.act}^R(p) = m_{\sigma.act}^R(p)$ holds in order to obtain $m_{\tau.act}^R = m_{\sigma.act}^R$.

◇ **Case 2.3.3:** $com \equiv x := \text{exp}$ and $x \in IVar$

Let $up = [x \mapsto v]$. We show that $\sigma' = \sigma.act'$ with $act' = \langle t, com', up \rangle$ is appropriate. By the assumptions on the MGC, act' exists and is enabled, $\sigma.act' \in \mathcal{O}[P]_{Adr}^{Adr}$. We have $act' \in \text{Act}(\sigma, t, com)$ and thus satisfy (2) as stated before. Moreover, we get:

$$m_{\tau.act} \downarrow IVar = m_\tau \downarrow IVar[up] = m_\sigma \downarrow IVar[up] = m_{\sigma.act} \downarrow IVar$$

and conclude the remaining properties by definition and induction as before.

◇ **Case 2.3.4:** $com \equiv \text{exp} := \text{exp}'$ with $\text{exp} \notin IVar$

We show that $\sigma' = \sigma$ satisfies the desired properties. We get (2) for $pc'_2 = pc_2$. Next, we show $m_{\tau.act}^R = m_\sigma^R$. To do so, it is sufficient to show $m_{\tau.act}^R = m_\tau^R$ since $m_\tau^R = m_\sigma^R$ holds by induction. To the contrary, assume $m_{\tau.act}^R \neq m_\tau^R$. Note that, by definition, we have $\text{dom}(m_\tau^R) = \text{dom}(m_{\tau.act}^R)$. Consider $p \in \text{dom}(m_\tau^R)$. Because $\tau.act$ is free from separation violations, we have $\text{exp} \neq p$. Hence, $m_{\tau.act}^R(p) = m_\tau^R(p)$ holds. That is, there must be $a.sel \in \text{dom}(m_\tau^R)$ such that $m_{\tau.act}^R(a.sel) \neq m_\tau^R(a.sel)$. Hence, we have $\text{exp} \equiv q.sel$

with $m_\tau(q) = a$. Observe that $a.\text{sel} \in \text{dom}(m_\tau^R)$ means $\text{sel} \in \text{Sel}^R$. So, act is a separation violation. This contradicts the assumptions and thus concludes the desired $m_{\tau.\text{act}}^R = m_\tau^R$. The remaining properties follow by definition together with induction.

◇ **Case 2.3.5:** $\text{com} \in \{\text{assume } \text{cond}, \text{beginAtomic}, \text{endAtomic}, \text{skip}\}$

We can choose $\sigma' = \sigma$ and immediately obtain the desired properties by induction.

◇ **Case 3:** $Q = P$ and $\text{com} \equiv \text{env}(a)$

Step (1) is due to Rule (SOS-STD-ENV). We have $pc_1' \equiv pc_1$ and $pc_3' \equiv pc_3$ and $t = \perp$. By definition, $a \in \text{fresh}_\tau \cup \text{freed}_\tau$. By induction, $a \in \text{fresh}_\sigma \cup \text{freed}_\sigma$. So, $\text{act} \in \text{Act}(\sigma, \perp, \text{com})$. That is, we obtain the step $(pc_2 \circ pc_3, \sigma) \dashv\vdash_{P,\perp} (pc_2 \circ pc_3, \sigma.\text{act})$ which satisfies (2). Next, we establish $m_{\tau.\text{act}}^R = m_{\sigma.\text{act}}^R$. To that end, consider $\text{exp} \in \text{PExp} \cup \text{DExp}$. If $\text{exp} \cap \text{Adr} \neq \{a\}$, then we get $m_{\tau.\text{act}}^R \text{exp} = m_\tau^R \text{exp} = m_\sigma^R \text{exp} = m_{\sigma.\text{act}}^R \text{exp}$ where the second equality holds by induction and the first/third equality holds by up . It remains to show, for all $\text{sel} \in \text{Sel}^R$, that $m_{\tau.\text{act}}^R(a.\text{sel}) = m_{\sigma.\text{act}}^R(a.\text{sel})$ holds. This follows from the fact that up and up' agree on the updates they perform on selectors. The remaining properties follow by induction since act does not affect the control locations nor the valuation of variables nor the history nor the fresh/freed/used addresses.

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

Proof C.6 (Corollary A.3). Consequence of Theorem A.2. ■

Proof C.7 (Theorem A.4). We proceed by induction over the structure of τ . In the base case, we have the empty computation $\tau = \epsilon$. Then, the claim follows by definition for $\sigma = \epsilon$. For the induction step, consider $\tau \in \llbracket P(R) \rrbracket_{\text{Adr}}^{\text{Adr}}$ and the following program step in the standard semantics:

$$(pc_1 \circ pc_2, \tau) \dashv\vdash_{Q,t} (pc_1' \circ pc_2', \tau.\text{act}) \quad \text{with} \quad pc_1 \circ pc_2 \in \text{ctrl}(\tau). \quad (6)$$

By definition, $\tau.\text{act} \in \llbracket P(R) \rrbracket_{\text{Adr}}^{\text{Adr}}$. Assume we already constructed for τ some $\sigma \in \llbracket P(R) \rrbracket_{\text{Adr}}^{\text{Adr}}$ with the following: $\text{stmt}_1 \in \text{ctrl}(\sigma)$, $m_\tau^P = m_\sigma^P$, $\mathcal{H}(\tau) = \mathcal{H}(\sigma)$, $\text{fresh}_\tau \subseteq \text{fresh}_\sigma$, $\text{freed}_\tau \subseteq \text{freed}_\sigma$, and $\text{retired}_\tau \subseteq \text{retired}_\sigma$. We now construct a computation $\sigma' \in \llbracket P(R) \rrbracket_{\text{Adr}}^{\text{Adr}}$ that mimics $\tau.\text{act}$. More precisely, we show that there is a program step in the SMR semantics of the form

$$(pc_1, \sigma) \rightarrow^* (pc_1', \sigma') \quad (7)$$

satisfying the following: $m_{\tau.\text{act}}^P = m_{\sigma'}^P$, $\mathcal{H}(\tau.\text{act}) = \mathcal{H}(\sigma')$, $\text{fresh}_{\tau.\text{act}} \subseteq \text{fresh}_{\sigma'}$, $\text{freed}_{\tau.\text{act}} \subseteq \text{freed}_{\sigma'}$, and $\text{retired}_{\tau.\text{act}} \subseteq \text{retired}_{\sigma'}$. Let $\text{act} = \langle t, \text{com}, up \rangle$.

◇ **Case 1:** $Q = R$

Step (6) is due to Rule (SOS-STD-PAR) followed by Rule (SOS-STD-SMR). By definition, we have $pc_1' = pc_1$. Let $\text{stmt}_2 = pc_2(t)$. Then, $pc_2' = pc_2[t \mapsto \text{stmt}_2]$ with $\text{stmt}_2 \xrightarrow{\text{com}} \text{stmt}_2'$.

◇ **Case 1.1:** $\text{com} \in \{\text{in:func}(\bar{r}), \text{re:func}\}$

According to Rule (SOS-STD-SMR), the case does not apply.

◇ **Case 1.2:** $com \in \{\text{skip}, \text{beginAtomic}, \text{endAtomic}, \text{assume } cond\}$

We show that $\sigma' = \sigma$ is an appropriate choice. We immediately satisfy (7) by $pc_1' = pc_1$. For the remaining properties, we conclude by induction as follows:

$$\begin{aligned} m_{\tau.act}^P &= m_\tau^P = m_\sigma^P = m_{\sigma.act}^P \\ \mathcal{H}(\tau.act) &= \mathcal{H}(\tau) = \mathcal{H}(\sigma) = \mathcal{H}(\sigma.act) \\ \text{fresh}_{\tau.act} &= \text{fresh}_\tau \subseteq \text{fresh}_\sigma = \text{fresh}_{\sigma.act} \\ \text{freed}_{\tau.act} &= \text{freed}_\tau \subseteq \text{freed}_\sigma = \text{freed}_{\sigma.act} \\ \text{retired}_{\tau.act} &= \text{retired}_\tau \subseteq \text{retired}_\sigma = \text{retired}_{\sigma.act} \end{aligned}$$

◇ **Case 1.3:** $com \equiv exp := exp'$

We choose $\sigma' = \sigma$. We immediately satisfy (7) by $pc_1' = pc_1$. Next, we show $m_{\tau.act}^P = m_\sigma^P$. To that end, it suffices to establish $m_{\tau.act}^P = m_\tau^P$. First, consider the case where $exp \in Var$ holds. Then, $exp \in Var^R$ because $\tau.act$ is free from separation violations. By definition, we obtain $m_{\tau.act} = m_\tau[exp \mapsto m_\tau(exp')]$. That is, $m_{\tau.act}^P = m_\tau^P$ because $Var^R \cap Var^P = \emptyset$. Second, consider the remaining case where $exp \notin Var$ holds. So, $exp \equiv p.sel$ for some pointer p and selector sel . Let $a = m_\tau(p)$. We have $m_{\tau.act} = m_\tau[a.sel \mapsto m_\tau(exp')]$. Since $\tau.act$ is free from separation violations, $sel \in Sel^R$. So, we get $m_{\tau.act}^P = m_\tau^P$ because of $Sel^R \cap Sel^P = \emptyset$. The remaining properties follow by definition together with induction as before.

◇ **Case 1.4:** $com \equiv p := \text{malloc}$

Let $a = m_{\tau.act}(p)$. The update is $up = [p \mapsto a, a.next \mapsto \text{seg}, a.data \mapsto d]$ for some d . We choose $\sigma' = \sigma.act'$ with $act' = \langle t, \text{env}(a), up' \rangle$ and $up' = [a.next \mapsto \text{seg}, a.data \mapsto d]$. By definition, we have $a \in \text{fresh}_\tau \cup \text{freed}_\tau$. Hence, $a \in \text{fresh}_\sigma \cup \text{freed}_\sigma$ by induction. So we get $act' \in \overline{Act}(\sigma, t, \text{env}(a))$. Rule (SOS-ENV) yields the step $(pc_1, \sigma) \rightarrow (pc_1, \sigma.act')$ which satisfies (7). Next, we show $m_{\tau.act}^P = m_{\sigma.act'}^P$. To that end, we establish, for all expressions exp , that $m_{\tau.act}^P(exp) = m_{\sigma.act'}^P(exp)$ holds. By induction, we have:

$$m_{\tau.act}^P(exp) = m_\tau^P(exp) = m_\sigma^P(exp) = m_{\sigma.act'}^P(exp) \quad \text{if } exp \notin \{p, a.next, a.data\}.$$

We turn to p , $a.next$, and $a.data$. Regarding p , note that we have $p \in Var^R$ because $\tau.act$ is free from separation violations. That is, $m_{\tau.act}^P(p) = \perp = m_{\sigma.act'}^P(p)$. Consider $a.next$. If $next \in Sel^P$, we get $m_{\tau.act}^P(a.next) = a = m_{\sigma.act'}^P(a.next)$ due to the form of the updates up and up' . Otherwise, we have $m_{\tau.act}^P(a.next) = \perp = m_{\sigma.act'}^P(a.next)$ by definition of the memory separation. Similarly, we obtain $m_{\tau.act}^P(a.data) = m_{\sigma.act'}^P(a.data)$. Altogether, we conclude the desired $m_{\tau.act}^P = m_{\sigma.act'}^P$. The remaining properties follow by definition and induction as before.

◇ **Case 1.5:** $com \equiv \text{free}(p)$

Let $m_\tau(p) = a$. We choose $\sigma' = \sigma.act'$ with $act' = \langle t, \text{free}(a), \emptyset \rangle$. By the standard semantics, we have $act' \in Act(\tau, t, com)$. Hence, $act' \in \overline{Act}(\sigma, \perp, \text{free}(a))$ holds according to the SMR semantics. Then, Rule (SOS-FREE) yields $(pc_1, \sigma) \rightarrow (pc_1, \sigma.act')$ which satisfies (7). For the remaining properties, we conclude by definition and induction as before.

◇ **Case 2:** $Q = P$ and $com \neq \text{env}(a)$

Step (6) is due to Rule (SOS-STD-PAR). Let $stmt_1 = pc_1(t)$ and $stmt_2 = pc_2(t)$. By definition, $pc_1' = pc_1[t \mapsto stmt_1']$ and $pc_2' = pc_2[t \mapsto stmt_2']$ with $stmt_1 \circ stmt_2 \xrightarrow{com} stmt_1' \circ stmt_2'$. We show that $\sigma' = \sigma.act$ is an appropriate choice. By the SMR semantics, $stmt_1 \xrightarrow{com} stmt_1'$ holds. Hence, Rule (SOS-PAR) yields $(pc_1, \sigma) \rightarrow (pc_1', \sigma.act)$ satisfying (7), provided we have $act \in \overline{Act}(\sigma, t, com)$. We establish $act \in \overline{Act}(\sigma, t, com)$ and the remaining properties.

◇ **Case 2.1:** $com \in \{\text{skip}, \text{beginAtomic}, \text{endAtomic}\}$

By definition, $act \in \overline{Act}(\sigma, t, com)$. We conclude by definition and induction:

$$\begin{aligned} m_{\tau.act}^P &= m_\tau^P = m_\sigma^P = m_{\sigma.act}^P \\ \mathcal{H}(\tau.act) &= \mathcal{H}(\tau) = \mathcal{H}(\sigma) = \mathcal{H}(\sigma.act) \\ \text{fresh}_{\tau.act} &= \text{fresh}_\tau \subseteq \text{fresh}_\sigma = \text{fresh}_{\sigma.act} \\ \text{freed}_{\tau.act} &= \text{freed}_\tau \subseteq \text{freed}_\sigma = \text{freed}_{\sigma.act} \\ \text{retired}_{\tau.act} &= \text{retired}_\tau \subseteq \text{retired}_\sigma = \text{retired}_{\sigma.act} \end{aligned}$$

◇ **Case 2.2:** $com \equiv \text{in:func}(\bar{r})$

Step (6) involves Rule (SOS-STD-CALL). By assumption, we have $r_i \in IVar$. By induction, we get $m_\tau(r_i) = m_\sigma(r_i)$. Hence, $m_\sigma(r_i) \neq \text{seg}$ because $m_\tau(r_i) \neq \text{seg}$ according to the semantics. This gives $act \in \overline{Act}(\sigma, t, com)$ by definition. Now, let evt be the event emitted by act after τ , that is, $\mathcal{H}(\tau.act) = \mathcal{H}(\tau).evt$. Because we have already established $m_\tau(r_i) = m_\sigma(r_i)$, act must emit the same event after σ , i.e., $\mathcal{H}(\sigma.act) = \mathcal{H}(\sigma).evt$. So, $\mathcal{H}(\tau.act) = \mathcal{H}(\sigma.act)$ follows by induction. For the remaining property, let $M \subseteq \text{Adr}$ such that $M = \{a\}$ if $evt \equiv \text{in:retire}(t, a)$ and $M = \emptyset$ otherwise. Then, we get:

$$\text{retired}_{\tau.act} = \text{retired}_\tau \cup M \subseteq \text{retired}_\sigma \cup M = \text{retired}_{\sigma.act}.$$

The remaining properties follow by definition and induction as before.

◇ **Case 2.3:** $com \equiv \text{re:func}$

Step (6) involves Rule (SOS-STD-RETURN). By definition, $act \in \overline{Act}(\sigma, t, \text{re:func})$. We get:

$$\mathcal{H}(\tau.act) = \mathcal{H}(\tau).\text{re:func}(t) = \mathcal{H}(\sigma).\text{re:func}(t) = \mathcal{H}(\sigma.act)$$

and conclude the remaining properties by definition and induction as before.

◇ **Case 2.4:** $com \equiv p := \text{malloc}$

Let $a = m_{\tau.act}(p)$. Then, the update is $up = [p \mapsto a, a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d]$ for some data value d .

By definition, $a \in \text{fresh}_\tau \cup \text{freed}_\tau$. So, $a \in \text{fresh}_\sigma \cup \text{freed}_\sigma$ by induction. This means $\text{act} \in \overline{\text{Act}}(\sigma, t, \text{com})$. Moreover, we get:

$$\begin{aligned}\text{fresh}_{\tau.\text{act}} &= \text{fresh}_\tau \setminus \{a\} \subseteq \text{fresh}_\sigma \setminus \{a\} = \text{fresh}_{\sigma.\text{act}} \\ \text{freed}_{\tau.\text{act}} &= \text{freed}_\tau \setminus \{a\} \subseteq \text{freed}_\sigma \setminus \{a\} = \text{freed}_{\sigma.\text{act}} \\ \text{retired}_{\tau.\text{act}} &= \text{retired}_\tau \subseteq \text{retired}_\sigma = \text{retired}_{\sigma.\text{act}} \\ \mathcal{H}(\tau.\text{act}) &= \mathcal{H}(\tau) = \mathcal{H}(\sigma) = \mathcal{H}(\sigma.\text{act})\end{aligned}$$

It remains to establish $m_{\tau.\text{act}}^P = m_{\sigma.\text{act}}^P$. By induction and the form of up , we have:

$$m_{\tau.\text{act}}^P(\text{exp}) = m_\tau^P(\text{exp}) = m_\sigma^P(\text{exp}) = m_{\sigma.\text{act}'}^P(\text{exp}) \quad \text{if} \quad \text{exp} \notin \{p, a.\text{next}, a.\text{data}\}.$$

Hence, it suffices to show $m_{\tau.\text{act}}^P(\text{exp}') = m_{\sigma.\text{act}}^P(\text{exp}')$ for $\text{exp}' \in \{p, a.\text{next}, a.\text{data}\}$. By the definition of the memory separation, it suffices to show $m_{\tau.\text{act}}(\text{exp}') = m_{\sigma.\text{act}}(\text{exp}')$. This follows immediately from the performed update up .

◇ **Case 2.5:** $\text{com} \equiv p.\text{sel} := \text{exp}$

By definition of the syntax, we have $\text{exp} \in \text{Var}$. Let $a = m_\tau(p)$ and let $v = m_\tau(\text{exp})$. Then, the update is $up = [a.\text{sel} \mapsto v]$. Since $\tau.\text{act}$ is free from separation violations, we get $p, \text{exp} \in \text{Var}^P$ and $\text{sel} \in \text{Sel}^P$. Hence, $\{p, \text{exp}, a.\text{sel}\} \subseteq \text{dom}(m_\tau^P)$. By induction, we have $m_\sigma(p) = a$ and $m_\sigma(\text{exp}) = v$. This means up is a valid for act after σ . That is, we obtain $\text{act} \in \overline{\text{Act}}(\sigma, t, \text{com})$. From $\text{sel} \in \text{Sel}^P$ we get:

$$m_{\tau.\text{act}}^P = m_\tau^P[a.\text{sel} \mapsto v] = m_\sigma^P[a.\text{sel} \mapsto v] = m_{\sigma.\text{act}}^P.$$

The remaining properties follow by definition and induction as before.

◇ **Case 2.6:** $\text{com} \equiv p := \text{exp}'$

Analogous to the previous case.

◇ **Case 2.7:** $\text{com} \equiv \text{assume } \text{cond}$

Let exp be an expression in cond . Similarly to the previous cases, $m_\tau(\text{exp}) = m_\sigma(\text{exp})$ by induction together with the fact that $\tau.\text{act}$ is free from separation violations and thus only variables from Var^P and selectors from Sel^P can occur in exp . Then, we arrive at $\text{act} \in \overline{\text{Act}}(\sigma, t, \text{com})$ since cond has the same truth value after τ and σ . The remaining properties follow by induction.

◇ **Case 3:** $Q = P$ and $\text{com} \equiv \text{env}(a)$

Step (6) is due to Rule (SOS-STD-ENV). By definition of the rule, we have $pc_1 = pc_1'$ as well as $up = [a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d]$ for some value d . By definition, $a \in \text{fresh}_\tau \cup \text{freed}_\tau$. By induction, $a \in \text{fresh}_\sigma \cup \text{freed}_\sigma$. Hence, we obtain $\text{act} \in \overline{\text{Act}}(\sigma, t, \text{com})$ such that the step $(pc_1, \sigma) \rightarrow (pc_1', \sigma.\text{act})$ by Rule (SOS-ENV) satisfies (7). Next, we show $m_{\tau.\text{act}}^P = m_{\sigma.\text{act}}^P$. By induction together with the form of up , we have:

$$m_{\tau.\text{act}}^P(\text{exp}) = m_\tau^P(\text{exp}) = m_\sigma^P(\text{exp}) = m_{\sigma.\text{act}'}^P(\text{exp}) \quad \text{if} \quad \text{exp} \notin \{a.\text{next}, a.\text{data}\}.$$

Hence, it suffices to show $m_{\tau.act}^P(exp') = m_{\sigma.act}^P(exp')$ for $exp' \in \{p, a.next, a.data\}$. By the definition of the memory separation, it suffices to show $m_{\tau.act}(exp') = m_{\sigma.act}(exp')$. This follows from the performed update up . The remaining properties follow by definition and induction as before.

The above case distinction is complete and thus concludes the induction. \blacksquare

Proof C.8 (Theorem 5.10). Note that Theorem A.4 implicitly assumes that $\llbracket P(R) \rrbracket_{Adr}^{Adr}$ is free from separation violations—these requirements were stated informally in Section 5.3. This means that Theorem A.4 is applicable. Consider now some computation $\tau \in \llbracket P(R) \rrbracket_{Adr}^{Adr}$. From Theorem A.4 we get $\sigma \in \mathcal{O}\llbracket P \rrbracket_{Adr}^{Adr}$ with $ctrl^P(\tau) = ctrl(\sigma)$. By assumption, we have $good(\sigma)$. That is, $ctrl^P(\sigma) \cap Fault = \emptyset$. From this we get $ctrl^P(\tau) \cap Fault = \emptyset$. This gives $good(\tau)$ as required. \blacksquare

Proof C.9 (Theorem 5.11). As noted in Proof C.8 already, Theorem 5.11 comes with the implicit assumption that $\llbracket P(R) \rrbracket_{Adr}^{Adr}$ is free from separation violations. Towards a contradiction, assume that $\llbracket P(R) \rrbracket_{Adr}^{Adr}$ is not free from double retires. That is, there is a computation $\tau.act \in \llbracket P(R) \rrbracket_{Adr}^{Adr}$ with $act = \langle t, in:retire(p), up \rangle$ and $m_{\tau.act}(p) \in retired_\tau$. Theorem A.4 yields $\sigma \in \mathcal{O}\llbracket P \rrbracket_{Adr}^{Adr}$ with $m_\tau^P = m_\sigma^P$ and $retired_\tau \subseteq retired_\sigma$. We obtain $\sigma.act \in \mathcal{O}\llbracket P \rrbracket_{Adr}^{Adr}$. To see that act is enabled, note that $p \in IVar \subseteq Var^P$ by assumption and thus $m_\sigma(p) = m_\tau(p) \neq seg$. Moreover, this means $m_\sigma(p) \in retired_\sigma$. That is, $\sigma.act$ is a double retire. This contradicts the assumption of the semantics $\mathcal{O}\llbracket P \rrbracket_{Adr}^{Adr}$ being free from double retires. \blacksquare

C.2 Ownership

Proof C.10 (Theorem 6.7). We show the contrapositive:

$$\forall \tau, p, t. \quad p \notin local_t \wedge p \in valid_\tau \implies m_\tau(p) \notin owned_\tau(t).$$

To that end, we proceed by induction over the structure of $\tau \in \mathcal{O}\llbracket P \rrbracket_{Adr}^{Adr}$. In the base case, $\tau = \epsilon$. Then, the claim follows by $owned_\tau(t) = \emptyset$. For the induction step, consider $\tau.act \in \mathcal{O}\llbracket P \rrbracket_{Adr}^{Adr}$ and assume that the claim holds for τ . Consider some thread t and some $x \in PVar \setminus local_t$ such that $p \in valid_\tau$. We show that $m_\tau(p) \notin owned_\tau(t)$ holds. Let $act = \langle t', com, up \rangle$.

◇ **Case 1:** $t \neq t'$

By definition, we have $owned_{\tau.act}(t) \subseteq owned_\tau(t)$.

◇ **Case 1.1:** $x \notin shared$

If $x \notin shared$, then x cannot occur in com by the semantics. Hence, $x \in valid_{\tau.act}$ implies $x \in valid_\tau$. Moreover, $m_{\tau.act}(x) = m_\tau(x)$. By induction, $m_\tau(x) \notin owned_\tau(t)$. Hence, we obtain $m_{\tau.act}(x) \notin owned_{\tau.act}(t)$ as required.

◇ **Case 1.2:** $x \in shared$ and $[x \mapsto \bullet] \notin up$

That x does not receive an update means that it is not the target of an assignment nor an allocation. We

get $m_\tau(x) = m_{\tau.act}(x)$ by definition. Moreover, we obtain $x \in \text{valid}_\tau$ by $x \in \text{valid}_{\tau.act}$. By induction, $m_\tau(x) \notin \text{owned}_\tau(t)$. Hence, $m_{\tau.act}(x) \notin \text{owned}_{\tau.act}(t)$ as required.

◇ **Case 1.3:** $x \in \text{shared}$ and $[x \mapsto a] \subseteq \text{up}$

By $\text{owned}_{\tau.act}(t) \subseteq \text{owned}_\tau(t)$, we know that com cannot be an allocation targeting x . So, $\text{com} \equiv x := \text{pexp}$. First, consider $\text{pexp} \in \text{PVar}$. To arrive at $x \in \text{valid}_{\tau.act}$, we must have $\text{pexp} \in \text{valid}_\tau$. As this gives a contradicting $m_{\tau.act}(x) = a \notin \text{owned}_{\tau.act}(t)$, the case cannot apply. That is, $\text{pexp} \equiv p.\text{next}$. Let $b = m_\tau(p)$. To arrive at $x \in \text{valid}_{\tau.act}$, we must have $p, b.\text{next} \in \text{valid}_\tau$. By definition, this results in $m_{\tau.act}(x) = a \notin \text{owned}_{\tau.act}(t)$. Hence, the case cannot apply.

◇ **Case 2:** $t = t'$

We distinguish three cases.

◇ **Case 2.1:** $x \notin \text{shared}$

By the semantics, x cannot occur in com . We get $x \in \text{valid}_\tau$ and $m_\tau(x) = m_{\tau.act}(x)$. Hence, $m_{\tau.act}(x) \notin \text{owned}_\tau(t)$. If $\text{owned}_{\tau.act}(t) \subseteq \text{owned}_\tau(t)$, then nothing remains to be shown. Consider now $\text{owned}_{\tau.act}(t) \not\subseteq \text{owned}_\tau(t)$. By definition, this means we must have $\text{com} \equiv p := \text{malloc}$ and thus $\text{owned}_{\tau.act}(t) = \text{owned}_\tau(t) \cup \{a\}$ where $a = m_{\tau.act}(p)$. If $m_\tau(x) = a$, then $x \notin \text{valid}_\tau$ by the definition of validity. Since this contradicts the previous $x \in \text{valid}_\tau$, we must have $m_\tau(x) \neq a$. Hence, $m_{\tau.act}(x) \notin \text{owned}_{\tau.act}(t)$ follows as required.

◇ **Case 2.2:** $x \in \text{shared}$ and $[x \mapsto \bullet] \not\subseteq \text{up}$

That x does not receive an update means it is not the target of an assignment nor an allocation. We get $x \in \text{valid}_\tau$ and $m_\tau(x) = m_{\tau.act}(x)$. We conclude as in the previous case.

◇ **Case 2.3:** $x \in \text{shared}$ and $[x \mapsto a] \subseteq \text{up}$

To the contrary, assume $\text{com} \equiv x := \text{malloc}$. This means $m_{\tau.act}(x) \in \text{fresh}_\tau \cup \text{freed}_\tau$. By definition, $m_{\tau.act}(x) \notin \text{owned}_\tau(t)$. Because of $x \in \text{shared}$, the allocated address is not owned, that is, $m_{\tau.act}(x) \notin \text{owned}_{\tau.act}(t)$ by definition. Since this contradicts the choice of x , we must have $\text{com} \not\equiv x := \text{malloc}$. Hence, we get $m_{\tau.act}(x) \in \text{owned}_{\tau.act}(t) \subseteq \tau t$.

Because com is no allocation but updates x , it must be an assignment, $\text{com} \equiv x := \text{pexp}$. By $x \in \text{shared}$, we must have $\text{pexp} \in \text{PVar}$ in order to get $m_{\tau.act}(x) = a \in \text{owned}_{\tau.act}(t)$. To get $x \in \text{valid}_{\tau.act}$, we must have $\text{pexp} \in \text{valid}_\tau$. We get $m_{\tau.act}(x) = a \notin \text{owned}_{\tau.act}(t)$. Since this contradicts the choice of x , the case cannot apply.

The above case distinction is complete and thus concludes the induction. ■

C.3 Reductions

Proof C.11 (Lemma B.31). By definition. ■

Proof C.12 (Lemma B.32). By definition. ■

Proof C.13 (Lemma B.33). By definition. ■

Proof C.14 (Lemma B.34). By definition. ■

Proof C.15 (Lemma B.35). By definition we have:

$$\begin{aligned}
 \text{dom}(m_\tau|_{\text{valid}_\tau}) \cap \text{Adr} &= (\text{valid}_\tau \cup \text{DVar} \cup \{ a.\text{data} \mid a \in m_\tau(\text{valid}_\tau) \}) \cap \text{Adr} \\
 &= (\text{valid}_\tau \cap \text{Adr}) \cup (\{ a.\text{data} \mid a \in m_\tau(\text{valid}_\tau) \} \cap \text{Adr}) \\
 &= (\text{valid}_\tau \cap \text{Adr}) \cup (\{ a \mid a \in m_\tau(\text{valid}_\tau) \}) \\
 &= (\text{valid}_\tau \cap \text{Adr}) \cup m_\tau(\text{valid}_\tau) \\
 \text{and } \text{range}(m_\tau|_{\text{valid}_\tau}) \cap \text{Adr} &= m_\tau(\text{dom}(m_\tau|_{\text{valid}_\tau})) \cap \text{Adr} = m_\tau(\text{dom}(m_\tau|_{\text{valid}_\tau}) \cap \text{PExp}) \\
 &= m_\tau(\text{valid}_\tau)
 \end{aligned}$$

so that we conclude as follows

$$\begin{aligned}
 \text{adr}(m_\tau|_{\text{valid}_\tau}) \cap \text{Adr} &= (\text{dom}(m_\tau|_{\text{valid}_\tau}) \cap \text{range}(m_\tau|_{\text{valid}_\tau})) \cap \text{Adr} \\
 &= (\text{valid}_\tau \cap \text{Adr}) \cup m_\tau(\text{valid}_\tau) \cup m_\tau(\text{valid}_\tau) \\
 &= (\text{valid}_\tau \cap \text{Adr}) \cup m_\tau(\text{valid}_\tau) .
 \end{aligned}$$
■

Proof C.16 (Lemma B.36). The claim holds for ϵ since $\text{valid}_\tau = \text{PVar}$ and $\text{PVar} \subseteq \text{dom}(m_\epsilon)$ by definition. Towards a contradiction, assume the claim does not hold. Then, there is a shortest computation $\tau.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$ with $\text{valid}_{\tau.\text{act}} \not\subseteq \text{dom}(m_{\tau.\text{act}})$. That is, there is $p\text{exp} \in \text{valid}_{\tau.\text{act}}$ with $m_{\tau.\text{act}}(p\text{exp}) = \perp$. First, consider the case $p\text{exp} \notin \text{valid}_\tau$. In order to validate $p\text{exp}$, act must be (i) an allocation $p := \text{malloc}$ with $m_{\tau.\text{act}}(p) = a$ as well as $p\text{exp} \in \{p, a.\text{next}\}$, (ii) an assumption $\text{assume } p\text{exp} = q$ with $q \in \text{valid}_\tau$, or (iii) an assignment of the form $p\text{exp} := q\text{exp}$ with $q\text{exp} \in \text{valid}_\tau$. Case (i) cannot apply as it results in $p\text{exp} \in \text{dom}(m_{\tau.\text{act}})$. Case (ii) cannot apply because $m_\tau(p\text{exp}) = m_\tau(q)$ together with $q \in \text{dom}(m_\tau) = \text{dom}(m_{\tau.\text{act}})$ by minimality gives $p\text{exp} \in \text{dom}(m_{\tau.\text{act}})$. So case (iii) must apply. By minimality, however, $q\text{exp} \in \text{valid}_\tau$ yields $m_\tau(q\text{exp}) \neq \perp$. We get $m_{\tau.\text{act}}(p\text{exp}) \neq \perp$, contradicting the choice of $p\text{exp}$. So the case does not apply and $p\text{exp} \in \text{valid}_\tau$ must hold.

Consider now the $p\text{exp} \in \text{valid}_\tau$. By minimality, $m_\tau(p\text{exp}) \neq \perp$. So act must update $p\text{exp}$ to \perp . To do that, act must be an assignment $p\text{exp} := q\text{exp}$ with $m_\tau(q\text{exp}) = \perp$. We have $q\text{exp} \notin \text{valid}_\tau$. Hence, we get $p\text{exp} \notin \text{valid}_{\tau.\text{act}}$. This contradicts the choice of $p\text{exp}$. ■

Proof C.17 (Lemma B.37). We conclude as follows using the definition of $\tau \sim \sigma$, set theory, and the definition of restrictions:

$$\begin{aligned}
 \tau \sim \sigma &\implies m_\tau|_{\text{valid}_\tau} = m_\sigma|_{\text{valid}_\sigma} \implies \text{dom}(m_\tau|_{\text{valid}_\tau}) = \text{dom}(m_\sigma|_{\text{valid}_\sigma}) \\
 &\implies \text{dom}(m_\tau|_{\text{valid}_\tau}) \cap \text{PExp} = \text{dom}(m_\sigma|_{\text{valid}_\sigma}) \cap \text{PExp} \implies \text{valid}_\tau = \text{valid}_\sigma
 \end{aligned}$$

where the last implication is due to Lemma B.36. ■

Proof C.18 (Lemma B.38). Follows immediately from $\tau \sim \sigma$, Lemma B.35, and Lemma B.37. ■

Proof C.19 (Lemma B.39). Let $\tau \sim \sigma$ and let $com \in next-com(\tau)$. By definition, $pc \in ctrl(\tau)$ exists such that for some thread t we have $pc(t) \xrightarrow{com} \bullet$. By $\tau \sim \sigma$, we have $ctrl(\tau) = ctrl(\sigma)$. Hence, $com \in next-com(\sigma)$. ■

Proof C.20 (Lemma B.40). Let $\tau.act \in \mathcal{O}[[P]]_{Adr}^{Adr}$ with $act = \langle t, com, up \rangle$ and $t \neq \perp$. By the semantics, there is $(pc, \tau) \rightarrow (pc[t \mapsto stmt], \tau.act)$ for some $pc \in ctrl(\tau)$ with $pc(t) \xrightarrow{com} stmt$. Hence, the required $com \in next-com(\tau)$ follows by definition. ■

Proof C.21 (Lemma B.41). The direction from right to left follows by definition. So consider the direction from left to right. To that end, let $h_3 \in \mathcal{F}_O(h_1.h_2, a)$. Towards a contradicting, assume that $h_2.h_3 \notin \mathcal{F}_O(h_1, a)$ holds. We have $frees_{h_2} \subseteq \{a\}$ by assumption and $frees_{h_3} \subseteq \{a\}$ by definition. So, $h_1.h_2.h_3 \notin \mathcal{S}(\mathcal{O}_{SMR})$. By definition, however, this gives $h_3 \notin \mathcal{F}_O(h_1.h_2, a)$, contradicting the assumption. ■

Proof C.22 (Lemma B.42). To the contrary, let $\tau.act \in \mathcal{O}[[P]]_{Adr}^{Adr}$ be the shortest computation such that there is $a \in fresh_{\tau.act}$ with $a \in range(m_{\tau.act})$. By definition, there is $pexp \in PExp$ with $m_{\tau.act}(pexp) = a$. Moreover, $a \in fresh_\tau$. By minimality, we get $m_\tau(pexp) \neq a$. Hence, act must update $pexp$ to a . That is, act performs an allocation or a pointer assignment. In the former case, we have $act = \langle t, pexp := \text{malloc}, [p \mapsto a, \dots] \rangle$. Then, $a \notin fresh_{\tau.act}$ follows by definition. Since this contradicts the assumption, the case cannot apply. That is, act is of the form $act = \langle t, pexp := qexp, up \rangle$ with $m_\tau(qexp) = a$. This means $a \in range(m_\tau)$, contradicting the minimality of $\tau.act$. ■

Proof C.23 (Lemma B.43). Towards a contradiction, assume there is a shortest $\tau.act \in \mathcal{O}[[P]]_{Adr}^{Adr}$ such that there is some $a \in fresh_{\tau.act}$ with $a \in m_{\tau.act}(valid_{\tau.act}) \vee a.next \in valid_{\tau.act}$. Note that $\tau.act$ is indeed the shortest such computation since the claim holds for ϵ . By monotonicity, we have $a \in fresh_\tau$. By minimality of $\tau.act$ we have $a \notin m_\tau(valid_\tau)$ and $a.next \notin valid_\tau$. If $a.next \in valid_{\tau.act}$, then act is an assignment of the form $p.next := q$ with $m_\tau(p) = a$; note that an allocation of a could validate $a.next$ as well but would give $a \notin fresh_{\tau.act}$, contradicting the assumption. This means $a \in range(m_\tau)$. So we get $a \notin fresh_\tau$ from Lemma B.42. This contradicts $a \in fresh_\tau$ from above. Hence, the case does not apply and have $a.next \notin valid_{\tau.act}$. By assumption then, $a \in m_{\tau.act}(valid_{\tau.act})$. So there is some $pexp \in valid_{\tau.act}$ with $m_{\tau.act}(pexp) = a$. Since we already established $a \notin m_\tau(valid_\tau)$, we must have $pexp \notin valid_\tau$ or $m_\tau(pexp) \neq a$. Consider $pexp \notin valid_\tau$. That is, act validates $pexp$. To do so, act must be an assignment, an allocation, or an assumption. We do a case distinction.

◇ **Case 1:** $act = \langle t, pexp := qexp, up \rangle$

Then $qexp \in valid_\tau$ and $m_\tau(qexp) = a$. We obtain $a \in m_\tau(valid_\tau)$. This constitutes a contradiction, the case does not apply.

◇ **Case 2:** $act = \langle t, p := \text{malloc}, [p \mapsto a, a.next \mapsto seg, a.data \mapsto d] \rangle$

We obtain $a \notin fresh_{\tau.act}$ what contradicts the assumption. The case does not apply.

◇ **Case 3:** $act = \langle t, \text{assume } p = q, up \rangle$

Wlog. $pexp \equiv p$ and $q \in \text{valid}_\tau$ and $a = m_{\tau.act}(pexp) = m_\tau(pexp) = m_\tau(q)$ must hold. We then obtain $a \in m_\tau(\text{valid}_\tau)$. This constitutes a contradiction, the case does not apply.

So $pexp \in \text{valid}_\tau$ must hold and thus $m_\tau(pexp) \neq a$. That is, act updates $pexp$ to a (with $a \neq \text{seg}$). To do so, act must be an assignment or an allocation. We conclude a contradiction as before. ■

Proof C.24 (Lemma B.44). Towards a contradiction, assume $\text{fresh}_\tau \cap \text{freed}_\tau \neq \emptyset$. Let $a \in \text{fresh}_\tau$ and $a \in \text{freed}_\tau$. The latter means that τ is of the form $\tau = \tau_1.act.\tau_2$ with $act = \langle t, \text{free}(a), up \rangle$. Then, by definition, $a \notin \text{fresh}_{\tau_1.act}$. By monotonicity, this yields $a \notin \text{fresh}_\tau$. Since this contradicts the assumption, we must have $\text{fresh}_\tau \cap \text{freed}_\tau = \emptyset$ as required.

Towards a contradiction, assume $\text{fresh}_\tau \cap \text{retired}_\tau \neq \emptyset$. Let $a \in \text{fresh}_\tau$ and $a \in \text{retired}_\tau$. The latter means that τ is of the form $\tau = \tau_1.act.\tau_2$ with $act = \langle t, \text{in:retire}(p), up \rangle$ and $m_{\tau_1}(p) = a$. The contrapositive of Lemma B.42 gives $a \notin \text{fresh}_{\tau_1}$. By monotonicity, we get $a \notin \text{fresh}_\tau$. Since this contradicts the assumption, we must have $\text{fresh}_\tau \cap \text{retired}_\tau = \emptyset$ as required. ■

Proof C.25 (Lemma B.45). Let $a \in \text{Adr}$ and $\varphi = \{z_a \mapsto a\}$. The claim holds for ϵ . Towards a contradiction, assume there is a shortest $\tau.act \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$ such that $(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau.act)} (L_3, \varphi)$ and $a \notin \text{retired}_{\tau.act}$. If $\mathcal{H}(\tau.act) = \mathcal{H}(\tau)$, then we have $\text{retired}_{\tau.act} = \text{retired}_\tau$. This contradicts the minimality of $\tau.act$. So $\mathcal{H}(\tau.act)$ is of the form $\mathcal{H}(\tau.act) = h.evt$ with $h = \mathcal{H}(\tau)$.

◇ **Case 1:** $(L_2, \varphi) \xrightarrow{h} (L_1, \varphi)$

By definition of $\mathcal{O}_{\text{Base}}$, there is not step $(L_1, \varphi) \xrightarrow{evt} (L_3, \varphi)$. Hence, this case cannot apply.

◇ **Case 2:** $(L_2, \varphi) \xrightarrow{h} (L_2, \varphi)$

We must have $(L_2, \varphi) \xrightarrow{evt} (L_3, \varphi)$. This means evt is of the form $evt = \text{retire}(t, a)$ for some t . That is, $act = \langle t, \text{retire}(p), up \rangle$ with $m_\tau(p) = a$. Thus, $a \in \text{retired}_{\tau.act}$, contradicting the assumption.

◇ **Case 3:** $(L_2, \varphi) \xrightarrow{h} (L_3, \varphi)$

By minimality, we have $a \in \text{retired}_\tau$. To arrive at $a \notin \text{retired}_\tau$, we must have $evt = \text{free}(a)$. This yields $(L_3, \varphi) \xrightarrow{evt} (L_2, \varphi)$. So, $(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau.act)} (L_2, \varphi)$. This contradicts the assumption.

The above case distinction is complete and proves that $(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau)} (L_3, \varphi)$ implies $a \in \text{retired}_\tau$. Consider now the reverse direction. To that end, consider $\tau \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$ and $a \notin \text{retired}_\tau$. Using the contrapositive of the above, we get $(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau)} (l, \varphi)$ with $l \neq L_3$. We have $l \neq L_1$ as for otherwise $\tau \notin \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$. Hence, $l = L_2$. The remaining follows analogously. ■

Proof C.26 (Lemma B.46). Let $\tau.act \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$ with $act = \langle t, \text{free}(a), up \rangle$. Let $\varphi = \{z_a \mapsto a\}$. We have $\mathcal{H}(\tau.act) = h.\text{free}(a)$ with $h = \mathcal{H}(\tau)$. By definition, we have $h.\text{free}(a) \in \mathcal{S}(\mathcal{O})$. Since $\mathcal{O} = \mathcal{O}_{\text{Base}} \times \mathcal{O}_{\text{SMR}}$ by convention, we have $h.\text{free}(a) \in \mathcal{S}(\mathcal{O}_{\text{Base}})$. So $(L_2, \varphi) \xrightarrow{h} (L_3, \varphi)$ as for otherwise $\text{free}(a)$ would take $\mathcal{O}_{\text{Base}}$ to L_1 and thus give $\tau.act \notin \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$. Now, Lemma B.45 yields the desired $a \in \text{retired}_\tau$. ■

Proof C.27 (Lemma B.47). Let $\tau.act \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$ with $act = \langle t, \text{com}, up \rangle$.

◇ **Case 1:** $com \equiv exp := exp'$

We show $m_{\tau.act}(valid_{\tau.act}) \subseteq m_{\tau}(valid_{\tau})$ and $adr(m_{\tau.act}|_{valid_{\tau.act}}) \subseteq adr(m_{\tau}|_{valid_{\tau}})$.

◇ **Case 1.1:** $com \equiv p := q.next$

Let $a = m_{\tau}(q)$. By definition, $a \neq seg$. Let $b = m_{\tau}(a.next)$. Hence, $up = [p \mapsto b]$. If $a.next \in valid_{\tau}$ and $q \in valid_{\tau}$ we have:

$$\begin{aligned} m_{\tau.act}(valid_{\tau.act}) &= m_{\tau.act}(valid_{\tau} \setminus \{p\}) \cup \{m_{\tau.act}(p)\} = m_{\tau}(valid_{\tau} \setminus \{p\}) \cup \{b\} \\ &\subseteq m_{\tau}(valid_{\tau}) \cup \{b\} = m_{\tau}(valid_{\tau}) \end{aligned}$$

where the last equality holds by $b = m_{\tau}(a.next) \in m_{\tau}(valid_{\tau})$. Otherwise, we get:

$$m_{\tau.act}(valid_{\tau.act}) = m_{\tau.act}(valid_{\tau} \setminus \{p\}) = m_{\tau}(valid_{\tau} \setminus \{p\}) \subseteq m_{\tau}(valid_{\tau}).$$

Altogether, we obtain $m_{\tau.act}(valid_{\tau.act}) \subseteq m_{\tau}(valid_{\tau})$ as required. Combining this with

$$valid_{\tau.act} \cap Adr \subseteq (valid_{\tau} \cup \{p\}) \cap Adr = valid_{\tau} \cap Adr$$

yields $adr(m_{\tau.act}|_{valid_{\tau.act}}) \subseteq adr(m_{\tau}|_{valid_{\tau}})$ by Lemma B.35.

◇ **Case 1.2:** $com \equiv p := q$ or $com \equiv p.next := q$

Analogous to the previous case.

◇ **Case 1.3:** $com \in \{u := op(\bar{u}), u := q.data, p.data := u\}$

By definition, $valid_{\tau.act} = valid_{\tau}$ and $m_{\tau}(pexp) = m_{\tau.act}(pexp)$ for all $pexp \in PExp$. So, we conclude $m_{\tau.act}(valid_{\tau.act}) = m_{\tau}(valid_{\tau})$ and $adr(m_{\tau.act}|_{valid_{\tau.act}}) = adr(m_{\tau}|_{valid_{\tau}})$.

◇ **Case 2:** $com \equiv p := malloc$

Let $a = m_{\tau.act}(p)$. The update is $up = [p \mapsto a, a.next \mapsto seg, a.data \mapsto d]$ for some d . We have $valid_{\tau.act} = valid_{\tau} \cup \{p, a.next\}$. So, $valid_{\tau.act} \cap Adr = (valid_{\tau} \cap Adr) \cup \{a\}$. We get:

$$\begin{aligned} m_{\tau.act}(valid_{\tau.act}) &= m_{\tau.act}(valid_{\tau} \setminus \{p, a.next\}) \cup m_{\tau.act}(\{p, a.next\}) \\ &= m_{\tau}(valid_{\tau} \setminus \{p, a.next\}) \cup \{a\} \subseteq m_{\tau}(valid_{\tau}) \cup \{a\}. \end{aligned}$$

Combining the above yields $adr(m_{\tau.act}|_{valid_{\tau.act}}) \subseteq adr(m_{\tau}|_{valid_{\tau}})$ by Lemma B.35.

◇ **Case 3:** $com \equiv free(a)$

We have $m_{\tau.act} = m_{\tau}$ and $valid_{\tau.act} \subseteq valid_{\tau}$. Hence, $m_{\tau.act}(valid_{\tau.act}) \subseteq m_{\tau}(valid_{\tau})$. Moreover, $valid_{\tau.act} \cap Adr \subseteq valid_{\tau} \cap Adr$. So $adr(m_{\tau.act}|_{valid_{\tau.act}}) \subseteq adr(m_{\tau}|_{valid_{\tau}})$ by Lemma B.35.

◇ **Case 4:** $com \equiv \text{env}(a)$

The update takes the form $up = [a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d]$ for some value d . By definition, we have $\text{valid}_{\tau.\text{act}} = \text{valid}_\tau$ and thus $\text{valid}_{\tau.\text{act}} \cap \text{Adr} = \text{valid}_\tau \cap \text{Adr}$. Moreover, we get:

$$\begin{aligned} m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}}) &\subseteq m_{\tau.\text{act}}(\text{valid}_\tau \setminus \{a.\text{next}\}) \cup m_{\tau.\text{act}}(\{\{a.\text{next}\}\}) \\ &= m_\tau(\text{valid}_\tau \setminus \{a.\text{next}\}) \cup \emptyset \subseteq m_\tau(\text{valid}_\tau). \end{aligned}$$

Combining the above yields the desired $\text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) \subseteq \text{adr}(m_\tau|_{\text{valid}_\tau})$ by Lemma B.35.

◇ **Case 5:** otherwise

We show $m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}}) = m_\tau(\text{valid}_\tau)$ and $\text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) = \text{adr}(m_\tau|_{\text{valid}_\tau})$.

◇ **Case 5.1:** $com \in \{\text{in:func}(\bar{v}), \text{re:func}, \text{skip}, \text{beginAtomic}, \text{endAtomic}, @\text{inv} \bullet\}$

By definition, we have $\text{valid}_{\tau.\text{act}} = \text{valid}_\tau$ and $m_{\tau.\text{act}} = m_\tau$. Hence, we conclude the desired $m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}}) = m_\tau(\text{valid}_\tau)$ and $\text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) = \text{adr}(m_\tau|_{\text{valid}_\tau})$.

◇ **Case 5.2:** $com \equiv \text{assume cond}$

We have $m_\tau(\text{valid}_\tau) = m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}})$. This follows from act validating only pointers that are equal to already valid pointers. Further, $\text{valid}_{\tau.\text{act}} \cap \text{Adr} = \text{valid}_\tau \cap \text{Adr}$ since act may only validate pointers. Hence, $\text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) = \text{adr}(m_\tau|_{\text{valid}_\tau})$ by Lemma B.35.

The above case distinction is complete and concludes the claim. ■

Proof C.28 (Lemma B.48). To the contrary, assume there is a shortest $\tau.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$ PRF such that there is $a \in \text{freed}_{\tau.\text{act}}$ with $a \in m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}}) \vee a.\text{next} \in \text{valid}_{\tau.\text{act}}$. Note that $\tau.\text{act}$ is indeed the shortest such computation since the claim is vacuously true for ϵ . First, consider the case where $a \in \text{freed}_\tau$. By minimality, $a \notin m_\tau(\text{valid}_\tau)$ and $a.\text{next} \notin \text{valid}_\tau$. If $a.\text{next} \in \text{valid}_{\tau.\text{act}}$ holds, then act must be an assignment of the form $p.\text{next} := q$ with $m_\tau(p) = a$. (Note that an allocation of a could validate $a.\text{next}$ as well but would give $a \notin \text{freed}_{\tau.\text{act}}$ and thus contradict the assumption.) Now, $p \notin \text{valid}_\tau$ must hold because $a \notin m_\tau(\text{valid}_\tau)$. Hence, act raises an unsafe access. This contradicts the assumption. So we must have $a \in m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}})$. Recall that we have $a \notin m_\tau(\text{valid}_\tau)$. Along the lines of Proof C.23 of Lemma B.43, this means that act must be an allocation of a . So, $a \notin \text{freed}_{\tau.\text{act}}$ follows by definition. As this contradicts the assumption, the case cannot apply. Altogether, we must have $a \notin \text{freed}_\tau$. Then, act must execute $\text{free}(a)$. We get $a.\text{next} \notin \text{valid}_{\tau.\text{act}}$ and $p.\text{exp} \notin \text{valid}_{\tau.\text{act}}$ for all $p.\text{exp}$ that satisfy $m_\tau(p.\text{exp}) = a$. That is, we obtain $a \notin m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}})$ and $a.\text{next} \notin \text{valid}_{\tau.\text{act}}$. This contradicts the assumption. ■

Proof C.29 (Lemma B.49). We show the following proposition:

$$\begin{aligned} \forall \tau, p.\text{exp}. \quad & \tau \text{ UAF} \wedge m_\tau(p.\text{exp}) = \text{seg} \wedge p.\text{exp} \notin \text{valid}_\tau \\ \implies & (p.\text{exp} \cap \text{Adr}) \cap (\text{fresh}_\tau \cup \text{freed}_\tau) \neq \emptyset \end{aligned}$$

which implies the lemma. To see this, let $\tau \text{ UAF}$ and $p.\text{exp} \in \text{VExp}(\tau)$ with $m_\tau(p.\text{exp}) = \text{seg}$. To the contrary, assume $p.\text{exp} \notin \text{valid}_\tau$. Then the above yields $(p.\text{exp} \cap \text{Adr}) \cap (\text{fresh}_\tau \cup \text{freed}_\tau) \neq \emptyset$. Hence, $p.\text{exp}$ must be of the form

$pexp \equiv a.next$ with $a \in fresh_\tau \cup freed_\tau$. Lemmas B.42 and B.48 give $a \notin m_\tau(valid_\tau)$. This means $a.next \notin VExp(\tau)$. Because this contradicts $pexp \in VExp(\tau)$, we must have $pexp \in valid_\tau$ as required.

We now show the above proposition. To the contrary, assume there is a shortest $\tau.act \in \mathcal{O}[P]_{Adr}^{Adr}$ UAF with $m_{\tau.act}(pexp) = seg$ and $pexp \notin valid_{\tau.act}$ and $(pexp \cap Adr) \cap (fresh_{\tau.act} \cup freed_{\tau.act}) = \emptyset$.

First, consider $pexp \in PVar$. Then, we obtain $(pexp \cap Adr) \cap (fresh_\tau \cup freed_\tau) = \emptyset$ because of $pexp \cap Adr = \emptyset$. By minimality of $\tau.act$, we must have $m_\tau(pexp) \neq seg$ or $pexp \in valid_\tau$. In the former case, act executes an assignment of the form $pexp := qexp$ with $m_\tau(qexp) = seg$. To arrive at $pexp \notin valid_{\tau.act}$, we must have $qexp \in PVar \setminus valid_\tau$ or $qexp \equiv q.next \wedge q \notin valid_\tau$ or $qexp \equiv q.next \wedge m_\tau(q).next \notin valid_\tau$. By minimality and $PVar \cap Adr = \emptyset$, the first case cannot apply. The second case cannot apply because it raises an unsafe access, contradicting the assumption that $\tau.act$ is UAF. So, $qexp \equiv q.next$ with $m_\tau(q) = a$ and $a.next \notin valid_\tau$. This means $qexp \cap Adr = \{a\}$. We get $a \in fresh_\tau \cup freed_\tau$ by minimality. Then, Lemmas B.43 and B.48 yield $a \notin m_\tau(valid_\tau)$. That is, $q \notin valid_\tau$. Consequently, act raises an unsafe access, contradicting the UAF assumption. Hence, the case cannot apply; we must have $m_\tau(pexp) = seg$ and $pexp \in valid_\tau$. In order for act to invalidate $pexp$, it must assign to $pexp$ from an invalid seg-holding expression (note that value seg cannot be the target of a free command). We conclude a contradiction to $\tau.act$ UAF as before. Altogether, $pexp \in PVar$ cannot hold. We must have $pexp \notin PVar$.

Consider now $pexp \notin PVar$. That is, $pexp \equiv a.next$ for some address $a \in Adr$. By definition, we get $pexp \cap Adr = \{a\}$. By minimality, $m_\tau(pexp) \neq seg$ or $pexp \in valid_\tau$ or $a \in fresh_\tau \cup freed_\tau$. In the latter case, act must allocate a to arrive at $(pexp \cap Adr) \cap (fresh_{\tau.act} \cup freed_{\tau.act}) = \emptyset$. This, however, validates $a.next$, contradicting $pexp \notin valid_{\tau.act}$. Hence, we have $a \notin fresh_\tau \cup freed_\tau$. If we have $pexp \in valid_\tau$, then act must invalidate $pexp$. To do this, act must execute $free(a)$ or $p.next := q$ with $m_\tau(p) = a$. In the former case, we get $a \in freed_{\tau.act}$. Hence, we arrive at $(pexp \cap Adr) \cap (fresh_{\tau.act} \cup freed_{\tau.act}) = \{a\}$, contradicting the assumption. That is, the case cannot apply so that act must be an assignment. To get $m_{\tau.act}(pexp) = seg$ and $pexp \notin valid_{\tau.act}$, we must have $m_\tau(q) = seg$ and $q \notin valid_\tau$. Since we have $q \in PVar$, $q \cap Adr = \emptyset$ follows. That is, $(q \cap Adr) \cap (fresh_\tau \cup freed_\tau) = \emptyset$. Altogether, this contradicts the minimality of $\tau.act$. ■

Proof C.30 (Lemma B.50). We show the following proposition:

$$\forall \tau, pexp. \quad \tau \text{ UAF} \wedge m_\tau(pexp) = \perp \implies \{pexp\} \cap Adr \not\subseteq m_\tau(valid_\tau)$$

which implies the lemma. To see this, consider τ UAF and $pexp \in VExp(\tau)$. To the contrary, assume $pexp \notin dom(m_\tau)$. That is, $m_\tau(pexp) = \perp$. The above then yields $pexp \cap Adr \not\subseteq m_\tau(valid_\tau)$. Because $\emptyset \subseteq m_\tau(valid_\tau)$ trivially holds, we must have $pexp \cap Adr = \{a\}$ for some address a with $a \notin m_\tau(valid_\tau)$. This means $pexp \equiv a.next$. Hence, $pexp \notin VExp(\tau)$ by definition. Because this contradicts the choice of $pexp$, we must have $pexp \in dom(m_\tau)$ as required.

We now establish the above proposition. To the contrary, assume a shortest $\tau.act \in \mathcal{O}[P]_{Adr}^{Adr}$ UAF such that there is some $pexp$ with $m_{\tau.act}(pexp) = \perp$ and $\{pexp\} \cap Adr \subseteq m_{\tau.act}(valid_{\tau.act})$. By minimality, $m_\tau(pexp) \neq \perp$ or $\{pexp\} \cap Adr \not\subseteq m_\tau(valid_\tau)$. First, consider $m_\tau(pexp) \neq \perp$. In order to arrive at $m_{\tau.act}(pexp) = \perp$, action act must execute an assignment $pexp := qexp$ with $m_\tau(qexp) = \perp$. If $qexp \in PVar$, then we have $\{qexp\} \cap Adr = \emptyset \subseteq$

$m_\tau(\text{valid}_\tau)$. As this contradicts minimality, $qexp$ must be of the form $qexp \equiv b.\text{next}$. By minimality, $b \notin m_\tau(\text{valid}_\tau)$. This means act raises an unsafe access. This contradicts the assumption of $\tau.act$ being UAF. Altogether, we get $m_\tau(pexp) = \perp$ and thus we must have $\{pexp\} \cap \text{Adr} \not\subseteq m_\tau(\text{valid}_\tau)$.

Consider the case $\{pexp\} \cap \text{Adr} \not\subseteq m_\tau(\text{valid}_\tau)$ now. Observe that $PVar \cap \text{Adr} = \emptyset \subseteq m_\tau(\text{valid}_\tau)$. So, $pexp$ must be $pexp \equiv a.\text{next}$. We get $\{pexp\} \cap \text{Adr} = \{a\}$ and thus $a \notin m_\tau(\text{valid}_\tau)$. Similarly, to arrive at $\{pexp\} \cap \text{Adr} \subseteq m_{\tau.act}(\text{valid}_{\tau.act})$ we must have $a \in m_{\tau.act}(\text{valid}_{\tau.act})$. That is, act produces a valid pointer expression referencing a without having such an expression at hand. Along the lines of Proof C.23 of Lemma B.43, act must perform an allocation of a to do that:

$$act = \langle t, p := \text{malloc}, [p \mapsto a, a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d] \rangle$$

for some d . However, this results in $m_{\tau.act}(pexp) = \text{seg} \neq \perp$, contradicting the assumption. \blacksquare

Proof C.31 (Lemma B.51). We proceed by induction over the structure of τ . In the base case, $\tau = \epsilon$. Then, the claim follows from $VExp(\epsilon) = PVar \subseteq \text{valid}_\epsilon$ by definition. For the induction step, consider some $\tau.act \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{UAF}}$ and assume we have already establish that $pexp \notin \text{valid}_\tau$ implies $m_\tau(pexp) \in \text{frees}_\tau$. Let $pexp \in VExp(\tau.act)$ with $pexp \notin \text{valid}_{\tau.act}$. We now establish that $m_{\tau.act}(pexp) \in \text{frees}_{\tau.act}$ holds. Let $act = \langle t, com, up \rangle$.

◇ **Case 1:** $com \equiv p := q.\text{next}$

Let $a = m_\tau(q)$. By definition, we have $a \neq \text{seg}$. Let $b = m_\tau(a.\text{next})$. So, $up = [p \mapsto b]$. First, consider the case $pexp \equiv p$. By definition, $pexp \in VExp(\tau)$. Further, $q \in \text{valid}_\tau$ since $\tau.act$ is UAF by assumption. So, we must have $a.\text{next} \notin \text{valid}_\tau$ to arrive at $pexp \notin \text{valid}_\tau$. By induction, we get $m_\tau(a.\text{next}) \in \text{frees}_\tau = \text{frees}_{\tau.act}$. Hence, $m_{\tau.act}(p) \in \text{frees}_{\tau.act}$ as required

Consider now $pexp \neq p$. Then, $pexp \notin \text{valid}_\tau$ by definition and. Moreover, Lemma B.47 yields $m_{\tau.act}(\text{valid}_{\tau.act}) \subseteq m_\tau(\text{valid}_\tau)$. Hence, we obtain $pexp \in VExp(\tau)$. By induction, we get $m_\tau(pexp) \in \text{frees}_\tau$. By definition, this means $m_{\tau.act}(pexp) \in \text{frees}_{\tau.act}$ as required.

◇ **Case 2:** $com \equiv p := q$ or $com \equiv p.\text{next} := q$

Analogous to the previous case.

◇ **Case 3:** $com \equiv u := \text{op}(\bar{u})$

By definition, $\text{frees}_\tau = \text{frees}_{\tau.act}$ and $\text{valid}_\tau = \text{valid}_{\tau.act}$. Moreover, $m_\tau(pexp) = m_{\tau.act}(pexp)$ for all $pexp \in PExp$. We obtain $VExp(\tau) = VExp(\tau.act)$. Hence, we conclude by induction.

◇ **Case 4:** $com \in \{u := q.\text{data}, p.\text{data} := u\}$

Analogous to the previous case.

◇ **Case 5:** $com \in \{\text{in:func}(\bar{v}), \text{re:func}, \text{skip}, \text{beginAtomic}, \text{endAtomic}, @\text{inv} \bullet\}$

Analogous to the previous case.

◇ **Case 6:** $com \equiv p := \text{malloc}$

Let $a = m_{\tau.act}(p)$. The update is $up = [p \mapsto a, a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d]$ for some d . We have $p, a.\text{next} \in \text{valid}_{\tau.act}$ by definition. So, $pexp \notin \{p, a.\text{next}\}$ and $pexp \notin \text{valid}_\tau$. Then, we obtain $m_{\tau.act}(pexp) = m_\tau(pexp)$.

Moreover, we get $m_{\tau.act}(valid_{\tau.act}) \subseteq m_{\tau}(valid_{\tau}) \cup \{a\}$ by Lemma B.47. Hence, $pexp \in VExp(\tau)$ because $pexp \neq a.next$. So by induction, we obtain the desired $m_{\tau.act}(pexp) = m_{\tau}(pexp) \in frees_{\tau} = frees_{\tau.act}$.

◇ **Case 7:** $com \equiv \text{assume } cond$

By definition, we have $frees_{\tau} = frees_{\tau.act}$ and $m_{\tau}(valid_{\tau}) = m_{\tau.act}valid_{\tau.act}$. The latter follows from the fact that act validates only pointers that are equal to already valid pointers. Hence, we obtain $pexp \in VExp(\tau)$. Then, we conclude $m_{\tau.act}(pexp) = m_{\tau}(pexp) \in frees_{\tau} = frees_{\tau.act}$ by induction.

◇ **Case 8:** $com \equiv \text{free}(a)$

We have $m_{\tau.act} = m_{\tau}$ and $valid_{\tau.act} \subseteq valid_{\tau}$ by definition. Further, we get that $m_{\tau}(qexp) = a$ implies $qexp \notin valid_{\tau.act}$ for all $qexp$. That is, we have $a \notin m_{\tau.act}(valid_{\tau.act}) \subseteq m_{\tau}(valid_{\tau})$. Hence, $pexp \in VExp(\tau) \setminus \{a.next\}$. Moreover, $frees_{\tau.act} = frees_{\tau} \cup \{a\}$. If $pexp \notin valid_{\tau}$, then we conclude by induction. Otherwise, we have $pexp \in valid_{\tau}$. That is, $pexp$ is invalidated by act . This means $pexp \equiv a.next$ or $m_{\tau}(pexp) = a$. Since we already showed that the former case cannot apply, we have $m_{\tau}(pexp) = a$. Then, $m_{\tau.act}(pexp) = a \in frees_{\tau.act}$.

◇ **Case 9:** $com \equiv \text{env}(a)$

The update takes the form $up = [a.next \mapsto \text{seg}, a.data \mapsto d]$ for some data value d . By definition, we have $valid_{\tau.act} = valid_{\tau}$ and so we get:

$$\begin{aligned} m_{\tau.act}(valid_{\tau.act}) &= m_{\tau.act}(valid_{\tau}) \subseteq m_{\tau.act}(valid_{\tau} \setminus \{a.next\}) \cup m_{\tau.act}(\{a.next\}) \\ &= m_{\tau}(valid_{\tau} \setminus \{a.next\}) \cup \emptyset \subseteq m_{\tau}(valid_{\tau}). \end{aligned}$$

This means $pexp \in VExp(\tau)$ and $pexp \notin valid_{\tau}$. By induction, $m_{\tau}(pexp) \in frees_{\tau} = frees_{\tau.act}$. Because $a \in fresh_{\tau} \cup freed_{\tau}$ according to the semantics, we get $a \notin m_{\tau}(valid_{\tau})$ by Lemmas B.43 and B.48. That is, $a.next \notin VExp(\tau)$ by definition. Thus, $pexp \neq a.next$. By the update, we get $m_{\tau}(pexp) = m_{\tau.act}(pexp)$. We arrive at the desired $m_{\tau.act}(pexp) \in frees_{\tau.act}$.

The above case distinction is complete and concludes the induction. ■

Proof C.32 (Lemma B.52). Let $pexp \in VExp(\tau) \setminus valid_{\tau}$. By Lemma B.51, $m_{\tau}(pexp) = a \in frees_{\tau}$ for some address a . Since $\tau \in \mathcal{O}[\![P]\!]_{Addr}^{\emptyset}$, address a remains free once it is freed, it cannot be reallocated. Hence, $m_{\tau}(pexp) \in freed_{\tau}$ follows as required. ■

Proof C.33 (Lemma B.53). Let $\tau \in \mathcal{O}[\![P]\!]_{Addr}^A$ UAF. Consider some $a \in Addr$ with $a \in adr(m_{\tau}|_{valid_{\tau}})$ and $a \in m_{\tau}(VExp(\tau) \setminus valid_{\tau})$. We show $a \in A$. The latter and Lemma B.51 yields $a \in frees_{\tau}$. The former and Lemma B.35 yields $a \in valid_{\tau} \cap Addr$ or $a \in m_{\tau}(valid_{\tau})$. Note that $a \in valid_{\tau} \cap Addr$ implies $a.next \in valid_{\tau}$. Hence, the contrapositive of Lemma B.48 gives $a \notin freed_{\tau}$. Altogether, this means that a has been freed and reallocated in τ . That is, $a \in A$ must hold. ■

Proof C.34 (Lemma B.54). Let $\tau \in \mathcal{O}[\![P]\!]_{Addr}^{Adr}$ UAF. Consider some $C \in CVar$ and $p \in PVar \setminus valid_{\tau}$. By Assumption A.9, we have $m_{\tau}(C) \notin frees_{\tau} \cup retired_{\tau}$. By definition, $p \in VExp(\tau) \setminus valid_{\tau}$. Then, Lemma B.51 yields $m_{\tau}(p) \in frees_{\tau}$. Hence, $m_{\tau}(C) \neq m_{\tau}(p)$ must hold as required. ■

Proof C.35 (Lemma B.55). Let \mathcal{O} support elision. Let $\tau \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$ with $\mathcal{H}(\tau) = h$. By the semantics, we have $h \in \mathcal{S}(\mathcal{O}_{Base})$ and can thus invoke Definition 7.14 with h . Let $a, b, c \in Adr$ be addresses with $a \neq c \neq b$. We show $\mathcal{F}_{\mathcal{O}}(h, c) = \mathcal{F}_{\mathcal{O}}(h[a/b], c)$. By assumption, \mathcal{O} supports elision. By Assumption 5.6, $\mathcal{O} = \mathcal{O}_{Base} \times \mathcal{O}_{SMR}$. Hence, $\mathcal{F}_{\mathcal{O}_{SMR}}(h, c) = \mathcal{F}_{\mathcal{O}_{SMR}}(h[a/b], c)$ by Property (i) of Definition 7.14. We get $\mathcal{F}_{\mathcal{O}_{Base}}(h, c) = \mathcal{F}_{\mathcal{O}_{Base}}(h[a/b], c)$ by definition of \mathcal{O}_{Base} . To see this, consider some $h' \in \mathcal{F}_{\mathcal{O}_{Base}}(h, c)$. By definition, $frees_{h'} \subseteq \{c\}$ and $h.h' \in \mathcal{S}(\mathcal{O}_{Base})$. Note that only events of the form $free(\bullet)$ can reach an accepting location in \mathcal{O}_{Base} . Because all $free$ events in h' are of the form $free(c)$ and $(free(c))[a/b] = free(c)$, we derive that h' takes \mathcal{O}_{Base} to an accepting location (after h) if and only if $h'[a/b]$ does. This concludes the required $\mathcal{F}_{\mathcal{O}_{Base}}(h, c) = \mathcal{F}_{\mathcal{O}_{Base}}(h[a/b], c)$.

Now, consider $h' \in \mathcal{F}_{\mathcal{O}}(h, c)$. By definition, we have $frees_{h'} \subseteq \{c\}$ and $h.h' \in \mathcal{S}(\mathcal{O})$. The former gives $frees_{h'[a/b]} \subseteq \{c\}$. Because of $\mathcal{O} = \mathcal{O}_{Base} \times \mathcal{O}_{SMR}$, we have both $h.h' \in \mathcal{S}(\mathcal{O}_{Base})$ and $h.h' \in \mathcal{S}(\mathcal{O}_{SMR})$. Then, $h' \in \mathcal{F}_{\mathcal{O}_{Base}}(h, c)$ and $h' \in \mathcal{F}_{\mathcal{O}_{SMR}}(h, c)$. So $h' \in \mathcal{F}_{\mathcal{O}_{Base}}(h[a/b], c)$ and $h' \in \mathcal{F}_{\mathcal{O}_{SMR}}(h[a/b], c)$. That is, $h.h'[a/b] \in \mathcal{S}(\mathcal{O}_{Base})$ as well as $h.h'[a/b] \in \mathcal{S}(\mathcal{O}_{SMR})$. Finally, we arrive at $h.h'[a/b] \in \mathcal{S}(\mathcal{O})$. That is, we have $h'[a/b] \in \mathcal{F}_{\mathcal{O}}(h, c)$. Altogether, we have established $\mathcal{F}_{\mathcal{O}}(h, c) \subseteq \mathcal{F}_{\mathcal{O}}(h[a/b], c)$. The reverse direction follows analogously. ■

Proof C.36 (Lemma B.56). Let \mathcal{O} support elision. Let $\tau \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$ with $\mathcal{H}(\tau) = h$. By the semantics, we have $h \in \mathcal{S}(\mathcal{O}_{Base})$ and can thus invoke Definition 7.14 with h . Let $a \neq b$. Assume we have $h.free(a) \in \mathcal{S}(\mathcal{O})$. By Assumption 5.6, $\mathcal{O} = \mathcal{O}_{Base} \times \mathcal{O}_{SMR}$. Elision support, Property (iii) of Definition 7.14, gives $\mathcal{F}_{\mathcal{O}_{SMR}}(h.free(a), b) = \mathcal{F}_{\mathcal{O}_{SMR}}(h, b)$. To conclude, it remains to show $\mathcal{F}_{\mathcal{O}_{Base}}(h.free(a), b) = \mathcal{F}_{\mathcal{O}_{Base}}(h, b)$. The inclusion $\mathcal{F}_{\mathcal{O}_{Base}}(h.free(a), b) \subseteq \mathcal{F}_{\mathcal{O}_{Base}}(h, b)$ follows from the definition of \mathcal{O}_{Base} . The interesting case is the reverse inclusion. Consider some $h' \in \mathcal{F}_{\mathcal{O}_{Base}}(h, b)$. By definition, $frees_{h'} \subseteq \{b\}$ and $h.h' \in \mathcal{S}(\mathcal{O}_{Base})$. The latter means that for all addresses c and all steps $(L_2, \varphi) \xrightarrow{h} (l_1, \varphi) \xrightarrow{h'} (l_2, \varphi)$ of \mathcal{O} with $\varphi = \{z_a \mapsto c\}$ we have that l_2 is not accepting. Observe that $h.free(a) \in \mathcal{S}(\mathcal{O})$ means l_1 is not accepting as well. If we have $c \neq a$, then $(l_1, \varphi) \xrightarrow{free(a)} (l_1, \varphi)$. So we get $(L_2, \varphi) \xrightarrow{h.free(a).h'} (l_2, \varphi)$ by Assumption 5.2. Otherwise, there is a program step $(l_1, \varphi) \xrightarrow{free(a)} (l'_1, \varphi)$ for some l'_1 . Note that location l'_1 is not accepting because we have $h.free(a) \in \mathcal{S}(\mathcal{O})$. Now, let l'_2 be any location with $(l'_1, \varphi) \xrightarrow{h'} (l'_2, \varphi)$. By definition, we know that only a $free(a)$ event can take \mathcal{O}_{Base} to an accepting location for φ . That is, l'_2 is not accepting because of $a \notin frees_{h'}$. Altogether, we obtain $h.free(a).h' \in \mathcal{S}(\mathcal{O}_{Base})$ in any case. That is, we arrive at $h' \in \mathcal{F}_{\mathcal{O}_{Base}}(h.free(a), b)$. We conclude the desired $\mathcal{F}_{\mathcal{O}_{Base}}(h.free(a), b) = \mathcal{F}_{\mathcal{O}_{Base}}(h, b)$. ■

Proof C.37 (Lemma B.57). Let $\tau, \sigma \in \mathcal{O}[\![P]\!]_{Adr}^{Adr}$. Let $h = \mathcal{H}(\tau)$ and let $h' = \mathcal{H}(\sigma)$. Let $a, b \in Adr$. By assumption, we have $\mathcal{F}_{\mathcal{O}}(h, a) \subseteq \mathcal{F}_{\mathcal{O}}(h', a)$ as well as $b \notin retired_h$ and $b \in fresh_{h'}$. We now establish that $\mathcal{F}_{\mathcal{O}}(h, b) \subseteq \mathcal{F}_{\mathcal{O}}(h', b)$ holds. By Assumption 5.6, $\mathcal{O} = \mathcal{O}_{Base} \times \mathcal{O}_{SMR}$. By elision support, Property (ii) of Definition 7.14, we have $\mathcal{F}_{\mathcal{O}_{SMR}}(h, b) \subseteq \mathcal{F}_{\mathcal{O}_{SMR}}(h', b)$. In order to conclude, it remains to show $\mathcal{F}_{\mathcal{O}_{Base}}(h, b) \subseteq \mathcal{F}_{\mathcal{O}_{Base}}(h', b)$. Consider $\hat{h} \in \mathcal{F}_{\mathcal{O}_{Base}}(h, b)$. By definition, we have:

$$frees_{\hat{h}} \subseteq \{b\} \quad \text{and} \quad h.\hat{h} \in \mathcal{S}(\mathcal{O}_{Base}).$$

Let c be some address and let $\varphi = \{z_a \mapsto c\}$. Consider the following steps:

$$(L_2, \varphi) \xrightarrow{h} (l_1, \varphi) \xrightarrow{\hat{h}} (l_2, \varphi) \quad \text{and} \quad (L_2, \varphi) \xrightarrow{h'} (l'_1, \varphi) \xrightarrow{\hat{h}} (l'_2, \varphi).$$

From $h.\hat{h} \in \mathcal{S}(\mathcal{O}_{Base})$ follows that l_2 is not accepting. We show that l'_2 is not accepting either.

◇ **Case 1:** $b = c$

We first show that $l_1 = l'_1$ holds. Since $b \in fresh_{h'}$, we have $l'_1 = L_2$. To that end, observe that l_1 is not accepting. This follows from l_2 being not accepting together with $c \notin frees_{\hat{h}}$ and the fact that only $free(c)$ can take \mathcal{O}_{Base} to an accepting location for φ . That is, we have $l_1 \in \{L_2, L_3\}$. Recall that $b \notin retired_h$. This means that, every $in:retire(\bullet, b)$ event is followed by an $free(b)$ event. Hence, $l_1 \neq L_3$ according to the transitions in \mathcal{O}_{Base} . We arrive at the desired $l_1 = L_2 = l'_1$. By Assumption 5.2, we get $l_2 = l'_2$. Hence, l'_2 is not accepting.

◇ **Case 2:** $b \neq c$

Assume for a moment that l'_1 is not accepting. Then l_2 is not accepting because $c \notin frees_{\hat{h}}$ and only $free(c)$ can take \mathcal{O}_{Base} to an accepting location for φ . So it remains to show that location l'_1 is not accepting indeed. To that end, recall $h' = \mathcal{H}(\sigma)$ and $\sigma \in \mathcal{O}[[P]]_{Adr}^{\emptyset}$. As argued before, \mathcal{O}_{Base} reaches its final location L_1 only upon events evt of the form $evt = free(\bullet)$ and cannot leave L_1 afterwards. So, if $h' \notin \mathcal{S}(\mathcal{O}_{Base})$ was true, then there is decomposition of σ that takes the form $\sigma = \sigma_1.act.\sigma_2$ with $act = \langle t, free(\bullet), up \rangle$ and $\mathcal{H}(\sigma_1.act) \notin \mathcal{S}(\mathcal{O}_{Base})$. The latter means $\mathcal{H}(\sigma_1.act) \notin \mathcal{S}(\mathcal{O})$. This contradicts the enabledness of $\sigma_1.act$ and thus contradicts $\sigma \in \mathcal{O}[[P]]_{Adr}^{Adr}$. Hence, $h' \in \mathcal{S}(\mathcal{O}_{Base})$. That is, l'_1 is not accepting, as required.

That l'_2 is not accepting for any φ means $h'.\hat{h} \in \mathcal{S}(\mathcal{O}_{Base})$. Hence, $\hat{h} \in \mathcal{F}_{\mathcal{O}_{Base}}(h', b)$ as required. ■

Proof C.38 (Lemma B.58). Follows from the fact that $swap_{adr}$ is a bijection. ■

Proof C.39 (Lemma B.59). To the contrary, assume there is a shortest history $h.evt$ such that there are $h_1.evt_1 \neq h_2.evt_2$ with $swap_{adr}(h_1.evt_1) = h.evt = swap_{adr}(h_2.evt_2)$. Note that $h_1.evt_1$ and $h_2.evt_2$ and $h.evt$ all must have the same length. Also note that $h.evt$ is indeed a shortest such history because the claim holds for ϵ . Let evt be of the form $evt \equiv in:func(t, \bar{v})$. By choice, we have $swap_{hist}(evt_i) = evt$. That is, evt_i must be of the form $evt_i \equiv in:func(t, \bar{v}_i)$ with $\bar{v} = swap_{adr}(\bar{v}_i)$. So $\bar{v}_1 = swap_{adr}^{-1}(\bar{v}) = \bar{v}_2$. Hence, $evt_1 = evt_2$. Moreover, $h_1 = h_2$ by minimality of $h.evt$. Altogether, we obtain $h_1.evt_1 = h_2.evt_2$ which contradicts the assumption. The remaining cases follow analogously. ■

Proof C.40 (Lemma B.60). If $h \in H$, then $h' \in swap_{hist}(H)$ for $swap_{hist}(h) = h'$ by definition. For the reverse direction, we know that there is some $\hat{h} \in H$ with $swap_{hist}(\hat{h}) = h'$. Lemma B.59 yields $\hat{h} = h$. This yields $h \in H$ as desired. ■

Proof C.41 (Lemma B.61). Consider $a' \in Adr$. Since $swap_{adr}$ is a bijection, there is $a \in Adr$ such that $a' = swap_{adr}(a)$ and $a' \notin swap_{adr}(Adr \setminus \{a\})$. Hence, $a \notin A_1 \implies a' \notin swap_{adr}(A_1)$. Moreover, $a \in A_1 \implies a' \in swap_{adr}(A_1)$ by choice of a . So $a' \in swap_{adr}(A_1) \iff a \in A_1$. Similarly, $a' \in swap_{adr}(A_2) \iff a \in A_2$. Thus, $a' \in swap_{adr}(A_1) \otimes swap_{adr}(A_2) \iff a \in A_1 \otimes A_2$. With the same arguments we get $a \in A_1 \otimes A_2 \iff a' \in swap_{adr}(A_1 \otimes A_2)$, concluding the first equivalence.

As before, we have

$$a'.next \in swap_{exp}(B_1) \iff a.next \in B_1 \quad \text{and} \quad a'.next \in swap_{exp}(B_2) \iff a.next \in B_2$$

and derive $a'.next \in swap_{exp}(B_1) \otimes swap_{exp}(B_2) \iff a.next \in B_1 \otimes B_2$. And with the same arguments we get $a.next \in B_1 \otimes B_2 \iff a'.next \in swap_{exp}(B_1 \otimes B_2)$. This concludes the second equivalence. ■

Consider now some history h' . Since $swap_{adr}$ is a bijection, there is h with $swap_{hist}(h) = h'$. Then, Lemma B.60 yields $h' \in swap_{hist}(C_1) \iff h \in C_1$. Similarly, $h' \in swap_{hist}(C_2) \iff h \in C_2$. We arrive at $h' \in swap_{hist}(C_1) \otimes swap_{hist}(C_2) \iff h \in C_1 \otimes C_2$. Finally, Lemma B.60 yields that $h \in C_1 \otimes C_2 \iff h' \in swap_{hist}(C_1 \otimes C_2)$ holds. This concludes the third equivalence. ■

Proof C.42 (Lemma B.62). Consider some event $evt \equiv in:func(t, \bar{v})$. Then we have:

$$\begin{aligned} swap_{hist}^{-1}(swap_{hist}(evt)) &= swap_{hist}^{-1}(swap_{hist}(in:func(t, \bar{v}))) = swap_{hist}^{-1}(in:func(t, swap_{adr}(\bar{v}))) \\ &= in:func(t, swap_{adr}^{-1}(swap_{adr}(\bar{v}))) = in:func(t, \bar{v}). \end{aligned}$$

Analogously, for $free(a)$ and $re:func(t)$. The overall claim follows then from inductively applying the above to h . In the base case, we have $h = \epsilon$ and $swap_{hist}^{-1}(swap_{hist}(\epsilon)) = \epsilon$ by definition. For $h.evt$ one has

$$\begin{aligned} swap_{hist}^{-1}(swap_{hist}(h.evt)) &= swap_{hist}^{-1}(swap_{hist}(h).swap_{hist}(evt)) \\ &= swap_{hist}^{-1}(swap_{hist}(h)).swap_{hist}^{-1}(swap_{hist}(evt)) = h.evt \end{aligned}$$

where the last equality is due to induction (for h) and due to the above reasoning (for evt). ■

Proof C.43 (Lemma B.63). We first show the following auxiliary:

$$(l, \varphi) \xrightarrow{h} (l', \varphi) \iff (l, swap_{adr} \circ \varphi) \xrightarrow{swap_{hist}(h)} (l', swap_{adr} \circ \varphi). \quad (8)$$

To that end, let $(l, \varphi) \xrightarrow{evt} (l', \varphi)$ be some SMR automaton step and let $\bar{z} = dom(\varphi)$ be the SMR automaton variables. We show that also

$$(l, \varphi') \xrightarrow{swap_{hist}(evt)} (l', \varphi') \quad \text{with} \quad \varphi' = swap_{adr} \circ \varphi$$

is an SMR automaton step. We focus on events of the form $evt \equiv in:func(t, \bar{v})$; the remaining cases follow analogously. By the definition, there is a transition

$$l \xrightarrow{func(\bar{r}), g} l' \quad \text{such that} \quad g[\bar{r} \mapsto \bar{v}, \bar{z} \mapsto \varphi(\bar{z})] \models \text{true}.$$

Now, turn to $swap_{hist}(evt)$. It takes the form $swap_{hist}(evt) \equiv in:func(\bar{w})$ with $\bar{w} = swap_{adr}(\bar{v})$. Hence, the above transition matches. We show that it is also enabled. To that end, we need show that the guard evaluates to true, i.e., $g[\bar{r} \mapsto \bar{w}, \bar{z} \mapsto \varphi'(\bar{z})] \models \text{true}$. Intuitively, this holds because guards are composed of (in)equalities which are *stable* under the bijection $swap_{adr}$. Formally, g is equivalent to:

$$g \models \bigvee_i \bigwedge_j var_{i,j,1} \triangleq var_{i,j,2} \quad \text{with} \quad var_{i,j,k} \in \{\bar{r}, \bar{z}\} \quad \text{and} \quad \triangleq \in \{=, \neq\}.$$

Hence, we have to show

$$(var \triangleq var')[\bar{r} \mapsto \bar{v}, \bar{z} \mapsto \varphi(\bar{z})] \models \text{true} \iff (var \triangleq var')[\bar{r} \mapsto \bar{w}, \bar{z} \mapsto \varphi'(\bar{z})] \models \text{true}$$

for every i, j and $var = var_{i,j,1}$ and $var' = var_{i,j,2}$. We conclude this as follows:

$$\begin{aligned}
& (var \triangleq var')[\bar{r} \mapsto \bar{v}, \bar{z} \mapsto \varphi(\bar{z})] \models \text{true} \\
\iff & var[\bar{r} \mapsto \bar{v}, \bar{z} \mapsto \varphi(\bar{z})] \triangleq var'[\bar{r} \mapsto \bar{v}, \bar{z} \mapsto \varphi(\bar{z})] \\
\iff & swap_{adr}(var[\bar{r} \mapsto \bar{v}, \bar{z} \mapsto \varphi(\bar{z})]) \triangleq swap_{adr}(var'[\bar{r} \mapsto \bar{v}, \bar{z} \mapsto \varphi(\bar{z})]) \\
\iff & var[\bar{r} \mapsto swap_{adr}(\bar{v}), \bar{z} \mapsto swap_{adr}(\varphi(\bar{z}))] \triangleq var'[\bar{r} \mapsto swap_{adr}(\bar{v}), \bar{z} \mapsto swap_{adr}(\varphi(\bar{z}))] \\
\iff & var[\bar{r} \mapsto \bar{w}, \bar{z} \mapsto \varphi'(\bar{z})] \triangleq var'[\bar{r} \mapsto \bar{w}, \bar{z} \mapsto \varphi'(\bar{z})] \\
\iff & (var \triangleq var')[\bar{r} \mapsto \bar{w}, \bar{z} \mapsto \varphi'(\bar{z})] \models \text{true}
\end{aligned}$$

where the second equivalence holds because $swap_{adr}$ is a bijections, and the third equivalence holds because var and var' are either contained in \bar{r} or \bar{z} by the definition of SMR automata. We conclude (8).

Altogether, the overall implication

$$(l, \varphi) \xrightarrow{h} (l', \varphi) \implies (l, swap_{adr} \circ \varphi) \xrightarrow{swap_{hist}(h)} (l', swap_{adr} \circ \varphi)$$

follows by applying (8) inductively to every event/step of history h . For the reverse direction, we use Lemma B.58. More precisely, we apply (8) to $(l, swap_{adr} \circ \varphi) \xrightarrow{swap_{hist}(h)} (l', swap_{adr} \circ \varphi)$ using the address mapping $swap_{adr}^{-1}$. This yields:

$$\begin{aligned}
& (l, swap_{adr} \circ \varphi) \xrightarrow{swap_{hist}(h)} (l', swap_{adr} \circ \varphi) \\
\implies & (l, swap_{adr}^{-1} \circ swap_{adr} \circ \varphi) \xrightarrow{swap_{hist}^{-1}(swap_{hist}(h))} (l', swap_{adr}^{-1} \circ swap_{adr} \circ \varphi) .
\end{aligned}$$

Then, Lemma B.62 together with $swap_{adr}^{-1} \circ swap_{adr} = id$ gives:

$$\begin{aligned}
& (l, swap_{adr}^{-1} \circ swap_{adr} \circ \varphi) \xrightarrow{swap_{hist}^{-1}(swap_{hist}(h))} (l', swap_{adr}^{-1} \circ swap_{adr} \circ \varphi) \\
\implies & (l, swap_{adr} \circ \varphi) \xrightarrow{swap_{hist}(h)} (l', swap_{adr} \circ \varphi) .
\end{aligned}$$

This concludes the claim. ■

Proof C.44 (Lemma B.64). Let $a \in Adr$. We conclude as follows:

$$\begin{aligned}
& h' \in \mathcal{F}_{\mathcal{O}}(h, a) \\
\iff & h.h' \in \mathcal{S}(\mathcal{O}) \wedge \text{frees}(h') \subseteq \{a\} \\
\iff & swap_{hist}(h).swap_{hist}(h') \in \mathcal{S}(\mathcal{O}) \wedge \text{frees}(swap_{hist}(h')) \subseteq \{swap_{adr}(a)\} \\
\iff & swap_{hist}(h') \in \mathcal{F}_{\mathcal{O}}(swap_{hist}(h), swap_{adr}(a))
\end{aligned}$$

where the second equivalence holds because of Lemma B.63. ■

Proof C.45 (Theorem B.65). We proceed by induction over the structure of computations. In the base case, we have $\tau = \epsilon$. By definition, choosing $\sigma = \epsilon$ satisfies the claim. For the induction step, consider some $\tau.act \in \mathcal{O}[[P]]_{Adr}^A$ and assume that we have already constructed σ with:

- (P1) $\sigma \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{swap}_{\text{adr}}(A)}$
- (P2) $\forall pexp \in PExp. m_\sigma(\text{swap}_{\text{exp}}(pexp)) = \text{swap}_{\text{adr}}(m_\tau(pexp))$
- (P3) $\forall dexp \in DExp. m_\sigma(\text{swap}_{\text{exp}}(dexp)) = m_\tau(dexp)$
- (P4) $\text{valid}_\sigma = \text{swap}_{\text{exp}}(\text{valid}_\tau)$
- (P5) $\text{freed}_\sigma = \text{swap}_{\text{adr}}(\text{freed}_\tau)$
- (P6) $\text{fresh}_\sigma = \text{swap}_{\text{adr}}(\text{fresh}_\tau)$
- (P7) $\mathcal{H}(\sigma) = \text{swap}_{\text{hist}}(\mathcal{H}(\tau))$
- (P8) $\text{ctrl}(\sigma) = \text{ctrl}(\tau)$

Let $\text{act} = \langle t, \text{com}, \text{up} \rangle$. We show that there is $\text{act}' = \langle t, \text{com}', \text{up}' \rangle$ such that $\sigma.\text{act}'$ satisfies the claim, that is, satisfies the following:

- (G1) $\sigma.\text{act}' \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{swap}_{\text{adr}}(A)}$
- (G2) $\forall pexp \in PExp. m_{\sigma.\text{act}'}(\text{swap}_{\text{exp}}(pexp)) = \text{swap}_{\text{adr}}(m_{\tau.\text{act}}(pexp))$
- (G3) $\forall dexp \in DExp. m_{\sigma.\text{act}'}(\text{swap}_{\text{exp}}(dexp)) = m_{\tau.\text{act}}(dexp)$
- (G4) $\text{valid}_{\sigma.\text{act}'} = \text{swap}_{\text{exp}}(\text{valid}_{\tau.\text{act}})$
- (G5) $\text{freed}_{\sigma.\text{act}'} = \text{swap}_{\text{adr}}(\text{freed}_{\tau.\text{act}})$
- (G6) $\text{fresh}_{\sigma.\text{act}'} = \text{swap}_{\text{adr}}(\text{fresh}_{\tau.\text{act}})$
- (G7) $\mathcal{H}(\sigma.\text{act}') = \text{swap}_{\text{hist}}(\mathcal{H}(\tau.\text{act}))$
- (G8) $\text{ctrl}(\sigma.\text{act}') = \text{ctrl}(\tau.\text{act})$

We choose $\text{com}' = \text{com}$ if $\text{com} \not\equiv \text{free}(a)$ and $\text{com} \not\equiv \text{env}(a)$. Otherwise, we replace the address that is used: $\text{com}' = \text{free}(\text{swap}_{\text{adr}}(a))$ or $\text{com}' = \text{env}(\text{swap}_{\text{adr}}(a))$. Thus, (G8) will follow from (P8) together with the semantics; we will not comment on this property hereafter. For (G1) we will only argue that act' is enabled after σ . This, together with (P1) yields the desired property. We do a case distinction on the executed command com .

◇ **Case 1:** $\text{com} \equiv p := q$

Let $a = m_\tau(b)$. The update is $\text{up} = [p \mapsto a]$. Choose $\text{up}' = [p \mapsto \text{swap}_{\text{adr}}(a)]$. Then, (G5) to (G7) follow from (P5) to (P7) because the fresh/freed addresses are not changed and no event is emitted.

◇ *Ad (G1).* We have to show that $\text{swap}_{\text{adr}}(a) = m_\sigma(q)$ holds. By the choice of a , this boils down to showing $m_\sigma(q) = \text{swap}_{\text{adr}}(m_\tau(q))$. This follows from (P2) with $\text{swap}_{\text{exp}}(q) = q$.

◇ *Ad (G2).* For p the claim follows by choice of up' . So consider $pexp \in PExp \setminus \{p\}$. Then, we conclude as follows:

$$\begin{aligned} m_{\sigma.\text{act}'}(\text{swap}_{\text{exp}}(pexp)) &= m_\sigma(\text{swap}_{\text{exp}}(pexp)) = \text{swap}_{\text{adr}}(m_\tau(pexp)) \\ &= \text{swap}_{\text{adr}}(m_{\tau.\text{act}}(pexp)) \end{aligned}$$

where the first equality holds because only p is updated by up' and $\text{swap}_{\text{exp}}(pexp) \neq p$, the second equality holds by (P2), and the third equality holds because only p is updated by up .

◇ *Ad (G3).* Neither up nor up' update data expressions. We get $m_{\tau.\text{act}}(dexp) = m_\tau(dexp)$ and $m_{\sigma.\text{act}'}(dexp) = m_\sigma(dexp)$ for all $dexp \in DExp$. Hence, the claim follows from (P3).

- ◇ *Ad (G4).* By $\text{swap}_{\text{exp}}(q) = q$ we have $q \in \text{valid}_\tau \iff q \in \text{valid}_\sigma$. We conclude by (P4), Lemma B.61, and the definition of validity and swap_{exp} as follows:

$$\begin{aligned} \text{valid}_{\sigma.\text{act}'} &= \text{valid}_\sigma \otimes \{p\} = \text{swap}_{\text{exp}}(\text{valid}_\tau) \otimes \text{swap}_{\text{exp}}(\{p\}) \\ &= \text{swap}_{\text{exp}}(\text{valid}_\tau \otimes \{p\}) = \text{swap}_{\text{exp}}(\text{valid}_{\tau.\text{act}}) \end{aligned}$$

with $\otimes := \cup$ if $q \in \text{valid}_\tau$ and $\otimes := \setminus$ otherwise.

- ◇ **Case 2:** $\text{com} \equiv p = q.\text{next}$

Let $a = m_\tau(q)$. By definition, $a \neq \text{seg}$. Let $b = m_\tau(a.\text{next})$. So the update is $up = [p \mapsto b]$. We choose $up' = [p \mapsto \text{swap}_{\text{adr}}(b)]$. Then, (G5) to (G7) follow immediately from (P5) to (P7) because the fresh/freed addresses are not changed and no event is emitted.

- ◇ *Ad (G1).* Let $a' = m_\sigma(q)$ and $b' = m_\sigma(a'.\text{next})$. We have to show $b' = \text{swap}_{\text{adr}}(b)$. First, note that by (P2) we have:

$$a' = m_\sigma(q) = m_\sigma(\text{swap}_{\text{exp}}(q)) = \text{swap}_{\text{adr}}(m_\tau(q)) = \text{swap}_{\text{adr}}(a).$$

Observe that this means $a' \neq \text{seg}$. Then, we conclude as follows:

$$\begin{aligned} b' &= m_\sigma(a'.\text{next}) = m_\sigma(\text{swap}_{\text{adr}}(a).\text{next}) = m_\sigma(\text{swap}_{\text{exp}}(a.\text{next})) \\ &= \text{swap}_{\text{adr}}(m_\tau(a.\text{next})) = \text{swap}_{\text{adr}}(b). \end{aligned}$$

- ◇ *Ad (G2).* For p the claim follows by choice of up' . For $p\text{exp} \in P\text{Exp} \setminus \{p\}$ the claim follows from (P2) together with up and up' not modifying the valuation of $p\text{exp}$, as in the previous case.
- ◇ *Ad (G3).* Neither up nor up' update data expressions. Hence, the claim follows by (P3).
- ◇ *Ad (G4).* By definition, $\text{swap}_{\text{exp}}(q) = q$. So (P4) yields $q \in \text{valid}_\tau \iff q \in \text{valid}_\sigma$. Since (G1) from above gives $a' = \text{swap}_{\text{adr}}(a)$, we have $\text{swap}_{\text{exp}}(a.\text{next}) = a'.\text{next}$. As a consequence, (P4) yields $a.\text{next} \in \text{valid}_\tau \iff a'.\text{next} \in \text{valid}_\sigma$. This means we have $p \in \text{valid}_{\tau.\text{act}} \iff p \in \text{valid}_{\sigma.\text{act}'}$ because $m_\tau(q) \in \text{Adr}$. Together with (P4) this concludes the claim because act/act' affects only the validity of p .

- ◇ **Case 3:** $\text{com} \equiv p.\text{next} = q$

Analogous to the previous case.

- ◇ **Case 4:** $\text{com} \equiv u = \text{op}(u_1, \dots, u_n)$

The update is $up = [u \mapsto d]$ with $d = \text{op}(m_\tau(u_1), \dots, m_\tau(u_n))$. Choose $up' = up$. Since the pointer expression valuations, the validity, and the fresh/freed address are not altered as well as no event is emitted by act/act' , (G2) and (G4) to (G7) follow immediately from (P2) and (P4) to (P7).

- ◇ *Ad (G1).* By (P3) we have $m_\sigma(u_i) = m_\tau(u_i)$. Hence, act' is enabled after σ .
- ◇ *Ad (G3).* We have $m_{\sigma.\text{act}'}(\text{swap}_{\text{exp}}(u)) = m_{\sigma.\text{act}'}(u) = d = m_{\tau.\text{act}}(u)$. And because no other data expressions are updated by up/up' , the claim follows from (P3).

◇ **Case 5:** $com \equiv u = q.data$

Let $a = m_\tau(q)$. By definition, $a \neq seg$. Let $d = m_\tau(a.data)$. So $up = [u \mapsto d]$. Choose $up' = up$. Since the pointer expression valuations, the validity, and the fresh/freed address are not altered as well as no event is emitted by act/act' , (G2) and (G4) to (G7) follow from (P2) and (P4) to (P7)

◇ *Ad (G1).* Let $a' = m_\sigma(q)$ and $d' = m_\sigma(a'.data)$. For enabledness of act' , we have to show $d' = d$. To see this, first note that by (P2) we have:

$$a' = m_\sigma(q) = m_\sigma(\text{swap}_{exp}(q)) = \text{swap}_{adr}(m_\tau(q)) = \text{swap}_{adr}(a) .$$

This means $a' \neq seg$. Together with (P3) we conclude as follows:

$$d' = m_\sigma(a'.data) = m_\sigma(\text{swap}_{exp}(a.data)) = m_\tau(a.data) = d .$$

◇ *Ad (G3).* We have $m_{\sigma.act'}(u) = d = m_{\tau.act}(u) = m_{\tau.act}(\text{swap}_{exp}(u))$. And because no other data expressions are updated by up/up' , the claim follows from (P3).

◇ **Case 6:** $com \equiv p.data = u$

Analogous to the previous case.

◇ **Case 7:** $com \equiv p := \text{malloc}$

Let $a = m_{\tau.act}(p)$. The update is $up = [p \mapsto a, a.next \mapsto seg, a.data \mapsto d]$ for some d . By definition, we have $a \in \text{fresh}_\tau \cup (\text{freed}_\tau \cap A)$. Choose up' as follows

$$up' = [p \mapsto \text{swap}_{adr}(a), \text{swap}_{adr}(a).next \mapsto seg, \text{swap}_{adr}(a).data \mapsto d] .$$

Then, (G7) follows immediately from (P7) because no event is emitted.

◇ *Ad (G1).* We have to show that $\text{swap}_{adr}(a) \in \text{fresh}_\sigma \cup (\text{freed}_\sigma \cap \text{swap}_{adr}(A))$. By (P5) and (P6) and Lemma B.61 we have:

$$\begin{aligned} & \text{fresh}_\sigma \cup (\text{freed}_\sigma \cap \text{swap}_{adr}(A)) \\ &= \text{swap}_{adr}(\text{fresh}_\tau) \cup (\text{swap}_{adr}(\text{freed}_\tau) \cap \text{swap}_{adr}(A)) \\ &= \text{swap}_{adr}(\text{fresh}_\tau \cup (\text{freed}_\tau \cap A)) . \end{aligned}$$

Then, $a \in \text{fresh}_\tau \cup (\text{freed}_\tau \cap A)$ yields $\text{swap}_{adr}(a) \in \text{fresh}_\sigma \cup (\text{freed}_\sigma \cap \text{swap}_{adr}(A))$.

◇ *Ad (G2).* We have

$$\begin{aligned} m_{\sigma.act'}(\text{swap}_{exp}(p)) &= m_{\sigma.act'}(p) = \text{swap}_{adr}(a) = \text{swap}_{adr}(m_{\tau.act}(p)) \\ \text{and } m_{\sigma.act'}(\text{swap}_{exp}(a.next)) &= m_{\sigma.act'}(\text{swap}_{adr}(a).next) = seg \\ &= \text{swap}_{adr}(seg) = \text{swap}_{adr}(m_{\tau.act}(a.next)) . \end{aligned}$$

Since no other pointer expressions are updated, the claim follows from (P2).

◇ *Ad (G3).* We have:

$$m_{\sigma.act'}(swap_{exp}(a.data)) = m_{\sigma.act'}(swap_{adr}(a).data) = d = m_{\tau.act}(a.data) .$$

Since no other data expression is updated, the follows from (P3).

◇ *Ad (G4).* We conclude using (P4) and Lemma B.61 as follows:

$$\begin{aligned} valid_{\sigma.act'} &= valid_{\sigma} \cup \{ p, swap_{adr}(a).next \} = swap_{exp}(valid_{\tau}) \cup swap_{exp}(\{ p, a.next \}) \\ &= swap_{exp}(valid_{\tau} \cup \{ p, a.next \}) = swap_{exp}(valid_{\tau.act}) . \end{aligned}$$

◇ *Ad (G5).* We conclude using (P5) and Lemma B.61 as follows:

$$\begin{aligned} freed_{\sigma.act'} &= freed_{\sigma} \setminus swap_{adr}(a) = swap_{adr}(freed_{\tau}) \setminus swap_{adr}(a) \\ &= swap_{adr}(freed_{\tau} \setminus a) = swap_{adr}(freed_{\tau.act}) . \end{aligned}$$

◇ *Ad (G6).* We conclude using (P6) and Lemma B.61 as follows:

$$\begin{aligned} fresh_{\sigma.act'} &= fresh_{\sigma} \setminus swap_{adr}(a) = swap_{adr}(fresh_{\tau}) \setminus swap_{adr}(a) \\ &= swap_{adr}(fresh_{\tau} \setminus a) = swap_{adr}(fresh_{\tau.act}) . \end{aligned}$$

◇ **Case 8:** $com \equiv free(a)$

The update is $up = \emptyset$. Choose $com' = free(swap_{adr}(a))$ and $up' = \emptyset$. Then, (G2) and (G3) follow from (P2) and (P3) because $m_{\tau} = m_{\tau.act}$ and $m_{\sigma} = m_{\sigma.act'}$.

◇ *Ad (G1).* By the semantics, $free(a) \in \mathcal{F}_{\mathcal{O}}(\tau, a)$. By (P7) together with Lemma B.64 we get $free(swap_{adr}(a)) \in \mathcal{F}_{\mathcal{O}}(\sigma, swap_{adr}(a))$. Hence, act' is enabled after σ .

◇ *Ad (G4).* Consider $pexp' \in PExp$. Since $swap_{adr}$ is a bijection, there is some $pexp \in PExp$ such that $swap_{exp}(pexp) = pexp'$. First, we get:

$$\begin{aligned} pexp' \in valid_{\sigma} &\iff swap_{exp}(pexp) \in valid_{\sigma} \\ &\iff swap_{exp}(pexp) \in swap_{exp}(valid_{\tau}) \iff pexp \in valid_{\tau} \end{aligned}$$

where the second equivalence is due to (P4) and the third equivalence holds since $swap_{adr}$ is a bijection. Second, we have:

$$\begin{aligned} m_{\sigma}(pexp') \neq swap_{adr}(a) &\iff m_{\sigma}(swap_{exp}(pexp)) \neq swap_{adr}(a) \\ &\iff swap_{adr}(m_{\tau}(pexp)) \neq swap_{adr}(a) \iff m_{\tau}(pexp) \neq a \end{aligned}$$

where the second equivalence is due to (P2) and the third equivalence holds since swap_{adr} is a bijection. Last, we have:

$$\begin{aligned} \text{pexp}' \cap \text{Adr} \neq \{ \text{swap}_{\text{adr}}(a) \} &\iff \text{swap}_{\text{exp}}(\text{pexp}) \cap \text{Adr} \neq \{ \text{swap}_{\text{adr}}(a) \} \\ &\iff \text{pexp} \cap \text{Adr} \neq \{ a \} \end{aligned}$$

Altogether, this gives:

$$\begin{aligned} \text{pexp}' &\in \text{valid}_{\sigma.\text{act}'} \\ \iff \text{pexp}' &\in \text{valid}_{\sigma} \wedge m_{\sigma}(\text{pexp}') \neq \text{swap}_{\text{adr}}(a) \wedge \text{pexp}' \cap \text{Adr} \neq \{ \text{swap}_{\text{adr}}(a) \} \\ \iff \text{pexp} &\in \text{valid}_{\tau} \wedge m_{\tau}(\text{pexp}) \neq a \wedge \text{pexp} \cap \text{Adr} \neq \{ a \} \\ \iff \text{pexp} &\in \text{valid}_{\tau.\text{act}} \\ \iff \text{swap}_{\text{exp}}(\text{pexp}) &\in \text{swap}_{\text{exp}}(\text{valid}_{\tau.\text{act}}) \\ \iff \text{pexp}' &\in \text{swap}_{\text{exp}}(\text{valid}_{\tau.\text{act}}) \end{aligned}$$

where the last but first equivalence holds because swap_{adr} is a bijection.

◇ *Ad (G5).* We conclude using (P5) and Lemma B.61:

$$\begin{aligned} \text{freed}_{\sigma.\text{act}'} &= \text{freed}_{\sigma} \cup \{ \text{swap}_{\text{adr}}(a) \} = \text{swap}_{\text{adr}}(\text{freed}_{\tau}) \cup \{ \text{swap}_{\text{adr}}(a) \} \\ &= \text{swap}_{\text{adr}}(\text{freed}_{\tau} \cup \{ a \}) = \text{swap}_{\text{adr}}(\text{freed}_{\tau.\text{act}}) . \end{aligned}$$

◇ *Ad (G6).* We conclude using (P6) and Lemma B.61:

$$\begin{aligned} \text{fresh}_{\sigma.\text{act}'} &= \text{fresh}_{\sigma} \setminus \{ \text{swap}_{\text{adr}}(a) \} = \text{swap}_{\text{adr}}(\text{fresh}_{\tau}) \setminus \{ \text{swap}_{\text{adr}}(a) \} \\ &= \text{swap}_{\text{adr}}(\text{fresh}_{\tau} \setminus \{ a \}) = \text{swap}_{\text{adr}}(\text{fresh}_{\tau.\text{act}}) . \end{aligned}$$

◇ *Ad (G7).* We conclude using (P7)

$$\begin{aligned} \mathcal{H}(\sigma.\text{act}') &= \mathcal{H}(\sigma).\text{free}(\text{swap}_{\text{adr}}(a)) = \text{swap}_{\text{hist}}(\mathcal{H}(\tau)).\text{swap}_{\text{hist}}(\text{free}(a)) \\ &= \text{swap}_{\text{hist}}(\mathcal{H}(\tau).\text{free}(a)) = \text{swap}_{\text{hist}}(\mathcal{H}(\tau.\text{act})) . \end{aligned}$$

◇ **Case 9:** $\text{com} \equiv \text{assume } \text{cond}$

The update is $\text{up} = \emptyset$. Choose $\text{up}' = \emptyset$. Since the memory and the fresh/freed address are not altered as well as no event is emitted by act/act' , (G2), (G3) and (G5) to (G7) follow immediately from (P2), (P3) and (P5) to (P7).

◇ *Ad (G1).* First, consider the case where cond is a condition over data variables u, u' . By (P3) we have $m_{\sigma}(u) = m_{\tau}(u)$ and $m_{\sigma}(u') = m_{\tau}(u')$. Hence, act' is enabled after σ . Now, consider the case where cond is a condition over pointer variables p, q . We arrive at $m_{\sigma}(p) = \text{swap}_{\text{adr}}(m_{\tau}(p))$ and $m_{\sigma}(q) = \text{swap}_{\text{adr}}(m_{\tau}(q))$. So we get $m_{\sigma}(p) = m_{\sigma}(q)$ iff $m_{\tau}(p) = m_{\tau}(q)$. Hence, act' is enabled after σ .

◇ *Ad (G4)*. Note that we have:

$$\begin{aligned}
& \text{valid}_{\sigma.act'} \neq \text{valid}_{\sigma} \\
\iff & \text{cond} \equiv p = q \wedge \{p, q\} \cap \text{valid}_{\sigma} \neq \emptyset \wedge \{p, q\} \not\subseteq \text{valid}_{\sigma} \\
\iff & \text{cond} \equiv p = q \wedge \{p, q\} \cap \text{swap}_{exp}(\text{valid}_{\tau}) \neq \emptyset \wedge \{p, q\} \not\subseteq \text{swap}_{exp}(\text{valid}_{\tau}) \\
\iff & \text{cond} \equiv p = q \wedge \{p, q\} \cap \text{valid}_{\tau} \neq \emptyset \wedge \{p, q\} \not\subseteq \text{valid}_{\tau} \\
\iff & \text{valid}_{\tau.act} \neq \text{valid}_{\tau}
\end{aligned}$$

where the second equivalence is by (P4). Consider the case $\text{valid}_{\sigma.act'} \neq \text{valid}_{\sigma}$. So we conclude using (P4) and Lemma B.61:

$$\begin{aligned}
\text{valid}_{\sigma.act'} &= \text{valid}_{\sigma} \cup \{p, q\} = \text{swap}_{exp}(\text{valid}_{\tau}) \cup \text{swap}_{exp}(\{p, q\}) \\
&= \text{swap}_{exp}(\text{valid}_{\tau} \cup \{p, q\}) = \text{swap}_{exp}(\text{valid}_{\tau.act}).
\end{aligned}$$

In all other cases, we have $\text{valid}_{\tau.act} = \text{valid}_{\tau}$ and $\text{valid}_{\sigma.act'} = \text{valid}_{\sigma}$. Then, the claim follows immediately from (P4).

◇ **Case 10:** $\text{com} \equiv \text{in:func}(r_1, \dots, r_n)$

The update is $up = \emptyset$ and we choose $up' = \emptyset$. Since the memory, the validity, and the fresh/freed address are not altered, (G2) to (G6) follow immediately from (P2) to (P6).

◇ *Ad (G1)*. Consider some $i \in \{1, \dots, n\}$. By definition, we have $m_{\tau}(p_i) \in \text{Adr} \cup \text{Dom}$. Hence, we get $\text{swap}_{adr}(m_{\tau}(p_i)) \in \text{Adr} \cup \text{Dom}$. Then (P2) yields $m_{\sigma}(p_i) \in \text{Adr} \cup \text{Dom}$. So act is enabled.

◇ *Ad (G7)*. We have $m_{\tau}(r_i), m_{\sigma}(r_i) \in \text{Adr} \cup \text{Dom}$ as observed for (G1) above. Let act emit the event $\text{in:func}(t, v_1, \dots, v_n)$ with $v_i = m_{\tau}(r_i)$. By (P2) and (P3), $m_{\sigma}(r_i) = \text{swap}_{adr}(v_i)$. So, act' emits the event $\text{in:func}(t, \text{swap}_{adr}(v_1), \dots, \text{swap}_{adr}(v_n))$. We conclude by (P7):

$$\begin{aligned}
\mathcal{H}(\sigma.act') &= \mathcal{H}(\sigma).\text{in:func}(t, \text{swap}_{adr}(v_1), \dots, \text{swap}_{adr}(v_n)) \\
&= \text{swap}_{hist}(\mathcal{H}(\tau)).\text{swap}_{hist}(\text{in:func}(t, v_1, \dots, v_n)) \\
&= \text{swap}_{hist}(\mathcal{H}(\tau).\text{in:func}(t, v_1, \dots, v_n)) = \text{swap}_{hist}(\mathcal{H}(\tau.act)).
\end{aligned}$$

◇ **Case 11:** $\text{com} \equiv \text{re}$

Follows analogously to the previous case.

◇ **Case 12:** $\text{com} \equiv \text{env}(a)$

We have $up = [a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d]$ for some d . By definition, $a \in \text{fresh}_{\tau} \cup \text{freed}_{\tau}$. Choose $\text{com}' = \text{env}(\text{swap}_{adr}(a))$ and $[\text{swap}_{adr}(a).\text{next} \mapsto \text{seg}, \text{swap}_{adr}(a).\text{data} \mapsto d]$. Then, (G4) to (G7) follow immediately from (P4) to (P7).

◇ *Ad (G1)*. We have to show that $\text{swap}_{adr}(a) \in \text{fresh}_{\sigma} \cup \text{freed}_{\sigma}$. This follows immediately from (P5) and (P6) together with $a \in \text{fresh}_{\tau} \cup \text{freed}_{\tau}$.

◇ *Ad (G2).* We have:

$$\begin{aligned} m_{\sigma.act'}(swap_{exp}(a.next)) &= m_{\sigma.act'}(swap_{adr}(a).next) = \perp \\ &= m_{\tau.act}(a.next) = swap_{adr}(m_{\tau.act}(a.next)) . \end{aligned}$$

For all other $pexp \in PExp \setminus a.next$ we conclude with (P2) as follows:

$$\begin{aligned} m_{\sigma.act'}(swap_{exp}(pexp)) &= m_{\sigma}(swap_{exp}(pexp)) = swap_{adr}(m_{\tau}(pexp)) \\ &= swap_{adr}(m_{\tau.act}(pexp)) . \end{aligned}$$

◇ *Ad (G3).* We have:

$$m_{\sigma.act'}(swap_{exp}(a.data)) = m_{\sigma.act'}(swap_{adr}(a).data) = d = m_{\tau.act}(a.data) .$$

For all other $dexp \in DExp \setminus \{a.data\}$ we conclude with (P3) as follows:

$$m_{\sigma.act'}(swap_{exp}(dexp)) = m_{\sigma}(swap_{exp}(dexp)) = m_{\tau}(dexp) = m_{\tau.act}(dexp) .$$

◇ **Case 13:** $com \in \{\text{skip}, \text{beginAtomic}, \text{endAtomic}, @inv \bullet\}$

Since act does not affect the computation, (G1) to (G8) follows immediately from (P1) to (P8).

The case distinction is complete and thus concludes the claim. ■

Proof C.46 (Lemma B.66). Let \mathcal{O} support elision. Let $b \in fresh_{\tau} \setminus A$. Let $swap_{adr} : Adr \rightarrow Adr$ be the address mapping defined by $swap_{adr}(a) = b$, $swap_{adr}(b) = a$, and $swap_{adr}(c) = c$ such that $a \neq c \neq b$. Theorem B.65 yields $\sigma \in \mathcal{O}[\![P]\!]_{Adr}^{swap_{adr}(A)}$ with

- $m_{\sigma} \circ swap_{exp} = swap_{adr} \circ m_{\tau}$,
- $\mathcal{H}(\sigma) = swap_{hist}(\mathcal{H}(\tau))$,
- $valid_{\sigma} = swap_{exp}(valid_{\tau})$,
- $fresh_{\sigma} = swap_{adr}(fresh_{\tau})$,
- $freed_{\sigma} = swap_{adr}(freed_{\tau})$, and
- $ctrl(\sigma) = ctrl(\tau)$.

We show that σ satisfies the claim. First, note that $\text{swap}_{adr}(A) = A$ by $a, b \notin A$. Consequently, we have $\sigma \in \mathcal{O}[[P]]_{Adr}^A$. We first show the following auxiliaries:

$$a.\text{next}, b.\text{next} \notin \text{valid}_\tau \quad (9)$$

$$f = f^{-1} \text{ for } f \in \{ \text{swap}_{adr}, \text{swap}_{exp}, \text{swap}_{hist} \} \quad (10)$$

$$\forall pexp \in \text{valid}_\tau. pexp \equiv \text{swap}_{exp}(pexp) \quad (11)$$

$$\forall pexp \in \text{valid}_\tau. m_\tau(pexp) = \text{swap}_{adr}(m_\tau(pexp)) \quad (12)$$

$$\forall c \in m_\tau(\text{valid}_\tau). c.\text{data} \equiv \text{swap}_{exp}(c.\text{data}) \quad (13)$$

$$\forall c \in \text{Adr} \setminus \{a, b\}. \mathcal{F}_\mathcal{O}(\tau, c) \subseteq \mathcal{F}_\mathcal{O}(\sigma, c) \quad (14)$$

$$\forall c \in A. c \in \text{retired}_\tau \iff c \in \text{retired}_\sigma \quad (15)$$

- ◇ *Ad (9).* Holds by $a \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$, and by $b \in \text{fresh}_\tau$ together with Lemma B.48.
- ◇ *Ad (10).* Holds by choice of swap_{adr} .
- ◇ *Ad (11).* Holds by (9) and the definition of swap_{adr} and its induced swap_{exp} .
- ◇ *Ad (12).* Holds by $a \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$ giving $a \notin m_\tau(\text{valid}_\tau)$ by Lemma B.35, and $b \in \text{fresh}_\tau$ giving $b \notin \text{range}(m_\tau)$ by Lemma B.42 and thus $b \notin m_\tau(\text{valid}_\tau)$.
- ◇ *Ad (13).* From $c \in m_\tau(\text{valid}_\tau)$ we get some $pexp \in \text{valid}_\tau$ with $m_\tau(pexp) = c$. Thus, (12) yields the following:
 $c = m_\tau(pexp) = \text{swap}_{adr}(m_\tau(pexp)) = \text{swap}_{adr}(c)$.
- ◇ *Ad (14).* Let $c \in \text{Adr} \setminus \{a, c\}$. Then,

$$\mathcal{F}_\mathcal{O}(\tau, c) = \mathcal{F}_\mathcal{O}(\mathcal{H}(\tau), c) = \mathcal{F}_\mathcal{O}(\text{swap}_{hist}(\mathcal{H}(\tau)), c) = \mathcal{F}_\mathcal{O}(\mathcal{H}(\sigma), c) = \mathcal{F}_\mathcal{O}(\sigma, c)$$

where the second equality holds by Lemma B.55 and the third equality holds by the properties given by Theorem B.65 listed above.

- ◇ *Ad (15).* Let $c \in A$. We have $a \neq c \neq b$ and thus $\text{swap}_{adr}(c) = c$. To the contrary, assume that we have $c \in \text{retired}_\tau$ but $c \notin \text{retired}_\sigma$. The former means that τ is of the form $\tau = \tau_1.act.\tau_2$ with $act = \langle t, com, up \rangle$ and $com \equiv \text{in}:\text{retired}_p$ and $m_{\tau_1}(p) = c$. Let $\mathcal{H}(\tau) = h_1.evt.h_2$ such that we have $\mathcal{H}(\tau_1) = h_1$. Then, $c \notin \text{frees}_{h_2}$ since $c \in \text{retired}_\tau$. By the construction of σ in Proof C.45 of Theorem B.65, we know that σ is of the form $\sigma = \sigma_1.act.\sigma_2$. Moreover, we have $\mathcal{H}(\sigma) = \text{swap}_{hist}(\mathcal{H}(\tau))$. Hence, we get $\text{swap}_{hist}(\mathcal{H}(\sigma)) = \mathcal{H}(\tau)$. This, in turn, means that $c \notin \text{swap}_{hist}^{-1}(\text{frees}_{h_2})$. By $\text{swap}_{adr}(c) = c$, we obtain that $c \notin \text{frees}_{h_2}$ holds. Further, $\text{swap}_{hist}^{-1}(\text{in}:\text{retire}(t, c)) = \text{in}:\text{retire}(t, c)$ by $\text{swap}_{adr}(c) = c$. Hence, we arrive at $c \in \text{retired}_\sigma$. The reverse direction follows analogously.

◇ *Ad* $\tau \sim \sigma$. We already have $ctrl(\sigma) = ctrl(\tau)$. We get:

$$\begin{aligned}
& dom(m_\sigma|_{valid_\sigma}) \\
&= valid_\sigma \cup DVar \cup \{ c.data \mid c \in m_\sigma(valid_\sigma) \} \\
&= swap_{exp}(valid_\tau) \cup DVar \cup \{ c.data \mid c \in swap_{adr}(m_\tau(swap_{exp}^{-1}(valid_\tau))) \} \\
&= swap_{exp}(valid_\tau) \cup DVar \cup \{ c.data \mid c \in swap_{adr}(m_\tau(swap_{exp}(valid_\tau))) \} \\
&= valid_\tau \cup DVar \cup \{ c.data \mid c \in swap_{adr}(m_\tau(valid_\tau)) \} \\
&= valid_\tau \cup DVar \cup \{ c.data \mid c \in m_\tau(valid_\tau) \} = dom(m_\tau|_{valid_\tau})
\end{aligned}$$

where the first equality is by definition, the second by the properties of σ , the third by (10), the fourth by (11), the fifth by (12), and the last by definition. Then, we get for all pointer expression $pexp \in dom(m_\sigma|_{valid_\sigma}) \cap PExp$:

$$m_\sigma(pexp) = swap_{adr}(m_\tau(swap_{exp}^{-1}(pexp))) = swap_{adr}(m_\tau(pexp)) = m_\tau(pexp)$$

using the properties of σ , the fact that $pexp \in valid_\tau$ must hold, and (10) to (12). Moreover, we get for $dexp \in dom(m_\sigma|_{valid_\sigma}) \cap DExp$:

$$m_\sigma(dexp) = m_\tau(swap_{exp}^{-1}(dexp)) = m_\tau(dexp) .$$

by (13) because $c.data \in dom(m_\sigma|_{valid_\sigma}) = dom(m_\tau|_{valid_\tau})$ implies $c \in m_\tau(valid_\tau)$. Altogether, this yields the desired $m_\tau|_{valid_\tau} = m_\sigma|_{valid_\sigma}$.

◇ *Ad* $\tau \preceq_A \sigma$. Let $c \in A$. We show $\tau \preceq_c \sigma$. We have $a \neq c \neq b$ and thus $swap_{adr}(c) = c$. So using (10) we get:

$$\begin{aligned}
c \in fresh_\sigma \cup freed_\sigma &\iff c \in swap_{adr}(fresh_\tau) \cup swap_{adr}(freed_\tau) \\
&\iff c \in swap_{adr}(fresh_\tau \cup freed_\tau) \\
&\iff swap_{adr}(c) \in swap_{adr}(swap_{adr}(fresh_\tau \cup freed_\tau)) \\
&\iff c \in fresh_\tau \cup freed_\tau .
\end{aligned}$$

Moreover, we have by (15):

$$c \in retired_\tau \iff c \in retired_\sigma .$$

From (14) we get $\mathcal{F}_O(\tau, c) \subseteq \mathcal{F}_O(\sigma, c)$. Then, we have for $p \in PVar$:

$$\begin{aligned}
c = m_\sigma(p) &\iff c = swap_{adr}(m_\tau(swap_{exp}(p))) \\
&\iff swap_{adr}(c) = swap_{adr}(swap_{adr}(m_\tau(p))) \\
&\iff c = m_\tau(p) .
\end{aligned}$$

Now, consider $c' \in m_\tau(\text{valid}_\tau)$. We have:

$$\begin{aligned} c = m_\sigma(c'.\text{next}) &\iff c = \text{swap}_{\text{adr}}(m_\tau(\text{swap}_{\text{exp}}(c'.\text{next}))) \\ &\iff \text{swap}_{\text{adr}}(c) = \text{swap}_{\text{adr}}(\text{swap}_{\text{adr}}(m_\tau(\text{swap}_{\text{exp}}(c'.\text{next})))) \\ &\iff c = m_\tau(\text{swap}_{\text{exp}}(c'.\text{next})). \end{aligned}$$

In order to conclude, it remains to show that $\text{swap}_{\text{exp}}(c'.\text{next}) = c.\text{next}$. To that end, it suffices to show that $c' \neq a$ and $c' \neq b$. The former follows from $a \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$ together with Lemma B.35. The latter follows from $b \in \text{fresh}_\tau$ together with Lemma B.43.

- ◇ *Ad $\tau < \sigma$.* Follows from (14) with $a, b \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$ by Lemmas B.35 and B.43.
- ◇ *Ad $a \in \text{fresh}_\sigma$.* We have $b \in \text{fresh}_\tau$. So $a = \text{swap}_{\text{adr}}(b) \in \text{swap}_{\text{adr}}(\text{fresh}_\tau) = \text{fresh}_\sigma$.
- ◇ *Ad $\text{retired}_\tau \subseteq \text{retired}_\sigma \cup \{a\}$.* Along the lines of (15), we get $\text{retired}_\sigma = \text{swap}_{\text{adr}}(\text{retired}_\tau)$. Let $c \in \text{retired}_\tau$. If $a \neq c \neq b$, we immediately get $c \in \text{retired}_\sigma$. If $a = c$, then $c \in \{a\}$ holds. Otherwise, $b = c$. The case cannot apply since $b \in \text{fresh}_\tau$ gives $b \notin \text{retired}_\tau$ which contradicts the choice of $c \in \text{retired}_\tau$.
- ◇ *Ad Memory Implication.* Let $\text{exp}, \text{exp}' \in \text{VExp}(\tau)$ with $m_\tau(\text{exp}) \neq m_\tau(\text{exp}')$. If $\text{exp} \in \text{PVar}$, then $\text{swap}_{\text{exp}}(\text{exp}) = \text{exp}$. Otherwise, $\text{exp} \equiv c.\text{next}$ with $c \in m_\tau(\text{valid}_\tau)$. By assumption together with Lemma B.35, $c \neq a$. And by Lemma B.43, $c \neq b$. So $\text{swap}_{\text{exp}}(\text{exp}) = \text{exp}$ must hold. Similarly, we get $\text{swap}_{\text{exp}}(\text{exp}') = \text{exp}'$. Then, we have:

$$\begin{aligned} m_\sigma(\text{exp}) &= \text{swap}_{\text{adr}}(m_\tau(\text{swap}_{\text{exp}}(\text{exp}))) = \text{swap}_{\text{adr}}(m_\tau(\text{exp})) \\ \text{and } m_\sigma(\text{exp}') &= \text{swap}_{\text{adr}}(m_\tau(\text{swap}_{\text{exp}}(\text{exp}'))) = \text{swap}_{\text{adr}}(m_\tau(\text{exp}')). \end{aligned}$$

Since swap_{adr} is a bijection, $m_\tau(\text{exp}) \neq m_\tau(\text{exp}')$ implies the desired $m_\sigma(\text{exp}) \neq m_\sigma(\text{exp}')$.

- ◇ *Ad SMR automaton implication.* Assume $a \notin \text{fresh}_\tau$. Let $c \in \text{fresh}_\sigma \setminus \{a\}$. From $a \notin \text{fresh}_\tau$ we get $\text{swap}_{\text{adr}}(a) \notin \text{swap}_{\text{adr}}(\text{fresh}_\tau)$. Hence, $b \notin \text{fresh}_\sigma$ follows from the properties of σ . That is, $c \in \text{Adr} \setminus \{a, b\}$. So (14) yields the desired $\mathcal{F}_\mathcal{O}(\tau, c) \subseteq \mathcal{F}_\mathcal{O}(\sigma, c)$.

This concludes the claim. ■

Proof C.47 (Lemma B.67). Let $\tau.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$ and $\sigma \in \mathcal{O}[\![P]\!]_{\text{Adr}}^A$ with $\tau \sim \sigma$, $\tau \preceq_A \sigma$, and $\tau <_A \sigma$. Unrolling the premise gives:

- (P1) $\text{ctrl}(\tau) = \text{ctrl}(\sigma)$
- (P2) $m_\tau|_{\text{valid}_\tau} = m_\sigma|_{\text{valid}_\sigma}$
- (P3) $\forall b \in \text{adr}(m_\tau|_{\text{valid}_\tau}) \cup A. \mathcal{F}_\mathcal{O}(\tau, b) \subseteq \mathcal{F}_\mathcal{O}(\sigma, b)$
- (P4) $\forall a \in A \forall p \in \text{PVar}. m_\tau(p) = a \iff m_\sigma(p) = a$
- (P5) $\forall a \in A \forall b \in m_\tau(\text{valid}_\tau). m_\tau(b.\text{next}) = a \iff m_\sigma(b.\text{next}) = a$
- (P6) $\forall a \in A. a \in \text{fresh}_\tau \cup \text{freed}_\tau \iff a \in \text{fresh}_\sigma \cup \text{freed}_\sigma$
- (P7) $\forall a \in A. a \in \text{retired}_\tau \iff a \in \text{retired}_\sigma$

Let $\text{act} = \langle t, \text{com}, \text{up} \rangle$. We show that there is $\text{act}' = \langle t, \text{com}, \text{up}' \rangle$ such that:

- (G0) $\sigma.act' \in \mathcal{O}[[P]]_{Adr}^A$
- (G1) $ctrl(\tau.act) = ctrl(\sigma.act')$
- (G2) $m_{\tau.act}|_{valid_{\tau.act}} = m_{\sigma.act'}|_{valid_{\sigma.act'}}$
- (G3) $\forall b \in adr(m_{\tau.act}|_{valid_{\tau.act}}) \cup A. \mathcal{F}_O(\tau.act, b) \subseteq \mathcal{F}_O(\sigma.act', b)$
- (G4) $\forall a \in A \forall p \in PVar. m_{\tau.act}(p) = a \iff m_{\sigma.act'}(p) = a$
- (G5) $\forall a \in A \forall b \in m_{\tau.act}(valid_{\tau.act}). m_{\tau.act}(b.next) = a \iff m_{\sigma.act'}(b.next) = a$
- (G6) $\forall a \in A. a \in fresh_{\tau.act} \cup freed_{\tau.act} \iff a \in fresh_{\sigma.act'} \cup freed_{\sigma.act'}$
- (G7) $\forall a \in A. a \in retired_{\tau.act} \iff a \in retired_{\sigma.act'}$

Because act' executes the same command com as act , we get (G1) from (P1) provided (G0) holds. We do not comment on (G1) in the following.

◇ **Case 1:** $com \equiv x := y$

Let $b = m_{\tau}(q)$ and $b = m_{\sigma}(q)$. The update is $up = [p \mapsto b]$. We choose $up' = [p \mapsto b']$. Then, (G0) holds by the choice of up' together with (P1).

◇ *Ad (G2) for $q \in valid_{\tau}$.* By (P2) and Lemma B.37 we have $q \in valid_{\sigma}$. Furthermore, (P2) yields $b = b'$. We get

$$m_{\tau.act}|_{valid_{\tau.act}} = m_{\tau}[p \mapsto b]|_{valid_{\tau} \cup \{p\}} = (m_{\tau}|_{valid_{\tau}})[p \mapsto b].$$

The second equality holds by $m_{\tau}(q) = b$. That is, we preserve $b.data$ in $m_{\tau}|_{valid_{\tau}}$ and only need to update the mapping of p . Similarly, we get $m_{\sigma.act'}|_{valid_{\sigma.act'}} = (m_{\sigma}|_{valid_{\sigma}})[p \mapsto b']$. Then, we conclude by $m_{\tau}|_{valid_{\tau}} = m_{\sigma}|_{valid_{\sigma}}$ from (P2) together with $b = b'$.

◇ *Ad (G2) for $q \notin valid_{\tau}$.* By (P2) together with Lemma B.37 we have $q \notin valid_{\sigma}$. We get

$$m_{\tau.act}|_{valid_{\tau.act}} = m_{\tau}[p \mapsto b]|_{valid_{\tau} \setminus \{p\}} = m_{\tau}|_{valid_{\tau} \setminus \{p\}}.$$

The last equality holds because the update does not survive the restriction to a set which is guaranteed not to contain p . Similarly, we get $m_{\sigma.act'}|_{valid_{\sigma.act'}} = m_{\sigma}|_{valid_{\sigma} \setminus \{p\}}$. We conclude by:

$$\begin{aligned} m_{\tau}|_{valid_{\tau} \setminus \{p\}} &= (m_{\tau}|_{valid_{\tau}})|_{valid_{\tau} \setminus \{p\}} = (m_{\sigma}|_{valid_{\sigma}})|_{valid_{\tau} \setminus \{p\}} \\ &= (m_{\sigma}|_{valid_{\sigma}})|_{valid_{\sigma} \setminus \{p\}} = m_{\sigma}|_{valid_{\sigma} \setminus \{p\}}. \end{aligned}$$

The first equality is by definition of restrictions, the second equality is due to (P2), the third one is by (P2) together with Lemma B.37, and the last is again by definition.

◇ *Ad (G3).* By (P3) together with the fact that act and act' do not emit an event, it is sufficient to show $adr(m_{\tau.act}|_{valid_{\tau.act}}) \subseteq adr(m_{\tau}|_{valid_{\tau}})$. This follows from Lemma B.47.

◇ *Ad (G4).* Let $a \in A$. Only the valuation of p is changed by both act and act' . So by (P4) it suffices to show that $m_{\tau.act}(p) = a \iff m_{\sigma.act'}(p) = a$ holds. We conclude by:

$$m_{\tau.act}(p) = a \iff m_{\tau}(q) = a \iff m_{\sigma}(q) = a \iff m_{\sigma.act'}(p) = a$$

where the first equivalence holds due to the update up , the second equivalence holds due to (P4), and the last equivalence holds by up' .

◇ *Ad (G5).* Let $a \in A$. Let $c \in m_{\tau.act}(valid_{\tau.act})$. By Lemma B.47, $c \in m_{\tau}(valid_{\tau})$. We conclude:

$$\begin{aligned} m_{\tau.act}(c.next) = a &\iff m_{\tau}(c.next) = a \iff m_{\sigma}(c.next) = a \\ &\iff m_{\sigma.act'}(c.next) = a \end{aligned}$$

where the first/last equivalence hold because act/act' does not update any pointer selector and the second equivalence holds by $c \in m_{\tau}(valid_{\tau})$ together with (P5).

◇ *Ad (G6) and (G7).* By definition, we have $fresh_{\tau} = fresh_{\tau.act}$ and $freed_{\tau} = freed_{\tau.act}$ as well as $retired_{\tau} = retired_{\tau.act}$. Similarly, for σ . Then, we conclude by (P6) and (P7).

◇ **Case 2:** $com \equiv x := q.sel$

By assumption, we have $q \in valid_{\tau}$. Let $b = m_{\tau}(q)$. By the semantics, we get $b \neq seg$. Hence, we obtain $m_{\sigma}(q) = b$ from (P2). Let $c = m_{\tau}(b.next)$ and let $c' = m_{\sigma}(b.next)$. The update up is of the form $up = [p \mapsto c]$. We choose $up' = [p \mapsto c']$. Then, (G0) holds by the choice of up' and (P1).

◇ *Ad (G2) for $b.next \in valid_{\tau}$.* By (P2) and Lemma B.37, $b.next \in valid_{\sigma}$. So, $c = c'$. We get

$$m_{\tau.act}|_{valid_{\tau.act}} = m_{\tau}[p \mapsto c]|_{valid_{\tau} \cup \{p\}} = (m_{\tau}|_{valid_{\tau}})[p \mapsto c].$$

The second equality holds by $m_{\tau}(b.next) = c$. That is, we preserve $c.data$ in $m_{\tau}|_{valid_{\tau}}$ and only need to update p . Similarly, $m_{\sigma.act'}|_{valid_{\sigma.act'}} = (m_{\sigma}|_{valid_{\sigma}})[p \mapsto c']$. Then, we conclude by $m_{\tau}|_{valid_{\tau}} = m_{\sigma}|_{valid_{\sigma}}$ from (P2) together with $c = c'$.

◇ *Ad (G2).* By (P2) together with Lemma B.37, we have $b.next \notin valid_{\sigma}$. We get

$$m_{\tau.act}|_{valid_{\tau.act}} = m_{\tau}[p \mapsto c]|_{valid_{\tau} \setminus \{p\}} = m_{\tau}|_{valid_{\tau} \setminus \{p\}}.$$

The last equality holds because the update does not survive the restriction to a set which is guaranteed not to contain p . Similarly, we get $m_{\sigma.act'}|_{valid_{\sigma.act'}} = m_{\sigma}|_{valid_{\sigma} \setminus \{p\}}$. We conclude by:

$$\begin{aligned} m_{\tau}|_{valid_{\tau} \setminus \{p\}} &= (m_{\tau}|_{valid_{\tau}})|_{valid_{\tau} \setminus \{p\}} = (m_{\sigma}|_{valid_{\sigma}})|_{valid_{\tau} \setminus \{p\}} \\ &= (m_{\sigma}|_{valid_{\sigma}})|_{valid_{\sigma} \setminus \{p\}} = m_{\sigma}|_{valid_{\sigma} \setminus \{p\}}. \end{aligned}$$

The first equality is by definition of restrictions, the second equality is due to (P2), the third one is by (P2) together with Lemma B.37, and the last is again by definition.

◇ *Ad (G3).* By (P3) together with the fact that act and act' do not emit an event, it is sufficient to show $adr(m_{\tau.act}|_{valid_{\tau.act}}) \subseteq adr(m_{\tau}|_{valid_{\tau}})$. This follows from Lemma B.47.

- ◇ *Ad (G4).* Let $a \in A$. Only the valuation of p is changed by both act and act' . So by (P4) it suffices to show that $m_{\tau.act}(p) = a \iff m_{\sigma.act'}(p) = a$ holds. We conclude by:

$$m_{\tau.act}(p) = a \iff m_{\tau}(b.next) = a \iff m_{\sigma}(b.next) = a \iff m_{\sigma.act'}(p) = a$$

where the first/last equivalence holds due to the update up/up' . The second equivalence is due to (P5) because $q \in valid_{\tau}$ from above yields $b = m_{\tau}(q) \in m_{\tau}(valid_{\tau})$.

- ◇ *Ad (G5).* Let $a \in A$. Let $\hat{a} \in m_{\tau.act}(valid_{\tau.act})$. By Lemma B.47, $\hat{a} \in m_{\tau}(valid_{\tau})$. We then conclude by:

$$\begin{aligned} m_{\tau.act}(\hat{a}.next) = a &\iff m_{\tau}(\hat{a}.next) = a \iff m_{\sigma}(\hat{a}.next) = a \\ &\iff m_{\sigma.act'}(\hat{a}.next) = a \end{aligned}$$

where the first/last equivalence hold because act/act' does not update any pointer selector and the second equivalence holds by $\hat{a} \in m_{\tau}(valid_{\tau})$ together with (P5).

- ◇ *Ad (G6) and (G7).* By definition, we have $fresh_{\tau} = fresh_{\tau.act}$ and $freed_{\tau} = freed_{\tau.act}$ as well as $retired_{\tau} = retired_{\tau.act}$. Similarly, for σ . Then, we conclude by (P6) and (P7).

- ◇ **Case 3:** $com \equiv p.sel := y$

By assumption, we have $p \in valid_{\tau}$ and thus $p \in valid_{\sigma}$ by (P2) together with Lemma B.37. Then, the claim follows analogously to the previous case.

- ◇ **Case 4:** $com \equiv u := op(u_1, \dots, u_n)$

Let $d_i = m_{\tau}(u_i)$. Then, $up = [u \mapsto d]$ with $d = op(d_1, \dots, d_n)$. We have $u_i \in dom(m_{\tau}|_{valid_{\tau}})$ by definition. So (P2) yields $m_{\sigma}(u_i) = d_i$. We choose $up' = [u \mapsto d]$. Then, (G0) holds by the choice of up' together with (P1). The remaining (G1) to (G7) follow from (P1) to (P7).

- ◇ **Case 5:** $com \equiv u := q.data$

By assumption, $q \in valid_{\tau}$. Let $b = m_{\tau}(q)$. By definition, $b \neq seg$. Let $d = m_{\tau}(b.data)$. The update is $up = [u \mapsto d]$. By definition, $b.data \in dom(m_{\tau}|_{valid_{\tau}})$. So (P2) and Lemma B.37 together with (P2) yields $m_{\sigma}(q) = b$ and $m_{\sigma}(b.data) = d$. We choose $up' = up$. Then, (G0) holds by the choice of up' together with (P1). The remaining (G1) to (G7) follow immediately from (P1) to (P7).

- ◇ **Case 6:** $com \equiv p.data := u'$

By assumption, we have $p \in valid_{\tau}$ and thus $p \in valid_{\sigma}$ by (P2) together with Lemma B.37. Then, the claim follows analogously to the previous case.

The above case distinction concludes the claim. ■

Proof C.48 (Lemma B.68). Let $\tau.act \in \mathcal{O}[P]_{Adr}^{Adr}$ and $\sigma.act \in \mathcal{O}[P]_{Adr}^A$ with $\tau \sim \sigma$, $\tau \leq_A \sigma$, and $\tau \leq_A \sigma$. Let $act = \langle t, com, up \rangle$. Unrolling the premise gives:

$$(P0) \quad \sigma.act \in \mathcal{O}[P]_{Adr}^A$$

$$(P1) \quad ctrl(\tau) = ctrl(\sigma)$$

- (P2) $m_\tau|_{\text{valid}_\tau} = m_\sigma|_{\text{valid}_\sigma}$
- (P3) $\forall b \in \text{adr}(m_\tau|_{\text{valid}_\tau}) \cup A. \mathcal{F}_\mathcal{O}(\tau, b) \subseteq \mathcal{F}_\mathcal{O}(\sigma, b)$
- (P4) $\forall a \in A \forall p \in \text{PVar}. m_\tau(p) = a \iff m_\sigma(p) = a$
- (P5) $\forall a \in A \forall b \in m_\tau(\text{valid}_\tau). m_\tau(b.\text{next}) = a \iff m_\sigma(b.\text{next}) = a$
- (P6) $\forall a \in A. a \in \text{fresh}_\tau \cup \text{freed}_\tau \iff a \in \text{fresh}_\sigma \cup \text{freed}_\sigma$
- (P7) $\forall a \in A. a \in \text{retired}_\tau \iff a \in \text{retired}_\sigma$

We show the following:

- (G1) $\text{ctrl}(\tau.\text{act}) = \text{ctrl}(\sigma.\text{act})$
- (G2) $m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}} = m_{\sigma.\text{act}}|_{\text{valid}_{\sigma.\text{act}}}$
- (G3) $\forall b \in \text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) \cup A. \mathcal{F}_\mathcal{O}(\tau.\text{act}, b) \subseteq \mathcal{F}_\mathcal{O}(\sigma.\text{act}, b)$
- (G4) $\forall a \in A \forall p \in \text{PVar}. m_{\tau.\text{act}}(p) = a \iff m_{\sigma.\text{act}}(p) = a$
- (G5) $\forall a \in A \forall b \in m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}}). m_{\tau.\text{act}}(b.\text{next}) = a \iff m_{\sigma.\text{act}}(b.\text{next}) = a$
- (G6) $\forall a \in A. a \in \text{fresh}_{\tau.\text{act}} \cup \text{freed}_{\tau.\text{act}} \iff a \in \text{fresh}_{\sigma.\text{act}} \cup \text{freed}_{\sigma.\text{act}}$
- (G7) $\forall a \in A. a \in \text{retired}_{\tau.\text{act}} \iff a \in \text{retired}_{\sigma.\text{act}}$

Because *act* executes the same command *com* as *act*, we get (G1) from (P0) and (P1) holds. We do not comment on (G1) in the following.

◇ **Case 1:** $\text{com} \equiv \text{in:func}(r_1, \dots, r_n)$

The update is $up = \emptyset$. We choose $up' = up$. Then, (G2) follows from (P2), and (G4) to (G7) follow from (P4) to (P7). It remains to establish (G3). To that end, let $v_i = m_\tau(r_i)$. By assumption, $m_\sigma(r_i) = v_i$. That is, *act* emits $\text{evt} = \text{in:func}(t, v_1, \dots, v_n)$ after both τ and σ . This means that we have $\mathcal{H}(\tau.\text{act}) = \mathcal{H}(\tau).\text{evt}$ as well as $\mathcal{H}(\sigma.\text{act}) = \mathcal{H}(\sigma).\text{evt}$. Thus, (G3) follows from (P3) together with Lemma B.41.

◇ **Case 2:** $\text{com} \equiv \text{re:func}$

Analogous to the previous case.

◇ **Case 3:** $\text{com} \equiv \text{assume } p = q$

The update is $up = \emptyset$. That is, we get $m_{\tau.\text{act}} = m_\tau$ as well as $m_{\sigma.\text{act}} = m_\sigma$. By the semantics, we have $m_\tau(p) = m_\tau(q)$ and $m_\sigma(p) = m_\sigma(q)$. Then, (G4), (G6) and (G7) follow immediately from (P4), (P6) and (P7).

◇ *Ad (G2).* If $\{p, q\} \cap \text{valid}_\tau = \emptyset$ we get $\text{valid}_\tau = \text{valid}_{\tau.\text{act}}$ and $\text{valid}_\sigma = \text{valid}_{\sigma.\text{act}}$ as before so that we conclude by (P2):

$$m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}} = m_\tau|_{\text{valid}_\tau} = m_\sigma|_{\text{valid}_\sigma} = m_{\sigma.\text{act}}|_{\text{valid}_{\sigma.\text{act}}}.$$

Consider now the case $\{p, q\} \cap \text{valid}_\tau \neq \emptyset$. Then, we have $\text{valid}_{\tau.act} = \text{valid}_\tau \cup \{p, q\}$ by definition. So we get $\text{valid}_{\sigma.act} = \text{valid}_\sigma \cup \{p, q\}$ by (P2) together with Lemma B.37. From Lemma B.47 we get $m_{\tau.act}(\text{valid}_{\tau.act}) = m_\tau(\text{valid}_\tau)$ and $m_{\sigma.act}(\text{valid}_{\sigma.act}) = m_\sigma(\text{valid}_\sigma)$. Combined with (P2), we obtain:

$$\begin{aligned}
\text{dom}(m_{\tau.act}|_{\text{valid}_{\tau.act}}) &= \text{valid}_{\tau.act} \cup \text{DVar} \cup \{c.\text{data} \mid c \in m_{\tau.act}(\text{valid}_{\tau.act})\} \\
&= \text{valid}_\tau \cup \{p, q\} \cup \text{DVar} \cup \{c.\text{data} \mid c \in m_\tau(\text{valid}_\tau)\} \\
&= \text{dom}(m_\tau|_{\text{valid}_\tau}) \cup \{p, q\} = \text{dom}(m_\sigma|_{\text{valid}_\sigma}) \cup \{p, q\} \\
&= \text{valid}_\sigma \cup \{p, q\} \cup \text{DVar} \cup \{c.\text{data} \mid c \in m_\sigma(\text{valid}_\sigma)\} \\
&= \text{valid}_{\sigma.act} \cup \text{DVar} \cup \{c.\text{data} \mid c \in m_{\sigma.act}(\text{valid}_{\sigma.act})\} \\
&= \text{dom}(m_{\sigma.act}|_{\text{valid}_{\sigma.act}})
\end{aligned}$$

Let $\text{exp} \in \text{dom}(m_{\tau.act}|_{\text{valid}_{\tau.act}})$. It remains to show $m_{\tau.act}(\text{exp}) = m_{\sigma.act}(\text{exp})$. By the above, we have $\text{exp} \in \text{dom}(m_\tau|_{\text{valid}_\tau}) \cup \{p, q\}$. So by (P2) together with $m_{\tau.act} = m_\tau$ and $m_{\sigma.act} = m_\sigma$ it remains to establish $m_{\tau.act}(p) = m_{\sigma.act}(p)$ and $m_{\tau.act}(q) = m_{\sigma.act}(q)$. Wlog. $p \in \text{valid}_\tau$. Then, we have $m_\tau(p) = m_\sigma(p)$. Hence, $m_{\tau.act}(p) = m_{\sigma.act}(p)$. And the assumption in *com* requires p and q to have the same valuation, $m_{\tau.act}(q) = m_{\sigma.act}(q)$. This concludes the claim.

◇ *Ad (G3).* We have $\mathcal{F}_O(\tau.act, c) = \mathcal{F}_O(\tau, c)$ as well as $\mathcal{F}_O(\sigma.act, c) = \mathcal{F}_O(\sigma, c)$ for all $c \in \text{Adr}$ since *act* does not emit an event. Thus, $\text{adr}(m_{\tau.act}|_{\text{valid}_{\tau.act}}) \subseteq \text{adr}(m_\tau|_{\text{valid}_\tau})$, by Lemma B.47. Then, we conclude by (P3).

◇ *Ad (G5).* Follows from $m_{\tau.act} = m_\tau$ and $m_{\sigma.act} = m_\sigma$ together with (P5) and Lemma B.47.

◇ **Case 4:** *com* \equiv assume $p \neq q$

Since *act* has no effect on τ and σ , (G1) to (G7) follow immediately from (P1) to (P7).

◇ **Case 5:** *com* \equiv assume $\text{pred}(\bar{u})$

Since *act* has no effect on τ and σ , (G1) to (G7) follow immediately from (P1) to (P7).

◇ **Case 6:** *com* $\equiv p := \text{malloc}$

Let $b = m_{\tau.act}(p)$. The update is $up = [p \mapsto b, b.\text{next} \mapsto \text{seg}, b.\text{data} \mapsto d]$ for some d . By definition, we have $b \in \text{fresh}_\tau \cup \text{freed}_\tau$. By (P0) we have $b \in \text{fresh}_\sigma \cup \text{freed}_\sigma$. Note that the following holds by assumption:

$$b \notin A \implies \mathcal{F}_O(\tau, b) \subseteq \mathcal{F}_O(\sigma, b) \quad (16)$$

Before we establish the remaining properties, we show two auxiliary statements:

$$m_{\tau.act}(\text{valid}_{\tau.act}) = m_\tau(\text{valid}_\tau \setminus \{p, b.\text{next}\}) \cup \{b\} \quad (17)$$

$$m_{\sigma.act}(\text{valid}_{\sigma.act}) = m_\sigma(\text{valid}_\sigma \setminus \{p, b.\text{next}\}) \cup \{b\} \quad (18)$$

- ◇ *Ad (17).* Note that act changes only the valuation of p and $b.next$. Moreover, by definition, we have $valid_{\tau.act} = valid_{\tau} \cup \{p, b.next\}$. So we conclude as follows:

$$\begin{aligned} m_{\tau.act}(valid_{\tau.act}) &= m_{\tau.act}(valid_{\tau} \cup \{p, b.next\}) \\ &= m_{\tau.act}(valid_{\tau} \setminus \{p, b.next\}) \cup m_{\tau.act}(\{p, b.next\}) \\ &= m_{\tau}(valid_{\tau} \setminus \{p, b.next\}) \cup \{b\}. \end{aligned}$$

- ◇ *Ad (18).* Follows analogously to (17).

- ◇ *Ad (G2).* Using (17) and $valid_{\tau.act} = valid_{\tau} \cup \{p, b.next\}$, we get:

$$\begin{aligned} dom(m_{\tau.act}|_{valid_{\tau.act}}) &= valid_{\tau.act} \cup DVar \cup \{c.data \mid c \in m_{\tau.act}(valid_{\tau.act})\} \\ &= (valid_{\tau} \setminus \{p, b.next\}) \cup \{p, b.next\} \cup DVar \\ &\quad \cup \{c.data \mid c \in m_{\tau}(valid_{\tau} \setminus \{p, b.next\})\} \cup \{b.data\} \\ &= dom(m_{\tau}|_{valid_{\tau} \setminus \{p, b.next\}}) \cup \{p, b.next, b.data\}. \end{aligned}$$

By definition then, we have:

$$\begin{aligned} m_{\tau.act}|_{valid_{\tau.act}} &= (m_{\tau}|_{valid_{\tau} \setminus \{p, b.next\}})[p \mapsto b, b.next \mapsto seg, b.data \mapsto d] \\ &= ((m_{\tau}|_{valid_{\tau}})|_{valid_{\tau} \setminus \{p, b.next\}})[p \mapsto b, b.next \mapsto seg, b.data \mapsto d]. \end{aligned}$$

Along the same lines, using (18), we get:

$$m_{\sigma.act}|_{valid_{\sigma.act}} = ((m_{\sigma}|_{valid_{\sigma}})|_{valid_{\sigma} \setminus \{p, b.next\}})[p \mapsto b, b.next \mapsto seg, b.data \mapsto d].$$

The above equalities now allow us to conclude the claim using (P2) and (P2) together with Lemma B.37:

$$(m_{\tau}|_{valid_{\tau}})|_{valid_{\tau} \setminus \{p, b.next\}} = (m_{\sigma}|_{valid_{\sigma}})|_{valid_{\sigma} \setminus \{p, b.next\}}.$$

- ◇ *Ad (G3).* By Lemma B.47, we have $adr(m_{\tau.act}|_{valid_{\tau.act}}) \subseteq adr(m_{\tau}|_{valid_{\tau}}) \cup \{b\}$. Moreover, we have $\mathcal{F}_{\mathcal{O}}(\tau.act, c) = \mathcal{F}_{\mathcal{O}}(\tau, c)$ and $\mathcal{F}_{\mathcal{O}}(\sigma.act, c) = \mathcal{F}_{\mathcal{O}}(\sigma, c)$ for all $c \in Adr$ since act does not emit an event. By (P3) it thus remains to show that $\mathcal{F}_{\mathcal{O}}(\tau, b) \subseteq \mathcal{F}_{\mathcal{O}}(\sigma, b)$. If $b \in A$, then we get the desired inclusion from (P3). Otherwise, we get it from (16).
- ◇ *Ad (G4).* We have $m_{\tau.act}(p) = m_{\sigma.act}(p)$. For $q \in PVar \setminus \{p\}$, we have $m_{\tau.act}(q) = m_{\tau}(q)$ as well as $m_{\sigma.act}(q) = m_{\sigma}(q)$. Hence, the claim follows from (P4).
- ◇ *Ad (G5).* We have $m_{\tau.act}(b.next) = m_{\sigma.act}(b.next)$. For $c.next \in PSel \setminus \{b.next\}$, we obtain then $m_{\tau.act}(c.next) = m_{\tau}(c.next)$ and $m_{\sigma.act}(c.next) = m_{\sigma}(c.next)$. We conclude by (P5).
- ◇ *Ad (G6).* We have $fresh_{\tau.act} = fresh_{\tau} \setminus \{b\}$ and $freed_{\tau.act} = freed_{\tau} \setminus \{b\}$. Similarly, we have $fresh_{\sigma.act} = fresh_{\sigma} \setminus \{b\}$ and $freed_{\sigma.act} = freed_{\sigma} \setminus \{b\}$. We conclude by (P6).
- ◇ *Ad (G7).* We have $retired_{\tau.act} = retired_{\tau}$ and $retired_{\sigma.act} = retired_{\sigma}$. Then, the claim follows from (P7).

◇ **Case 7:** $com \equiv \text{free}(b)$

The update is $up = \emptyset$. By assumption, we have:

$$\forall c \in \text{Adr} \setminus \{b\}. \mathcal{F}_\mathcal{O}(\tau.act, c) = \mathcal{F}_\mathcal{O}(\tau, c) \quad (19)$$

$$\forall c \in \text{Adr} \setminus \{b\}. \mathcal{F}_\mathcal{O}(\sigma.act, c) = \mathcal{F}_\mathcal{O}(\sigma, c) \quad (20)$$

- ◇ *Ad (G2).* We have $\text{valid}_{\tau.act} = \text{valid}_{\sigma.act}$. To see this, consider $pexp \in \text{valid}_{\tau.act}$. By definition, this means $pexp \in \text{valid}_\tau$ and $pexp \neq b.\text{next}$ and $m_\tau(pexp) \neq b$. From Lemma B.37 and (P2) we obtain $pexp \in \text{valid}_\sigma$. By (P2), $m_\sigma(pexp) = m_\tau(pexp) \neq b$. Hence, $pexp \in \text{valid}_{\sigma.act}$ holds by definition. This establishes $\text{valid}_{\tau.act} \subseteq \text{valid}_{\sigma.act}$. The reverse inclusion follows analogously. Next, observe that $m_{\tau.act}|_{\text{valid}_{\tau.act}} = m_\tau|_{\text{valid}_{\tau.act}}$ because of $m_\tau = m_{\tau.act}$ due to $up = \emptyset$. Along the same lines, $m_{\sigma.act}|_{\text{valid}_{\sigma.act}} = m_\sigma|_{\text{valid}_{\sigma.act}}$. Now, it is sufficient to show $m_\tau|_{\text{valid}_{\tau.act}} = m_\sigma|_{\text{valid}_{\sigma.act}}$. We conclude by:

$$\begin{aligned} m_\tau|_{\text{valid}_{\tau.act}} &= (m_\tau|_{\text{valid}_\tau})|_{\text{valid}_{\tau.act}} = (m_\sigma|_{\text{valid}_\sigma})|_{\text{valid}_{\tau.act}} \\ &= (m_\sigma|_{\text{valid}_\sigma})|_{\text{valid}_{\sigma.act}} = m_\sigma|_{\text{valid}_{\sigma.act}} \end{aligned}$$

where the first holds by $\text{valid}_{\tau.act} \subseteq \text{valid}_\tau$, the second equality holds by (P2), the third equality is shown above, and the last equality holds by $\text{valid}_{\sigma.act} \subseteq \text{valid}_\sigma$.

- ◇ *Ad (G3).* Let $c \in \text{adr}(m_{\tau.act}|_{\text{valid}_{\tau.act}}) \cup A$. We have $c \in \text{adr}(m_\tau|_{\text{valid}_\tau}) \cup A$ by Lemma B.35 together with the same reasoning as above. Let $h \in \mathcal{F}_\mathcal{O}(\tau.act, c)$. We now establish that $h \in \mathcal{F}_\mathcal{O}(\sigma.act, c)$ holds. First consider the case $b = c$. We get $\text{free}(c).h \in \mathcal{F}_\mathcal{O}(\tau, c)$ from Lemma B.41. So (P3) yields $\text{free}(c).h \in \mathcal{F}_\mathcal{O}(\sigma, c)$. Again by Lemma B.41, we obtain the desired $h \in \mathcal{F}_\mathcal{O}(\sigma.act, c)$. Now, consider $c \neq b$. We have $h \in \mathcal{F}_\mathcal{O}(\tau, c)$ by (19). From (P3) we obtain $\text{free}(c).h \in \mathcal{F}_\mathcal{O}(\sigma, c)$. Then, we get the desired $h \in \mathcal{F}_\mathcal{O}(\sigma.act, c)$ from (20).
- ◇ *Ad (G4).* The update up does not change the valuation of any pointer variable. Hence, the claim follows immediately from (P4).
- ◇ *Ad (G5).* Follows immediately from (P5) because we have $m_\tau = m_{\tau.act}$ and $m_\sigma = m_{\sigma.act}$ as well as $\text{valid}_{\tau.act} \subseteq \text{valid}_\tau$ and $\text{valid}_{\sigma.act} \subseteq \text{valid}_\sigma$.
- ◇ *Ad (G6).* Follows from (P6) because by definition we have $\text{fresh}_{\tau.act} = \text{fresh}_\tau \setminus \{b\}$ as well as $\text{freed}_{\tau.act} = \text{freed}_\tau \cup \{b\}$ and similarly for σ .
- ◇ *Ad (G7).* Follows from (P7) because $\text{retired}_{\tau.act} = \text{retired}_\tau \setminus \{b\}$ and similarly for σ .

◇ **Case 8:** $com \equiv \text{env}(b)$

The update is $up = [b.\text{next} \mapsto \text{seg}, b.\text{data} \mapsto d]$ for some d . We have $b \in \text{fresh}_\tau \cup \text{freed}_\tau$ by definition. By (P0), we have $b \in \text{fresh}_\sigma \cup \text{freed}_\sigma$. Then, (G4), (G6) and (G7) follow immediately from (P4), (P6) and (P7). For the remaining properties, we show the auxiliary statements:

$$m_{\tau.act}(\text{valid}_{\tau.act}) = m_\tau(\text{valid}_\tau) \setminus \{b\} \quad (21)$$

$$\text{adr}(m_{\tau.act}|_{\text{valid}_{\tau.act}}) = \text{adr}(m_\tau|_{\text{valid}_\tau}) \setminus \{b\} \quad (22)$$

◇ *Ad (21)*. By definition, $valid_{\tau.act} = valid_{\tau}$. Lemmas B.43 and B.48 yield $b.next \notin valid_{\tau}$. We get:

$$\begin{aligned} m_{\tau.act}(valid_{\tau.act}) &= m_{\tau.act}(valid_{\tau}) = m_{\tau.act}(valid_{\tau} \setminus \{b.next\}) \\ &= m_{\tau}(valid_{\tau} \setminus \{b.next\}) = m_{\tau}(valid_{\tau}). \end{aligned}$$

It remains to show that $b \notin m_{\tau}(valid_{\tau})$. This follows from Lemmas B.43 and B.48.

◇ *Ad (22)*. As before, Lemmas B.43 and B.48 yield $b.next \notin valid_{\tau}$. So, $b \notin valid_{\tau} \cap Adr$. Then, we conclude by Lemma B.35 together with $valid_{\tau.act} = valid_{\tau}$ and (21) as follows:

$$\begin{aligned} adr(m_{\tau.act}|_{valid_{\tau.act}}) &= (valid_{\tau.act} \cap Adr) \cup m_{\tau.act}(valid_{\tau.act}) \\ &= (valid_{\tau} \cap Adr) \cup (m_{\tau}(valid_{\tau}) \setminus \{b\}) \\ &= ((valid_{\tau} \cap Adr) \setminus \{b\}) \cup (m_{\tau}(valid_{\tau}) \setminus \{b\}) \\ &= ((valid_{\tau} \cap Adr) \cup m_{\tau}(valid_{\tau})) \setminus \{b\} = adr(m_{\tau}|_{valid_{\tau}}) \setminus \{b\}. \end{aligned}$$

◇ *Ad (G2)*. As before Lemmas B.43 and B.48 yield $b.next \notin valid_{\tau}$. Then, (21) gives:

$$\begin{aligned} dom(m_{\tau.act}|_{valid_{\tau.act}}) &= valid_{\tau.act} \cup DVar \cup \{c.data \mid c \in m_{\tau.act}(valid_{\tau.act})\} \\ &= (valid_{\tau} \setminus \{b.next\}) \cup DVar \cup \{c.data \mid c \in m_{\tau}(valid_{\tau}) \setminus \{b\}\} \\ &= (valid_{\tau} \cup DVar \cup \{c.data \mid c \in m_{\tau}(valid_{\tau})\}) \setminus \{b.next, b.data\} \\ &= dom(m_{\tau}|_{valid_{\tau}}) \setminus \{b.next, b.data\}. \end{aligned}$$

Similarly, we obtain $dom(m_{\sigma.act}|_{valid_{\sigma.act}}) = dom(m_{\sigma}|_{valid_{\sigma}}) \setminus \{b.next, b.data\}$. By (G2), we get $dom(m_{\tau.act}|_{valid_{\tau.act}}) = dom(m_{\sigma.act}|_{valid_{\sigma.act}})$. Consider $exp \in dom(m_{\tau.act}|_{valid_{\tau.act}})$. By the above, $exp \in dom(m_{\tau}|_{valid_{\tau}}) \setminus \{b.next, b.data\}$. Hence, $m_{\tau}(exp) = m_{\sigma}(exp)$ by (G2). Moreover, $exp \in dom(m_{\sigma.act}|_{valid_{\sigma.act}})$. This gives $m_{\tau.act}|_{valid_{\tau.act}} = m_{\sigma.act}|_{valid_{\sigma.act}}$, as required.

◇ *Ad (G3)*. Follows from (22) and (P3) together with act not emitting an event.

◇ *Ad (G5)*. Follows from (21) and (P5) together with up updating only the selectors of b .

The above case distinction concludes the claim. ■

Proof C.49 (Lemma B.69). Let $\tau_1.act.\tau_2 \in \mathcal{O}[P]_X^Y$ UAF with $act = \langle \perp, env(a), up \rangle$. The update is of the form $up = [a.next \mapsto seg, a.data \mapsto d]$ for some d . By definition, $a \in fresh_{\tau} \cup freed_{\tau}$. We proceed by induction over the structure of τ_2 . In the base case, $\tau_2 = \epsilon$. We get $\tau_1 \in \mathcal{O}[P]_X^Y$ by definition. Consider exp with $m_{\tau_1.act}(exp) \neq m_{\tau_1}(exp)$. Because of the update up , we know that $exp \in \{a.next, a.data\}$. This means $exp \cap Adr = \{a\}$. Then, $a \in fresh_{\tau_1} \cup freed_{\tau_1}$. Hence, we get $a \in fresh_{\tau_1.act} \cup freed_{\tau_1.act}$ by definition. The remaining properties follow immediately by definition. That is, τ_1 satisfies the claim. For the induction step, consider $\tau_1.act.\tau_2.act' \in \mathcal{O}[P]_X^Y$ UAF and assume we have already shown the desired correspondence between $\tau = \tau_1.act.\tau_2$ and $\tau' = \tau_1.\tau_2$. We now establish the correspondence between $\tau.act'$ and $\tau'.act'$. Note that by the semantics, $env(a)$ does not affect the

control location of threads, $ctrl(\tau.act') = ctrl(\tau'.act')$ by definition; we do not comment on this property hereafter. Let $act' = \langle t, com, up' \rangle$.

◇ **Case 1:** com is an assignment

We focus on the case $com \equiv p := q.next$. The remaining cases of assignments follow analogously. Let $b = m_\tau(q)$. By definition, $b \neq seg$. Let $c = m_\tau(q.next)$. The update is of the form $up' = [p \mapsto c]$. Note that we have $p \cap Adr = \emptyset$. We obtain $m_{\tau'}(q) = m_\tau(q) = b$ from induction. Moreover, $q \in valid_\tau$ follows from $\tau.act'$ is UAF by assumption. Lemmas B.43 and B.48 yield $b \notin fresh_\tau \cup freed_\tau$. Hence, we get $m_{\tau'}(b.next) = c$ and thus $\tau'.act' \in \mathcal{O}[\![P]\!]_X^Y$. Since act' emits no event nor affects the fresh/freed/retired addresses, it remains to show the desired memory correspondence. Consider some exp with $m_\tau(exp) \neq m_{\tau'}(exp)$. Since we have $m_{\tau.act'}(p) = m_{\tau'.act'}(p)$ due to the executed update up' , we must have $p \neq exp$. By definition, this means $m_{\tau.act'}(exp) = m_\tau(exp)$ and $m_{\tau'.act'}(exp) = m_{\tau'}(exp)$. Then, we conclude by induction:

$$(exp \cap Adr) \cap (fresh_{\tau.act'} \cup freed_{\tau.act'}) = (exp \cap Adr) \cap (fresh_\tau \cup freed_\tau) \neq \emptyset.$$

◇ **Case 2:** $com \equiv p := \text{malloc}$

Let $b = m_{\tau.act'}(p)$. By definition, $b \in fresh_\tau \cup (freed_\tau \cap Y)$. So $b \in fresh_{\tau'} \cup (freed_{\tau'} \cap Y)$ by induction. This means $\tau'.act' \in \mathcal{O}[\![P]\!]_X^Y$. By induction and definition, we get:

$$\begin{aligned} fresh_{\tau.act'} &= fresh_\tau \setminus b = fresh_{\tau'} \setminus b = fresh_{\tau'} \\ \text{and} \quad freed_{\tau.act'} &= freed_\tau \setminus b = freed_{\tau'} \setminus b = freed_{\tau'} \\ \text{and} \quad retired_{\tau.act'} &= retired_\tau \setminus b = retired_{\tau'} \setminus b = retired_{\tau'}. \end{aligned}$$

The remaining property, the memory correspondence, follows as in the previous case.

◇ **Case 3:** $com \equiv \text{in:func}(r_1, \dots, r_n)$

We have $r_i \cap Adr = \emptyset$. Hence, $m_\tau(r_i) = v_i = m_{\tau'}(r_i)$. Together with induction, this means:

$$\mathcal{H}(\tau.act') = \mathcal{H}(\tau).\text{in:func}(t, v_1, \dots, v_n) = \mathcal{H}(\tau').\text{in:func}(t, v_1, \dots, v_n) = \mathcal{H}(\tau'.act').$$

This concludes the claim as the remaining properties follow immediately by induction together with the fact that neither the memory nor the fresh/freed/retired addresses are altered.

◇ **Case 4:** $com \equiv \text{re:func}$

Analogous to the previous case.

◇ **Case 5:** $com \equiv \text{assume cond}$

As before, we have $m_\tau(x) = m_{\tau'}(x)$ for any variable $x \in PVar \cup DVar$ that appears in $cond$. Hence, condition $cond$ has the same truth value after τ and τ' . This means $\tau'.act' \in \mathcal{O}[\![P]\!]_X^Y$. This concludes the claim as the remaining properties follow immediately by induction together with the fact that neither the memory nor the fresh/freed/retired addresses are altered.

◇ **Case 6:** $com \equiv \text{free}(b)$

The update is $up = \emptyset$. By definition, we have $\mathcal{H}(\tau).\text{free}(b) \in \mathcal{S}(\mathcal{O})$. Hence, induction gives $\mathcal{H}(\tau').\text{free}(b) \in \mathcal{S}(\mathcal{O})$. This means $\tau'.act' \in \mathcal{O}[\![P]\!]_X^Y$. By induction and definition, we have:

$$\begin{aligned} \text{fresh}_{\tau.act'} &= \text{fresh}_{\tau} \setminus b = \text{fresh}_{\tau'} \setminus b = \text{fresh}_{\tau'} \\ \text{and} \quad \text{freed}_{\tau.act'} &= \text{freed}_{\tau} \cup b = \text{freed}_{\tau'} \cup b = \text{freed}_{\tau'} \\ \text{and} \quad \text{retired}_{\tau.act'} &= \text{retired}_{\tau} \setminus b = \text{retired}_{\tau'} \setminus b = \text{retired}_{\tau'} . \end{aligned}$$

For the remaining property, consider exp with $m_{\tau.act'}(exp) \neq m_{\tau'.act'}(exp)$. Since $m_{\tau.act'} = m_{\tau}$ and $m_{\tau'.act'} = m_{\tau'}$, we have $m_{\tau}(exp) \neq m_{\tau'}(exp)$. By induction, this means that exp is of the form $c.\text{sel}$ and $c \in \text{fresh}_{\tau} \cup \text{freed}_{\tau}$. By the above, $c \in \text{fresh}_{\tau.act'} \cup \text{freed}_{\tau.act'}$. This concludes the desired memory correspondence.

◇ **Case 7:** $com \equiv \text{env}(b)$

We have $up' = [b.\text{next} \mapsto \text{seg}, b.\text{data} \mapsto d]$ for some d . By definition, $b \in \text{fresh}_{\tau} \cup \text{freed}_{\tau}$. This means $b \in \text{fresh}_{\tau'} \cup \text{freed}_{\tau'}$ by induction. So $\tau'.act' \in \mathcal{O}[\![P]\!]_X^Y$. The remaining properties follows similarly to the previous case.

◇ **Case 8:** $com \in \{\text{skip}, \text{beginAtomic}, \text{endAtomic}, @\text{inv} \bullet\}$

The claim follows immediately by induction since no event is emitted, the memory is not updated, and the fresh/freed/retired addresses are not altered.

The case distinction is complete and thus concludes the induction. ■

Proof C.50 (Lemma B.70). Let $\tau \in \mathcal{O}[\![P]\!]_X^Y$ UAF and let $a \in \text{retired}_{\tau} \cap \text{freed}_{\tau}$. We show that there is a double retire in $\mathcal{O}[\![P]\!]_X^Y$. To that end, we first show that τ is of the following form:

$$\begin{aligned} \tau &= \overbrace{\tau_1 . \langle t_1, \text{retire}(p_1), up_1 \rangle}^{\sigma_1} . \underbrace{\tau_2 . \langle t_2, \text{free}(a), up_2 \rangle}_{act_2} . \overbrace{\tau_3 . \langle t_3, \text{retire}(p_3), up_3 \rangle}^{\sigma_2} . \tau_4 \\ &\quad \text{with } m_{\tau_1}(p_1) = a = m_{\sigma_1, \sigma_2}(p_3) \text{ and } a \notin \text{frees}_{\tau_2} \cup \text{frees}_{\tau_3} \cup \text{frees}_{\tau_4} . \end{aligned}$$

To see this, recall $a \in \text{retired}_{\tau} \cap \text{freed}_{\tau}$. By definition, this means that a has been freed. That is, there is a latest $\text{free}(a)$ in τ , say act_2 . Moreover, a has been retired. By definition, the retirement must have happened after the free of act_2 as otherwise we would not arrive at $a \in \text{retired}_{\tau}$. Let act_3 be the first retirement of a after act_2 . By Lemma B.46, we must have $a \in \text{retired}_{\sigma_1}$. That is, there must be a retirement of a prior to act_2 . Let act_1 be the latest such retirement. Next, observe that $a \notin \text{frees}_{\tau_2}$ must hold. If this was not the case, we would get $a \notin \text{retired}_{\sigma_1}$ since a is not retired in τ_2 by choice of act_1 . We already showed $a \in \text{retired}_{\sigma_1}$ so that we obtain $a \notin \text{frees}_{\tau_2}$ indeed. Lastly, $a \notin \text{frees}_{\tau_3}$ and $a \notin \text{frees}_{\tau_4}$ follows from the choice of act_2 being the last $\text{free}(a)$ in τ .

Now, consider $\gamma = \tau_1.act_1.\tau_2.act_2.\tau_3'$ where τ_3' corresponds to τ_3 with all actions removed that execute command $\text{env}(a)$. By Lemma B.69 we have $\gamma \in \mathcal{O}[\![P]\!]_X^Y$ and $m_{\gamma}(p_3) = m_{\sigma_1, \sigma_2}(p_3) = a$. Note that $a \notin \text{frees}_{\tau_3'}$. Finally, construct computation $\gamma' = \tau_1.act_1.\tau_2.\tau_3'$ which corresponds to γ up to the removal of act_2 . Since act_2 does not update the

memory, $up_2 = \emptyset$ by definition, we have $m_{\gamma'}(p_3) = a$. Moreover, we have $a \in \text{retired}_{\gamma'}$ because $a \in \text{retired}_{\tau_1, \text{act}_1, \tau_2}$ as shown above and $a \notin \text{frees}_{\tau_3}$. So by definition, this means γ' performs a double retire.

It remains to show that $\gamma' \in \mathcal{O}[\![P]\!]_X^Y$ holds. First, observe that act_2 does not update the memory nor the control locations of threads. Second, note that by Lemma B.56 the enabledness of $\text{free}(b)$ with $a \neq b$ remains unaffected whether or not $\text{free}(a)$ is executed. Combining these two properties, we conclude that an action of τ_3' may no longer be enabled only if it requires a to be free. Since τ_3 does not contain commands $\text{env}(a)$ by construction, enabledness is in question only for actions of a . Such an allocation would, by definition, render a allocated in the original prefix of τ , that is, $a \notin \text{freed}_{\sigma_1, \sigma_2}$. Because we have $a \notin \text{frees}_{\tau_3} \cup \text{frees}_{\tau_4}$, we arrive at $a \notin \text{freed}_{\tau}$ which contradicts the assumption. Altogether, this means that the enabledness of all actions in τ_3 does not rely on act_2 . Hence, $\gamma' \in \mathbb{X}Y$. \blacksquare

Proof C.51 (Theorem 7.20). Let \mathcal{O} support elision and let $\mathcal{O}[\![P]\!]_{\text{Addr}}^{\text{one}}$ be free from pointer races, double retires, and harmful ABAs. We proceed by induction over the structure of τ . In the base case, $\tau = \epsilon$. Choose $\sigma = \tau$. This satisfies the claim. For the induction step, consider some $\tau.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Addr}}^{\text{Addr}}$ PRF and assume we have already constructed, for all addresses $a \in \text{Addr}$, some $\sigma \in \mathcal{O}[\![P]\!]_{\text{Addr}}^{\{a\}}$ with $\tau \sim \sigma$, $\tau < \sigma$, and $\tau \preceq_a \sigma$. Let $a \in \text{Addr}$. Let $\text{act} = \langle t, \text{com}, up \rangle$. We construct some $\hat{\sigma} \in \mathcal{O}[\![P]\!]_{\text{Addr}}^{\{a\}}$ such that $\tau.\text{act} \sim \hat{\sigma}$, $\tau.\text{act} \preceq_a \hat{\sigma}$, and $\tau.\text{act} < \hat{\sigma}$. To that end, we do a case distinction over com .

◇ **Case 1:** com is an assignment

Our goal is to find some act' such that $\hat{\sigma} = \sigma.\text{act}'$ satisfies the requirements. We obtain act' from an invocation of Lemma B.67. That is, it remains to show that Lemma B.67 is enabled. To that end, we have to show: if com contains $p.\text{sel}$ then $p \in \text{valid}_{\tau}$. So, assume com contains $p.\text{sel}$ as nothing needs to be shown otherwise. We proceed as follows: we show that com is enabled after σ and then use pointer race freedom to establish the validity of p .

By Lemma B.50 we have $m_{\sigma}(p) \neq \perp$. That is, $m_{\sigma}(p) \in \text{Addr} \cup \{\text{seg}\}$. Towards a contradiction, assume $m_{\sigma}(p) = \text{seg}$. By Lemma B.49, we have $p \in \text{valid}_{\sigma}$. Then, $m_{\tau}(p) = \text{seg}$ follows from $\tau \sim \sigma$. This means act is not enabled after τ . This contradicts $\tau.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Addr}}^{\text{Addr}}$. Hence, we must have $m_{\sigma}(p) \neq \text{seg}$. So, com is enabled after σ . This means there is an update up' such that $\text{act}'' = \langle t, \text{com}, up' \rangle$ is enabled after σ , that is, $\sigma.\text{act}'' \in \mathcal{O}[\![P]\!]_{\text{Addr}}^{\{a\}}$. (Note that act'' is not necessarily the desired act' .) Because $\mathcal{O}[\![P]\!]_{\text{Addr}}^{\{a\}}$ is free from pointer races by assumption, we must have $p \in \text{valid}_{\tau}$. So, $\tau \in \text{valid}_{\tau}$ by $\tau \sim \sigma$ together with Lemma B.37. Altogether, we can invoke Lemma B.67 for $\tau.\text{act}$ and σ . We obtain act' such that $\hat{\sigma} = \sigma.\text{act}'$ satisfies the claim.

◇ **Case 2:** $\text{com} \equiv \text{in:func}(r_1, \dots, r_n)$

If $m_{\tau}(r_i) = m_{\sigma}(r_i)$ for all $1 \leq i \leq n$, then we get $\sigma.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Addr}}^{\{a\}}$ and thus $\hat{\sigma} = \sigma.\text{act}$ satisfies the claim by Lemma B.68. So assume $m_{\tau}(r_i) \neq m_{\sigma}(r_i)$ for some i . We show that again $\hat{\sigma} := \sigma.\text{act}$ is an adequate choice. To that end, we first show that act is enabled after σ and then show that it has the desired properties. To see enabledness of com , note that we have $m_{\sigma}(r_i) \in \text{Addr} \cup \{\text{seg}\}$ by Lemma B.50. To the contrary, assume $m_{\sigma}(r_i) = \text{seg}$. Then, $r_i \in \text{valid}_{\sigma}$ by Lemma B.49 and thus $m_{\tau}(r_i) = \text{seg}$ by $\tau \sim \sigma$. Since this contradicts enabledness of act after τ , we must have $m_{\sigma}(r_i) \neq \text{seg}$. So, $\sigma.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Addr}}^{\{a\}}$.

Now, we show that $\hat{\sigma} := \sigma.act$ satisfies the claim. Let $\bar{r} = r_1, \dots, r_n$. Let $\bar{v}_\tau = m_\tau(\bar{r})$ and let $\bar{v}_\sigma = m_\sigma(\bar{r})$. Let $\bar{v}_\tau = v_{\tau,1}, \dots, v_{\tau,n}$ and $\bar{v}_\sigma = v_{\sigma,1}, \dots, v_{\sigma,n}$. The main task is to show:

$$\forall c \in \text{adr}(m_{\tau.act}|_{\text{valid}_{\tau.act}}) \cup \{a\}. \mathcal{F}_\mathcal{O}(\tau.act, c) \subseteq \mathcal{F}_\mathcal{O}(\sigma.act, c).$$

Let $c \in \text{adr}(m_{\tau.act}|_{\text{valid}_{\tau.act}}) \cup \{a\}$. Because *act* does not alter the heap nor the validity of expressions, we get $c \in \text{adr}(m_\tau|_{\text{valid}_\tau}) \cup \{a\}$ by Lemma B.35. From $\tau \preceq_a \sigma$ and $\tau < \sigma$ we get $\mathcal{F}_\mathcal{O}(\tau, c) \subseteq \mathcal{F}_\mathcal{O}(\sigma, c)$. This implies:

$$\mathcal{F}_\mathcal{O}(\mathcal{H}(\tau).in:func(t, \bar{v}_\tau), c) \subseteq \mathcal{F}_\mathcal{O}(\mathcal{H}(\sigma).in:func(t, \bar{v}_\tau), c). \quad (23)$$

To see this, let $h \in \mathcal{F}_\mathcal{O}(\mathcal{H}(\tau).in:func(t, \bar{v}_\tau), c)$. So, $in:func(t, \bar{v}_\tau).h \in \mathcal{F}_\mathcal{O}(\mathcal{H}(\tau), c)$ by Lemma B.41. Using $\tau \preceq_a \sigma$ and $\tau < \sigma$ we then obtain $in:func(t, \bar{v}_\tau).h \in \mathcal{F}_\mathcal{O}(\mathcal{H}(\sigma), c)$. Again by Lemma B.41, we conclude $h \in \mathcal{F}_\mathcal{O}(\mathcal{H}(\sigma).in:func(t, \bar{v}_\tau), c)$. Next, recall that $\sigma.act$ is PRF. That is, *act* is not racy. From this we get:

$$\mathcal{F}_\mathcal{O}(\mathcal{H}(\sigma).in:func(t, \bar{v}_\tau), c) \subseteq \mathcal{F}_\mathcal{O}(\mathcal{H}(\sigma).in:func(t, \bar{v}_\sigma), c). \quad (24)$$

To see this, we have to show that the following holds for all i with $1 \leq i \leq n$:

$$(v_{\sigma,i} = c \vee r_i \in \text{valid}_\sigma \vee r_i \in \text{DExp}) \implies v_{\tau,i} = v_{\sigma,i}$$

If $r_i \in \text{valid}_\sigma$, then $v_{\sigma,i} = m_\sigma(r_i) = m_\tau(r_i) = v_{\tau,i}$ by $\tau \sim \sigma$. Along the same lines, we get $v_{\sigma,i} = v_{\tau,i}$ if $r_i \in \text{DExp}$ because this means $r_i \in \text{DVar}$ and $\text{DVar} \subseteq \text{dom}(m_\sigma|_{\text{valid}_\sigma})$ by definition. So consider the case where $v_{\sigma,i} = c$ and $r_i \notin \text{valid}_\sigma \cup \text{DExp}$. To the contrary, assume $c \neq a$. By the choice of c , we get $c \in \text{adr}(m_\sigma|_{\text{valid}_\sigma})$. Moreover, $c \in m_\sigma(\text{PVar} \setminus \text{valid}_\sigma)$ follows from $m_\sigma(r) = v_{\sigma,i} = c$ and $r_i \notin \text{valid}_\sigma$. Then, Lemma B.53 yields $c \in \{a\}$. Since this contradicts the assumption, we must have $c = a$. Hence, $m_\sigma(r_i) = c$ implies $m_\tau(r_i) = c$ by $\tau \preceq_a \sigma$ and $r_i \in \text{PVar}$. That is, $b_{1,i} = b_{2,i}$ as desired. Altogether, this proves the desired implication and establishes (24).

Combining (23) and (24), we obtain:

$$\begin{aligned} \mathcal{F}_\mathcal{O}(\tau.act, c) &= \mathcal{F}_\mathcal{O}(\mathcal{H}(\tau).in:func(t, \bar{v}_\tau), c) \subseteq \mathcal{F}_\mathcal{O}(\mathcal{H}(\sigma).in:func(t, \bar{v}_\tau), c) \\ &\subseteq \mathcal{F}_\mathcal{O}(\mathcal{H}(\sigma).in:func(t, \bar{v}_\sigma), c) = \mathcal{F}_\mathcal{O}(\sigma.act, c). \end{aligned}$$

Lastly, note that we have $m_\tau(r_i) = a$ iff $m_\sigma(r_i) = a$ because $\tau \preceq_a \sigma$. By induction, this means $a \in \text{retired}_{\tau.act}$ iff $a \in \text{retired}_{\sigma.act}$ because *act* retires a after τ iff it does so after σ . Altogether, we obtain the desired $\tau.act \sim \sigma.act$, $\tau.act \preceq_a \sigma.act$, and $\tau.act < \sigma.act$ from the above together with the fact that action *act* does not change the memory nor the validity nor the fresh/freed addresses.

◇ **Case 3:** $com \equiv \text{re}:func$

We choose $\hat{\sigma} = \sigma.act$. By $\tau \sim \sigma$, we know that *act* is enabled after σ , that is, $\sigma.act \in \mathcal{O}[[P]]_{\text{Addr}}^{\{a\}}$. By definition, *act* emits the same event $\text{re}:func(t)$ after both τ and σ . Then, the claim follows similarly to the previous case.

◇ **Case 4:** $com \equiv p := \text{malloc}$

Let $b = m_{\tau.act}(p)$. The update is $up = [p \mapsto b, b.\text{next} \mapsto \text{seg}, b.\text{data} \mapsto d]$ for some d . By definition, we have $b \in \text{fresh}_\tau \cup \text{freed}_\tau$. If $a = b$, then $b \in \text{fresh}_\sigma \cup \text{freed}_\sigma$ holds by $\tau \preceq_a \sigma$. Hence, *act* is enabled after σ , i.e.,

$\sigma.act \in \mathcal{O}[P]_{Adr}^{\{a\}}$. We choose $\hat{\sigma} = \sigma.act$. Then, $\hat{\sigma}$ satisfies the claim by Lemma B.68. So consider the remaining case of $a \neq b$ hereafter.

First, we show that $b \notin \text{adr}(m_\sigma|_{\text{valid}_\sigma})$ holds. To that end, we invoke the induction for τ and b . We obtain $\delta \in \mathcal{O}[P]_{Adr}^{\{b\}}$ with $\tau \sim \delta$ and $\tau \preceq_b \delta$. The latter gives $b \in \text{fresh}_\delta \cup \text{freed}_\delta$. Lemmas B.43 and B.48 yield $b \notin m_\delta(\text{valid}_\delta)$ and $b.\text{next} \notin \text{valid}_\delta$. (Note that we cannot conclude this directly for τ since we do not know whether or not τ is free from pointer races.) We combine this with Lemma B.35 to obtain $b \notin \text{adr}(m_\delta|_{\text{valid}_\delta})$. Now, Lemma B.38 for τ and δ yields $b \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$. Lemma B.38 for τ and σ then yields $b \notin \text{adr}(m_\sigma|_{\text{valid}_\sigma})$ as required.

With $b \notin \text{adr}(m_\sigma|_{\text{valid}_\sigma}) \cup \{a\}$ we invoke Lemma B.66 for σ . This gives $\gamma \in \mathcal{O}[P]_{Adr}^{\{a\}}$ such that $\sigma \sim \gamma$, $\sigma \preceq_a \gamma$, $\sigma < \gamma$, and $b \in \text{fresh}_\gamma$. We have $\tau \sim \sigma \sim \gamma$, $\tau \preceq_a \sigma \preceq_a \gamma$, and $\tau < \sigma < \gamma$. By definition and Lemma B.38, $m_\tau(\text{valid}_\tau) \subseteq m_\sigma(\text{valid}_\sigma)$ and $\text{adr}(m_\tau|_{\text{valid}_\tau}) = \text{adr}(m_\sigma|_{\text{valid}_\sigma})$. Then, Lemmas B.32 to B.34 yield $\tau \sim \gamma$, $\tau \preceq_a \gamma$, and $\tau < \gamma$. Moreover, $b \in \text{fresh}_\gamma$ means that act is enabled after γ . That is, $\gamma.act \in \mathcal{O}[P]_{Adr}^{\{a\}}$. We choose $\hat{\sigma} = \gamma.act$. That $\gamma.act$ satisfies the claim is established by Lemma B.68. For Lemma B.68 to apply, we have to show that $\mathcal{F}_\mathcal{O}(\tau, b) \subseteq \mathcal{F}_\mathcal{O}(\gamma, b)$ holds. This, in turn, follows from Lemma B.57. We show that Lemma B.57 is enabled. We already have $b \in \text{fresh}_\gamma$ and $\mathcal{F}_\mathcal{O}(\tau, a) \subseteq \mathcal{F}_\mathcal{O}(\gamma, a)$ where the latter holds by $\tau \preceq_a \gamma$. So, it remains to establish $b \notin \text{retired}_\tau$.

Towards a contradiction, assume $b \in \text{retired}_\tau$. By $\tau \preceq_b \delta$ then, $b \in \text{retired}_\delta$. That is, δ is of the form $\delta = \delta_2.act_1.\delta_1$ with act_1 performing $\text{in:retire}(q)$ and $m_{\delta_2}(q) = b$ and $b \notin \text{frees}_{\delta_1}$. (This decomposition chooses the latest retirement of b for act —such a decomposition must exist.) From the fact that $m_{\delta_2}(q) = b$ holds we get $b \notin \text{fresh}_{\delta_2}$ by Lemma B.42. By monotonicity, $b \notin \text{fresh}_\delta$. Recall that $b \in \text{fresh}_\tau \cup \text{freed}_\tau$. By $\tau \preceq_b \delta$ this gives $b \in \text{fresh}_\delta \cup \text{freed}_\delta$. Hence, we must have $b \in \text{freed}_\delta$ altogether. Then, Lemma B.70 yields a double retire in $\mathcal{O}[P]_{Adr}^{\{b\}}$. This contradicts the assumption of $\mathcal{O}[P]_{Adr}^{\text{one}}$ being free from double retires. Altogether, this means we obtain $b \notin \text{retired}_\tau$ so that Lemma B.57 is applicable. This concludes the claim.

◇ **Case 5:** $\text{com} \equiv \text{free}(b)$

The update is $up = \emptyset$. By definition, $\mathcal{H}(\tau).\text{free}(b) \in \mathcal{S}(\mathcal{O})$. That is, $\text{free}(b) \in \mathcal{F}_\mathcal{O}(\tau, b)$. If we have $\text{free}(b) \in \mathcal{F}_\mathcal{O}(\sigma, b)$, we get $\sigma.act \in \mathcal{O}[P]_{Adr}^{\{a\}}$ and $\mathcal{H}(\sigma).\text{free}(b) \in \mathcal{S}(\mathcal{O})$ by definition. Then, Lemma B.56 discharges the side condition of Lemma B.68 which yields that $\hat{\sigma} = \sigma.act$ satisfies the claim. So consider the remaining case of $\text{free}(b) \notin \mathcal{F}_\mathcal{O}(\sigma, b)$ hereafter. This implies $a \neq b$ because otherwise $\tau \preceq_a \sigma$ gives $\mathcal{F}_\mathcal{O}(\tau, b) \subseteq \mathcal{F}_\mathcal{O}(\sigma, b)$ and thus $\text{free}(b) \in \mathcal{F}_\mathcal{O}(\sigma, b)$.

By $\tau < \sigma$ we conclude that $b \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$ must hold as otherwise $\text{free}(b) \in \mathcal{F}_\mathcal{O}(\sigma, b)$. By $\tau \sim \sigma$ this means $b \notin m_\sigma(\text{valid}_\sigma)$. We now show that $\hat{\sigma} = \sigma$ is an adequate choice, that is, that we do not need to mimic act at all.

◇ *Ad $\tau.act \sim \sigma$.* By definition, $\text{ctrl}(\tau) = \text{ctrl}(\tau.act)$. So, $\text{ctrl}(\tau.act) = \text{ctrl}(\sigma)$ by $\tau \sim \sigma$. To show $m_{\tau.act}|_{\text{valid}_{\tau.act}} = m_\sigma|_{\text{valid}_\sigma}$ it suffices to show $m_\tau|_{\text{valid}_\tau} = m_{\tau.act}|_{\text{valid}_{\tau.act}}$ by $\tau \sim \sigma$. To arrive at this equality, we first show $\text{valid}_\tau = \text{valid}_{\tau.act}$. The inclusion $\text{valid}_{\tau.act} \subseteq \text{valid}_\tau$ holds by definition. To see $\text{valid}_\tau \subseteq \text{valid}_{\tau.act}$, consider $\text{pexp} \in \text{valid}_\tau$. Then, we must have $\text{pexp} \neq b.\text{next}$ as otherwise we had $b \in \text{adr}(m_\tau|_{\text{valid}_\tau})$

by Lemma B.35 which does not hold as shown before. Moreover, we must have $m_\tau(\text{pexp}) \neq b$ as otherwise we would again get $b \in \text{adr}(m_\tau|_{\text{valid}_\tau})$ by Lemma B.35. Hence, by the definition of validity, we have $\text{pexp} \in \text{valid}_{\tau.\text{act}}$. Altogether, this means we have $\text{valid}_\tau = \text{valid}_{\tau.\text{act}}$ indeed. Then, the desired $m_\tau|_{\text{valid}_\tau} = m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}$ follows immediately because $m_\tau = m_{\tau.\text{act}}$ due to $\text{up} = \emptyset$.

◇ *Ad $\tau.\text{act} < \sigma$.* Let $c \in \text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}})$. We get $c \in \text{adr}(m_\tau|_{\text{valid}_\tau})$ by Lemma B.47. This means $b \neq c$ holds due to the above. Moreover, we have $\mathcal{F}_\mathcal{O}(\tau, c) \subseteq \mathcal{F}_\mathcal{O}(\sigma, c)$ by $\tau < \sigma$. So it suffices to show $\mathcal{F}_\mathcal{O}(\tau, c) = \mathcal{F}_\mathcal{O}(\tau.\text{act}, c)$. This follows from Lemma B.56.

◇ *Ad $\tau.\text{act} \preceq_a \sigma$.* First, recall $b \neq a$. So $\tau \preceq_a \sigma$ gives:

$$\begin{aligned} a \in \text{retired}_{\tau.\text{act}} &\iff a \in \text{retired}_\tau \iff a \in \text{retired}_\sigma \\ \text{and } a \in \text{fresh}_{\tau.\text{act}} \cup \text{freed}_{\tau.\text{act}} &\iff a \in \text{fresh}_\tau \cup \text{freed}_\tau \\ &\iff a \in \text{fresh}_\sigma \cup \text{freed}_\sigma. \end{aligned}$$

Similarly to $\tau.\text{act} < \sigma$, we use Lemma B.56 and $\tau \preceq_a \sigma$ together with $b \neq a$ holds to obtain $\mathcal{F}_\mathcal{O}(\tau.\text{act}, a) = \mathcal{F}_\mathcal{O}(\tau, a) \subseteq \mathcal{F}_\mathcal{O}(\sigma, a)$. Since *act* does not modify the memory, the remaining properties follow from $\tau \preceq_a \sigma$ together with $m_\tau|_{\text{valid}_\tau} = m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}$ which we have established for $\tau.\text{act} \sim \sigma$ above.

◇ **Case 6:** $\text{com} \equiv \text{env}(b)$

The update is $\text{up} = [b.\text{next} \mapsto \text{seg}, b.\text{data} \mapsto d]$ for some d . By definition, $b \in \text{fresh}_\tau \cup \text{freed}_\tau$. If $a = b$, then we have $b \in \text{fresh}_\sigma \cup \text{freed}_\sigma$ by $\tau \preceq_a \sigma$. This means $\sigma.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\{a\}}$. Moreover, Lemma B.68 yields that $\hat{\sigma} = \sigma.\text{act}$ satisfies the claim. So, assume $a \neq b$ hereafter. We first establish that $b \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$ holds. To that end, we invoke the induction hypothesis for b . This gives $\gamma \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\{b\}}$ with $\tau \sim \gamma$, $\tau \preceq_b \gamma$, and $\tau < \gamma$. We get $b \in \text{fresh}_\tau \cup \text{freed}_\tau$ because of $\tau \preceq_b \gamma$. Then, Lemmas B.35, B.43 and B.48 yield $b \notin \text{adr}(m_\gamma|_{\text{valid}_\gamma})$. Hence, Lemma B.38 together with $\tau \sim \sigma$ yields the desired $b \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$.

Now, choose $\hat{\sigma} = \sigma$ and conclude as in the previous case. To see $m_\tau|_{\text{valid}_\tau} = m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}$, note that we have:

$$\begin{aligned} \text{dom}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) &= \text{valid}_{\tau.\text{act}} \cup \text{DVar} \cup \{c.\text{data} \mid c \in m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}})\} \\ &= \text{valid}_\tau \cup \text{DVar} \cup \{c.\text{data} \mid c \in m_{\tau.\text{act}}(\text{valid}_\tau)\} \\ &= \text{valid}_\tau \cup \text{DVar} \cup \{c.\text{data} \mid c \in m_\tau(\text{valid}_\tau)\} = \text{dom}(m_\tau|_{\text{valid}_\tau}) \end{aligned}$$

where the first and last equality is by definition, the second equality by $\text{valid}_{\tau.\text{act}} = \text{valid}_\tau$, and the third equality by $b.\text{next} \notin \text{valid}_\tau$ by $b \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$ together with Lemma B.35. Similarly, $b.\text{data} \notin \text{valid}_\tau$ because $b \notin m_\tau(\text{valid}_\tau)$ by $b \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$ together with Lemma B.35. Altogether, this gives $m_\tau|_{\text{valid}_\tau} = m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}$.

◇ **Case 7:** $\text{com} \equiv \text{assume cond}$

By definition, $\text{up} = \emptyset$. If $\sigma.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\{a\}}$, then $\hat{\sigma} = \sigma.\text{act}$ satisfies the claim by Lemma B.68. So assume $\sigma.\text{act} \notin \mathcal{O}[\![P]\!]_{\text{Adr}}^{\{a\}}$. Since we have $m_\tau(u) = m_\sigma(u)$ for all data variables $u \in \text{DVar}$ by $\tau \sim \sigma$, we must have $\text{cond} \equiv p \triangleq q$ with $p, q \in \text{PVar}$ and $\triangleq \in \{=, \neq\}$.

Let $m_\tau(p) = b$. By induction, there is $\delta \in \mathcal{O}[P]_{Adr}^{\{b\}}$ with $\tau \sim \delta$, $\tau \preceq_b \delta$, and $\tau \prec \delta$. By $\tau \preceq_b \delta$, the truth value of *cond* is the same after τ and δ . So, $m_\tau(p) = m_\delta(p)$. Moreover, $m_\delta(q) = b$ if $m_\tau(q) = b$ and otherwise $m_\tau(q) \neq b \neq m_\delta(q)$. Altogether, this means $\delta.act \in \mathcal{O}[P]_{Adr}^{\{b\}}$. From Lemma B.68 we get $\tau.act \sim \delta.act$, $\tau.act \preceq_b \delta.act$, and $\tau.act \prec \delta.act$.

Observe that we have $\delta \sim \sigma$ by Lemma B.32. Then, the absence of harmful ABAs for $\delta.act$ and σ yields a computation $\gamma \in \mathcal{O}[P]_{Adr}^{\{a\}}$ with $\delta.act \sim \gamma$, $\sigma \preceq_a \gamma$, and $\delta.act \prec \gamma$. We show that $\hat{\sigma} = \gamma$ satisfies the claim. To do so, we show an auxiliary property first.

$$m_{\tau.act}(valid_{\tau.act}) = m_\tau(valid_\tau) \quad (25)$$

- ◇ *Ad (25).* By definition, $m_\tau = m_{\tau.act}$. It remains to show $m_\tau(valid_{\tau.act}) = m_\tau(valid_\tau)$. In the case $valid_{\tau.act} = valid_\tau$ holds, nothing needs to be shown. Assume $valid_{\tau.act} \neq valid_\tau$. This means $cond \equiv p = q$ such that wlog. $p \in valid_\tau$ and $q \notin valid_\tau$. So, $m_\tau(p) = m_\tau(q)$ must hold. This means $m_\tau(q) \in m_\tau(valid_\tau)$. Further, $valid_{\tau.act} = valid_\tau \cup \{q\}$. We conclude by: $m_\tau(valid_{\tau.act}) = m_\tau(valid_\tau) \cup \{m_\tau(q)\} = m_\tau(valid_\tau)$.
- ◇ *Ad $\tau.act \sim \gamma$.* Follows from Lemma B.32 together with $\tau.act \sim \delta.act$ and $\delta.act \sim \gamma$.
- ◇ *Ad $\tau.act \prec \gamma$.* We have $\tau.act \prec \delta.act \prec \gamma$ and $adr(m_{\tau.act}|_{valid_{\tau.act}}) = adr(m_{\delta.act}|_{valid_{\delta.act}})$. The latter follows from $\tau.act \sim \delta.act$ together with Lemma B.38. Then, Lemma B.34 gives $\tau.act \prec \gamma$.
- ◇ *Ad $\tau.act \preceq_a \gamma$.* We have $\tau \preceq_a \sigma \preceq_a \gamma$. By $\tau \sim \sigma$, we have $m_\tau(valid_\tau) \subseteq m_\sigma(valid_\sigma)$. Lemma B.33 yields $\tau \preceq_a \gamma$. Assume for the moment we have $\tau.act \preceq_a \tau$. Then, (25) together with Lemma B.33 yields the desired $\tau.act \preceq_a \gamma$. It remains to show $\tau.act \preceq_a \tau$. This follows from $\mathcal{H}(\tau.act) = \mathcal{H}(\tau)$, $m_\tau = m_{\tau.act}$, $freed_{\tau.act} = freed_\tau$, $fresh_{\tau.act} = fresh_\tau$, and $retired_{\tau.act} = retired_\tau$ together with (25).

◇ **Case 8:** $com \in \{\text{skip}, \text{beginAtomic}, \text{endAtomic}, @inv \bullet\}$

We immediately obtain that $\hat{\sigma} = \sigma.act$ satisfies the claim.

The above case distinction is complete and thus concludes the induction. ■

Proof C.52 (Theorem 7.21). If $good(\mathcal{O}[P]_{Adr}^{Adr})$ then, $good(\mathcal{O}[P]_{Adr}^{one})$ by $\mathcal{O}[P]_{Adr}^{one} \subseteq \mathcal{O}[P]_{Adr}^{Adr}$. For the reverse direction, assume $good(\mathcal{O}[P]_{Adr}^{one})$ holds. To the contrary, assume $good(\mathcal{O}[P]_{Adr}^{Adr})$ does not hold. So there is $\tau \in \mathcal{O}[P]_{Adr}^{Adr}$ such that $good(\tau)$ is not satisfied. By definition, this means we have $ctrl(\tau) \cap Fault \neq \emptyset$ where *Fault* are the bad control locations. Theorem 7.20 yields $\sigma \in \mathcal{O}[P]_{Adr}^{one}$ with $\tau \sim \sigma$. The latter gives $ctrl(\sigma) \cap Fault \neq \emptyset$. Hence, $good(\mathcal{O}[P]_{Adr}^{one})$ does not hold. Since this contradicts the assumption, $good(\mathcal{O}[P]_{Adr}^{Adr})$ must hold. ■

Proof C.53 (Theorem 7.22). Towards a contradiction, assume that there is a shortest computation $\tau.act \in \mathcal{O}[P]_{Adr}^{Adr}$ that performs a double retire. That is, we have $act = \langle t, retire(p), up \rangle$ with $m_\tau(p) = a \in retired_\tau$. Theorem 7.20 yields $\sigma \in \mathcal{O}[P]_{Adr}^{\{a\}}$ with $\tau \sim \sigma$ and $\tau \preceq_a \sigma$. The latter gives $m_\sigma(p) = a$ and $a \in retired_\sigma$. By Lemma B.68 and $\tau \sim \sigma$, we have $\sigma.act \in \mathcal{O}[P]_{Adr}^{\{a\}}$. That is, $\mathcal{O}[P]_{Adr}^{one}$ is not free from double retires. Since this contradicts the assumption, $\mathcal{O}[P]_{Adr}^{Adr}$ must be free from double retires. ■

Proof C.54 (Proposition 7.15). In the following, we rely on the following properties of \mathcal{O}_{SMR} : (i) there is at most one variable z_a tracking addresses, (ii) accepting locations l are reached only via transitions of the form $\bullet \xrightarrow{\text{free}(r), r=z_a} l$, and (iii) all transitions of the form $l \xrightarrow{\text{free}(r), r=z_a} l'$ satisfy: $l = l'$ or l' is accepting. These properties are satisfied by \mathcal{O}_{EBR} , $\mathcal{O}_{HP}^0 \times \mathcal{O}_{HP}^1$, and $\mathcal{O}_{HP}^{0,1}$. Let location l_{init} uniformly refer to the initial location in the aforementioned SMR automata.

◇ *Ad Definition 7.14i.* Let $a \neq c \neq b$. We show $\mathcal{F}_{\mathcal{O}_{SMR}}(h, c) = \mathcal{F}_{\mathcal{O}_{SMR}}(h[a/b], c)$. By Lemma B.64 we have $\mathcal{F}_{\mathcal{O}_{SMR}}(h, c)[a/b] = \mathcal{F}_{\mathcal{O}_{SMR}}(h[a/b], c)$. So, it is sufficient to show that $\mathcal{F}_{\mathcal{O}_{SMR}}(h, c) = \mathcal{F}_{\mathcal{O}_{SMR}}(h, c)[a/b]$ holds. Let $h' \in \mathcal{F}_{\mathcal{O}_{SMR}}(h, c)$. By definition, this means $h.h' \in \mathcal{S}(\mathcal{O}_{SMR})$ and $\text{frees}_{h'} \subseteq \{c\}$. Because \mathcal{O}_{SMR} never leaves accepting locations, we have $h \in \mathcal{S}(\mathcal{O}_{SMR})$. To the contrary, assume $h.h' \notin \mathcal{F}_{\mathcal{O}_{SMR}}(h, c)[a/b]$. This means $h.h'[a/b] \notin \mathcal{F}_{\mathcal{O}_{SMR}}(h, c)$. Consequently, we have $h.h'[a/b] \notin \mathcal{S}(\mathcal{O}_{SMR})$. By definition, there are steps $(l_{init}, \varphi) \xrightarrow{h} (l_1, \varphi) \xrightarrow{h[a/b]} (l_2, \varphi)$ with l_2 accepting. By $h \in \mathcal{S}(\mathcal{O}_{SMR})$, we know that l_1 is not accepting. By $\text{frees}_{h'}[a/b] = \text{frees}_{h'} \subseteq \{c\}$ and \mathcal{O}_{SMR} reaching accepting locations only via transitions labeled with $\text{free}(r)$, $r = z_a$, we must have $\varphi(z_a = c)$ in order to arrive at an accepting location. By $a \neq c \neq b$ together with \mathcal{O}_{SMR} having only one variable z_a tracking addresses, we know that \mathcal{O}_{SMR} cannot distinguish a and b so that we obtain $(l_1, \varphi) \xrightarrow{h'} (l_2, \varphi)$. Since this contradicts $h.h' \in \mathcal{S}(\mathcal{O}_{SMR})$, we must have the required $h.h' \in \mathcal{F}_{\mathcal{O}_{SMR}}(h, c)[a/b]$. The reverse inclusion follows analogously.

◇ *Ad Definition 7.14iii.* Let $a \neq b$ and $h.\text{free}(a) \in \mathcal{S}(\mathcal{O})$. Consider some φ and some program step $(l_{init}, \varphi) \xrightarrow{h} (l_1, \varphi) \xrightarrow{\text{free}(a)} (l_2, \varphi)$. Because $h.\text{free}(a) \in \mathcal{S}(\mathcal{O})$, we know l_1 is not accepting. So, the properties of \mathcal{O}_{SMR} yield $l_2 = l_1$. Hence, $\mathcal{F}_{\mathcal{O}_{SMR}}(h, b) = \mathcal{F}_{\mathcal{O}_{SMR}}(h.\text{free}(a), b)$ as required.

◇ *Ad Definition 7.14ii.* Assume $\mathcal{F}_{\mathcal{O}}(h, a) \subseteq \mathcal{F}_{\mathcal{O}}(h', a)$ and $b \in \text{fresh}_{h'}$. We establish the following: $\mathcal{F}_{\mathcal{O}_{SMR}}(h, b) \subseteq \mathcal{F}_{\mathcal{O}_{SMR}}(h', b)$. As before, it suffices to consider φ with $\varphi(z_a) = b$.

For \mathcal{O}_{HP}^k and $\mathcal{O}_{HP}^{0,1}$, note that $b \in \text{fresh}_{h'}$ immediately gives $(l_{init}, \varphi) \xrightarrow{h'} (l_{init}, \varphi)$ as l_{init} can only be left with events where the parameter is b . Moreover, l_{init} is simulation relation maximal in the corresponding SMR automaton. Then, the desired inclusion follows from Proposition 5.3 together with Lemmas B.29 and B.30.

We turn to \mathcal{O}_{EBR} . From $b \in \text{fresh}_{h'}$ we get $(l_{init}, \varphi) \xrightarrow{h'} (l, \varphi)$ with $l \in \{L_4, L_5\}$. Assume for a moment that $l = L_5$ implies $(l_{init}, \varphi) \xrightarrow{h} (l', \varphi)$ with $l' \neq L_4$. Then, we conclude the desired inclusion by Proposition 5.3 together with Lemma B.28. Now, assume $l = L_5$. It remains to show that $(l_{init}, \varphi) \xrightarrow{h} (l', \varphi)$ with $l' \neq L_4$ holds. If $h \notin \mathcal{S}(\mathcal{O}_{EBR})$, then nothing needs to be shown as the desired inclusion is trivially true. So, assume $h \in \mathcal{S}(\mathcal{O}_{EBR})$. To the contrary, assume $(l_{init}, \varphi) \xrightarrow{h} (l_{init}, \varphi)$. We construct a history $h.h_1.h_2.h_3$ as follows:

- Let h_1 be a history that contains for every thread t , $t \neq \varphi(z_t)$ an event $\text{in:enterQ}(t)$. By definition and $h \in \mathcal{S}(\mathcal{O}_{EBR})$, we get $h.h_1 \in \mathcal{S}(\mathcal{O}_{EBR})$. Moreover, $h.h_1 \in \mathcal{S}(\mathcal{O})$ because \mathcal{O}_{Base} does not react on h_1 and because $h \in \mathcal{S}(\mathcal{O}_{Base})$. Observe that we have the step $(l_{init}, \varphi') \xrightarrow{h.h_1} (l_{init}, \varphi')$ for all valuations φ' .
- By $h.h_1 \in \mathcal{S}(\mathcal{O}_{Base})$ and the definition of \mathcal{O}_{Base} , there must be $h_2 \in \{\epsilon, \text{free}(a)\}$ such that $h.h_1.h_2 \in \mathcal{S}(\mathcal{O}_{Base})$. Since \mathcal{O}_{EBR} reaches l_{init} after $h.h_1$ for all φ' as noted above, we have $h.h_1.h_2 \in \mathcal{S}(\mathcal{O})$. Observe $(L_2, \varphi') \xrightarrow{h.h_1.h_2} (L_2, \varphi')$ for $\varphi'(z_a) = a$.
- Let $h_3 = \text{retire}(\varphi(z_t), a).\text{free}(a)$. By construction, we obtain $h.h_1.h_2.h_3 \in \mathcal{S}(\mathcal{O})$.

By $h.h_1.h_2.h_3 \in \mathcal{S}(\mathcal{O})$ together with $\text{frees}_{h_1.h_2.h_3} \subseteq \{a\}$, we have $h_1.h_2.h_3 \in \mathcal{F}_{\mathcal{O}}(h, a)$. Hence, we get $h_1.h_2.h_3 \in \mathcal{F}_{\mathcal{O}}(h', a)$ by the premise. This means $h'.h_1.h_2.h_3 \in \mathcal{S}(\mathcal{O})$. However, there are the following steps for $\varphi' = \{z_t \mapsto \varphi(z_t), z_a \mapsto a\}$ on $h'.h_1.h_2.h_3$:

$$(l_{\text{init}}, \varphi') \xrightarrow{h'} (l_1, \varphi') \xrightarrow{h_1} (l_2, \varphi') \xrightarrow{h_2} (l_3, \varphi') \xrightarrow{h_3} (l_4, \varphi')$$

where (i) $l_1 \neq l_{\text{init}}$ because of $l = L_5$ and $\varphi(z_t) = \varphi'(z_t)$ and $\text{in:leaveQ}()$ taking no parameters, (ii) $l_1 = l_2$ since \mathcal{O}_{EBR} ignores $\text{in:enterQ}(\bullet)$ events of threads other than $\varphi'(z_t)$, (iii) $l_3 = l_2$, and (iv) $l_3 = L_7$. Altogether, this means $h'.h_1.h_2.h_3 \notin \mathcal{S}(\mathcal{O})$ because L_7 is accepting. Since this contradicts the assumption, we must have $l' \neq L_4$ as required. ■

Proof C.55 (Proposition 7.23). The claim follows for $\mathcal{O}_{\text{Base}} \times \mathcal{O}_{\text{EBR}}$ since enterQ and leaveQ do not take parameters. We turn to $\mathcal{O} = \mathcal{O}_{\text{Base}} \times \mathcal{O}_{\text{HP}}^0 \times \mathcal{O}_{\text{HP}}^1$. Again, unprotect_k does not take parameters and thus never races. Consider some computation $\tau.\text{act}$ with $\text{act} = \langle t, \text{in:protect}_k(p), \text{up} \rangle$ and $m_\tau(p) = c$ and $\mathcal{H}(\tau) = h$. Assume act is a racy call. Then, there are $a, b, c \in \text{Adr}$ with $a \neq c$ such that $\mathcal{F}_{\mathcal{O}}(h.\text{in:protect}_k(b), a) \not\subseteq \mathcal{F}_{\mathcal{O}}(h.\text{in:protect}_k(c), a)$. This means there is $h' \in \mathcal{F}_{\mathcal{O}}(h.\text{in:protect}_k(b), a)$ with $h' \notin \mathcal{F}_{\mathcal{O}}(h.\text{in:protect}_k(c), a)$. We have $h \in \mathcal{S}(\mathcal{O})$ because of the membership of h' . There are steps $(l_{\text{init}}, \varphi) \xrightarrow{h, h'} (l, \varphi)$ with l accepting. By definition, we have $\text{frees}_{h'} \subseteq \{a\}$. Hence, for h' to reach an accepting location, we must have $\varphi(z_a) = a$ due to the definition of \mathcal{O} . Consequently, $\text{in:protect}_k(b)$ and $\text{in:protect}_k(c)$ are indistinguishable for \mathcal{O} . Under φ , all transitions taken by the former event can be taken by the latter event as well, and vice versa. Hence, protect_k does not race. The argument is analogous in $\mathcal{O}_{\text{Base}} \times \mathcal{O}_{\text{HP}}^{0,1}$. ■

Proof C.56 (Theorem B.71). Let \mathcal{O} support elision and let $\mathcal{O}[\![P]\!]_{\text{Adr}}^\emptyset$ be MPRF and DRF. Note that this means $\mathcal{O}[\![P]\!]_{\text{Adr}}^\emptyset$ is PRF. We proceed by induction over the structure of τ . In the base case, we have $\tau = \epsilon$. Choosing $\sigma = \epsilon$ satisfies the claim. For the induction step, consider some $\tau.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$ and assume that we have already constructed $\sigma \in \mathcal{O}[\![P]\!]_{\text{Adr}}^\emptyset$ and established the following properties:

- (P1) $\tau \sim \sigma$
- (P2) $\tau < \sigma$
- (P3) $\forall a \in \text{fresh}_\sigma. \mathcal{F}_{\mathcal{O}}(\tau, a) \subseteq \mathcal{F}_{\mathcal{O}}(\sigma, a)$
- (P4) $\forall \text{pexp}, \text{qexp} \in \text{VExp}(\tau). m_\tau(\text{pexp}) \neq m_\tau(\text{qexp}) \implies m_\sigma(\text{pexp}) \neq m_\sigma(\text{qexp})$
- (P5) $\text{retired}_\tau \subseteq \text{retired}_\sigma$
- (P6) $\text{freed}_\tau \cap \text{adr}(m_\tau|_{\text{valid}_\tau}) = \emptyset$
- (P7) $\text{freed}_\tau \cap \text{retired}_\tau = \emptyset$
- (P8) τ is UAF

We construct $\hat{\sigma}$ and show the following:

- (G0) $\hat{\sigma} \in \mathcal{O}[\![P]\!]_{\text{Adr}}^\emptyset$
- (G1) $\tau.\text{act} \sim \hat{\sigma}$
- (G2) $\tau.\text{act} < \hat{\sigma}$

- (G3) $\forall a \in \text{fresh}_{\hat{\sigma}}. \mathcal{F}_O(\tau.\text{act}, a) \subseteq \mathcal{F}_O(\hat{\sigma}, a)$
- (G4) $\forall pexp, qexp \in \text{VExp}(\tau.\text{act}). \left(\begin{array}{l} m_{\tau.\text{act}}(pexp) \neq m_{\tau.\text{act}}(qexp) \\ \implies m_{\hat{\sigma}}(pexp) \neq m_{\hat{\sigma}}(qexp) \end{array} \right)$
- (G5) $\text{retired}_{\tau.\text{act}} \subseteq \text{retired}_{\hat{\sigma}}$
- (G6) $\text{freed}_{\tau.\text{act}} \cap \text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) = \emptyset$
- (G7) $\text{freed}_{\tau.\text{act}} \cap \text{retired}_{\tau.\text{act}} = \emptyset$
- (G8) $\tau.\text{act}$ is UAF

Let $\text{act} = \langle t, \text{com}, \text{up} \rangle$. We do a case distinction over com .

◇ **Case 1:** com is an assignment

Our goal is to find some act' such that $\hat{\sigma} = \sigma.\text{act}'$ satisfies the requirements. We obtain act' from an invocation of Lemma B.67. That is, it remains to show that Lemma B.67 is enabled. To that end, we have to show: if com contains $p.\text{sel}$ then $p \in \text{valid}_{\tau}$. So, assume com contains $p.\text{sel}$ as nothing needs to be shown otherwise. We proceed as follows: we show that com is enabled after σ and then use pointer race freedom to establish the validity of p .

- ◇ *Ad (G0) to (G2).* By Lemma B.50 we have $m_{\sigma}(p) \neq \perp$. That is, $m_{\sigma}(p) \in \text{Adr} \cup \{\text{seg}\}$. Towards a contradiction, assume $m_{\sigma}(p) = \text{seg}$. By Lemma B.49, we have $p \in \text{valid}_{\sigma}$. Then, $m_{\tau}(p) = \text{seg}$ follows from $\tau \sim \sigma$. This means act is not enabled after τ . This contradicts $\tau.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\text{Adr}}$. Hence, $m_{\sigma}(p) \neq \text{seg}$ must hold. So, com is enabled after σ . This means there is an update up' such that $\text{act}'' = \langle t, \text{com}, \text{up}' \rangle$ is enabled after σ , that is, $\sigma.\text{act}'' \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\emptyset}$. Since $\mathcal{O}[\![P]\!]_{\text{Adr}}^{\emptyset}$ is free from pointer races by assumption, we must have $p \in \text{valid}_{\tau}$. We get $\tau \in \text{valid}_{\tau}$ from $\tau \sim \sigma$ together with Lemma B.37. Altogether, Lemma B.67 is enabled. An invocation for $\tau.\text{act}$ and σ yields act' such that $\hat{\sigma} = \sigma.\text{act}'$ satisfies $\hat{\sigma} \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\emptyset}$ as well as $\tau.\text{act} \sim \hat{\sigma}$ and $\tau.\text{act} \leq \hat{\sigma}$.
- ◇ *Ad (G3).* Let $a \in \text{fresh}_{\sigma.\text{act}'}$. By definition, $a \in \text{fresh}_{\sigma}$. We obtain $\mathcal{F}_O(\tau, a) \subseteq \mathcal{F}_O(\sigma, a)$ from (P3). Then, the desired $\mathcal{F}_O(\tau.\text{act}, a) \subseteq \mathcal{F}_O(\sigma.\text{act}', a)$ follows from act/act' not emitting an event, thus $\mathcal{F}_O(\tau.\text{act}, a) = \mathcal{F}_O(\tau, a)$ and $\mathcal{F}_O(\sigma.\text{act}', a) = \mathcal{F}_O(\sigma, a)$ by definition.
- ◇ *Ad (G4).* Consider $pexp, qexp \in \text{VExp}(\tau.\text{act})$ with $m_{\tau.\text{act}}(pexp) \neq m_{\tau.\text{act}}(qexp)$. We focus on the case $\text{com} \equiv p := q.\text{next}$; the other cases follow analogously. Let $a = m_{\tau}(q)$. By the semantics, $a \neq \text{seg}$. Let $b = m_{\tau}(a.\text{next})$. Then, $\text{up} = [p \mapsto b]$. Because $\sigma.\text{act}'$ is PRF, we have $q \in \text{valid}_{\sigma}$. Hence, $m_{\sigma}(q) = a$ by (P1). Let $c = m_{\sigma}(a.\text{next})$. Then, we have $\text{up}' = [p \mapsto c]$. If $pexp \neq p \neq qexp$, then we obtain $m_{\tau.\text{act}}(pexp) = m_{\tau}(pexp)$ as well $m_{\sigma.\text{act}'}(pexp) = m_{\sigma}(pexp)$ as well as $pexp \in \text{VExp}(\tau)$, and similarly for $qexp$, so that we conclude by (P4). So assume now wlog. $pexp \equiv p$. This means, $qexp \neq q$. We arrive at $m_{\tau}(a.\text{next}) = m_{\tau.\text{act}}(p) \neq m_{\tau.\text{act}}(qexp) = m_{\tau}(qexp)$. Furthermore, $q \in \text{VExp}(\tau)$ holds by definition and $qexp \in \text{VExp}(\tau)$ holds by $\text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) \subseteq \text{adr}(m_{\tau}|_{\text{valid}_{\tau}})$ from Lemma B.47. Then, (P4) yields $m_{\sigma}(a.\text{next}) \neq m_{\sigma}(qexp)$. From the update up' we get $m_{\sigma.\text{act}'}(p) = m_{\sigma}(a.\text{next})$ and $m_{\sigma}(qexp) = m_{\sigma.\text{act}'}(qexp)$. This concludes the desired inequality $m_{\sigma.\text{act}'}(p) \neq m_{\sigma.\text{act}'}(qexp)$.
- ◇ *Ad (G5).* We conclude by (P5), $\text{retired}_{\tau} = \text{retired}_{\tau.\text{act}}$, and $\text{retired}_{\sigma} = \text{retired}_{\sigma.\text{act}'}$.

- ◇ *Ad (G6).* We have $\text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) \subseteq \text{adr}(m_{\tau}|_{\text{valid}_{\tau}})$ by Lemma B.47. Then, we obtain the desired $\text{freed}_{\tau.\text{act}} \cap \text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) = \emptyset$ by (P6) together with $\text{freed}_{\tau} = \text{freed}_{\tau.\text{act}}$.
- ◇ *Ad (G7).* We conclude by (P7) as well as $\text{freed}_{\tau.\text{act}} = \text{freed}_{\tau}$ and $\text{retired}_{\tau.\text{act}} = \text{retired}_{\tau}$.
- ◇ *Ad (G8).* We have already shown that if com contains $p.\text{sel}$ then $p \in \text{valid}_{\tau}$. This means that $\tau.\text{act}$ does not perform an unsafe access. So $\tau.\text{act}$ is UAF by (P8).

◇ **Case 2:** $\text{com} \equiv \text{in:func}(r_1, \dots, r_n)$

The update is $\text{up} = \emptyset$. We show that $\hat{\sigma} = \sigma.\text{act}$ is an adequate choice.

- ◇ *Ad (G0).* We show that act is enabled after σ . By Lemma B.50, $m_{\sigma}(r_i) \in \text{Adr} \cup \{\text{seg}\}$. To the contrary, assume $m_{\sigma}(r_i) = \text{seg}$ for some i . Then, $r_i \in \text{valid}_{\sigma}$ by Lemma B.49. So, we get $m_{\tau}(r_i) = \text{seg}$ by $\tau \sim \sigma$. As this contradicts enabledness of act after τ , $m_{\sigma}(r_i) \neq \text{seg}$ must hold. Altogether, this means $\sigma.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Adr}}^{\emptyset}$.
- ◇ *Ad (G2) and (G3).* Let the parameters to the call be $\bar{r} = r_1, \dots, r_n$. Moreover, let the actual arguments to the call be $m_{\tau}(\bar{r}) = \bar{v}_{\tau} = v_{\tau,1}, \dots, v_{\tau,n}$ and $m_{\sigma}(\bar{r}) = \bar{v}_{\sigma} = v_{\sigma,1}, \dots, v_{\sigma,n}$. We show:

$$\forall b \in \text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) \cup \text{fresh}_{\sigma.\text{act}}. \mathcal{F}_{\mathcal{O}}(\tau.\text{act}, b) \subseteq \mathcal{F}_{\mathcal{O}}(\sigma.\text{act}, b).$$

Consider some $b \in \text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) \cup \text{fresh}_{\sigma.\text{act}}$. We obtain $b \in \text{adr}(m_{\tau}|_{\text{valid}_{\tau}}) \cup \text{fresh}_{\sigma}$ as before. Now, (P2) and (P3) give $\mathcal{F}_{\mathcal{O}}(\tau, b) \subseteq \mathcal{F}_{\mathcal{O}}(\sigma, b)$. Then, Lemma B.41 yields:

$$\mathcal{F}_{\mathcal{O}}(\mathcal{H}(\tau).\text{in:func}(t, \bar{v}_{\tau}), b) \subseteq \mathcal{F}_{\mathcal{O}}(\mathcal{H}(\sigma).\text{in:func}(t, \bar{v}_{\sigma}), b). \quad (26)$$

Next, we use the fact that $\sigma.\text{act}$ is PRF by assumption and (G0). That is, act is not racy:

$$\mathcal{F}_{\mathcal{O}}(\mathcal{H}(\sigma).\text{in:func}(t, \bar{v}_{\tau}), b) \subseteq \mathcal{F}_{\mathcal{O}}(\mathcal{H}(\sigma).\text{in:func}(t, \bar{v}_{\sigma}), b). \quad (27)$$

To see this, we have to show that the following holds for all i with $1 \leq i \leq n$:

$$(v_{\sigma,i} = b \vee r_i \in \text{valid}_{\sigma} \vee r_i \in \text{DExp}) \implies v_{\tau,i} = v_{\sigma,i}.$$

If $r_i \in \text{valid}_{\sigma}$, then $v_{\sigma,i} = m_{\sigma}(r_i) = m_{\tau}(r_i) = v_{\tau,i}$ because of $\tau \sim \sigma$. Similarly, $v_{\sigma,i} = v_{\tau,i}$ if $r_i \in \text{DExp}$ since this means $r_i \in \text{DVar} \subseteq \text{dom}(m_{\sigma}|_{\text{valid}_{\sigma}})$. Consider now $v_{\sigma,i} = b$ and $r_i \notin \text{valid}_{\sigma} \cup \text{DExp}$. There are two cases: $b \in \text{adr}(m_{\tau}|_{\text{valid}_{\tau}})$ or $b \in \text{fresh}_{\sigma}$. In the former case, we have $b \in \text{adr}(m_{\sigma}|_{\text{valid}_{\sigma}})$ by (P1). Moreover, $b \in m_{\sigma}(\text{PVar} \setminus \text{valid}_{\sigma})$ follows from $r_i \notin \text{valid}_{\sigma}$ and $m_{\sigma}(r_i) = v_{\sigma,i} = b$. Then, Lemma B.53 yields $b \in \emptyset$. Hence, the case cannot apply. Consider the latter case, $b \in \text{fresh}_{\sigma}$. This means $m_{\sigma}(r_i) \in \text{fresh}_{\sigma}$. Since this contradicts Lemma B.42, the case cannot apply either. Altogether, this proves the desired implication and establishes (27). Combining (26) and (27), we conclude by:

$$\begin{aligned} \mathcal{F}_{\mathcal{O}}(\tau.\text{act}, b) &= \mathcal{F}_{\mathcal{O}}(\mathcal{H}(\tau).\text{in:func}(t, \bar{v}_{\tau}), b) \subseteq \mathcal{F}_{\mathcal{O}}(\mathcal{H}(\sigma).\text{in:func}(t, \bar{v}_{\tau}), b) \\ &\subseteq \mathcal{F}_{\mathcal{O}}(\mathcal{H}(\sigma).\text{in:func}(t, \bar{v}_{\sigma}), b) = \mathcal{F}_{\mathcal{O}}(\sigma.\text{act}, b). \end{aligned}$$

◇ *Ad (G1).* We have $m_\tau = m_{\tau.act}$ and $valid_\tau = valid_{\tau.act}$ by definition, and similarly for σ . Hence, the desired $\tau.act \preceq_\emptyset \sigma.act$ follows from (P1).

◇ *Ad (G4).* Follows by (P4) and $m_\tau = m_{\tau.act}$, $m_\sigma = m_{\sigma.act}$, and $VExp(\tau) = VExp(\tau.act)$.

◇ *Ad (G5).* If $com \neq in:retire(p)$, we conclude by (P5) as well as $retired_\tau = retired_{\tau.act}$ and $retired_\sigma = retired_{\sigma.act}$. So consider $com \equiv in:retire(p)$. Towards a contradiction, assume $p \notin valid_\sigma$. By Lemma B.51, this means $m_\sigma(p) \in frees_\sigma \subseteq freed_\sigma$ where the inclusion holds because no addresses are reallocated in σ . Then, $m_{\sigma.act}(p) \in freed_{\sigma.act}$ and $m_{\sigma.act}(p) \in retired_{\sigma.act}$. Lemma B.70 now states that $\mathcal{O}[[P]]_{Adr}^\emptyset$ contains a double retire. Since this contradicts the assumption, we must have $p \in valid_\sigma$. Hence, obtain that $m_\tau(p) = m_\sigma(p) = a$ for some a . We conclude by (P5):

$$retired_{\tau.act} = retired_\tau \cup \{a\} \subseteq retired_\sigma \cup \{a\} = retired_{\sigma.act}.$$

◇ *Ad (G6).* We have $adr(m_\tau|_{valid_\tau}) = adr(m_{\tau.act}|_{valid_{\tau.act}})$ since act does not affect the memory nor the validity, as noted before. Combined with $freed_\tau = freed_{\tau.act}$ and (P6) implies (G6).

◇ *Ad (G7).* If $com \neq retire(p)$, nothing needs to be shown and we conclude by (P7). So consider now the case $com \equiv retire(p)$. We first show $p \in valid_\sigma$. To the contrary, assume $p \notin valid_\sigma$. Let $m_\sigma(p) = a$. From Lemma B.52 we obtain $a \in freed_\sigma$ and thus get $a \in freed_{\sigma.act}$ by definition. Moreover, we obtain $a \in retired_{\sigma.act}$. Then, Lemma B.70, which is enabled by (G0), states that $\mathcal{O}[[P]]_{Adr}^\emptyset$ contains a double retire. This contradicts the assumption. Hence, $p \in valid_\sigma$ must hold. By (G1), this means $p \in valid_\tau$ as well as $m_\tau(p) = a$. That is, $a \in m_\tau(valid_\tau) \subseteq adr(m_\tau|_{valid_\tau})$ where the inclusion follows from Lemma B.35. Now, (P6) yields $a \notin freed_\tau$. Altogether, this gives the desired $a \notin freed_{\tau.act}$ by definition.

◇ *Ad (G8).* Follows from (P8) together with act not raising unsafe accesses.

◇ **Case 3:** $com \equiv re:func$

We choose $\hat{\sigma} = \sigma.act$. By $\tau \sim \sigma$, we know that act is enabled after σ , that is, $\sigma.act \in \mathcal{O}[[P]]_{Adr}^\emptyset$. By definition, act emits the same event $re:func(t)$ after both τ and σ . Then, the claim follows similarly to the previous case.

◇ **Case 4:** $com \equiv p := malloc$ and $m_{\tau.act}(p) \in fresh_\sigma$

Let $a = m_{\tau.act}(p)$. The update is $up = [p \mapsto a, a.next \mapsto seg, a.data \mapsto d]$ for some d . By definition, we have $a \in fresh_\tau \cup freed_\tau$. By assumption, we have $a \in fresh_\sigma$. We now show that $\hat{\sigma} = \sigma.act$ is an adequate choice.

◇ *Ad (G0) to (G2).* From $a \in fresh_\sigma$ we immediately get $\sigma.act \in \mathcal{O}[[P]]_{Adr}^\emptyset$. Then, (G0) to (G2) follow from (P1) and (P2) together with Lemma B.68. Note that Lemma B.68 is applicable because we get $\mathcal{F}_\mathcal{O}(\tau, a) \subseteq \mathcal{F}_\mathcal{O}(\sigma, a)$ from $a \in fresh_\sigma$ together with (P3).

◇ *Ad (G3).* Follows from $fresh_{\sigma.act} \subseteq fresh_\sigma$ together with the fact that act does not emit an event, i.e., $\mathcal{H}(\tau.act) = \mathcal{H}(\tau)$ and $\mathcal{H}(\sigma.act) = \mathcal{H}(\sigma)$.

◇ *Ad (G4).* Consider $pexp, qexp \in VExp(\tau.act)$ with $m_{\tau.act}(pexp) \neq m_{\tau.act}(qexp)$. By definition, we have $VExp(\tau.act) = VExp(\tau) \cup \{a.next\}$.

◇ **Case 4.1:** $pexp, pexp \notin \{p, a.next\}$

We get $m_{\tau.act}(pexp) = m_{\tau}(pexp)$ and $m_{\sigma.act'}(pexp) = m_{\sigma}(pexp)$ and $pexp \in VExp(\tau)$, and similarly for $qexp$, so that we conclude by (P4).

◇ **Case 4.2:** $pexp \equiv p$

Then, $qexp \not\equiv p$. Because $a \in fresh_{\sigma}$, Lemma B.42 yields $m_{\sigma.act}(pexp) \neq m_{\sigma}(qexp)$. Furthermore, $m_{\sigma.act}(pexp) \neq seg$. Hence, we arrive at $m_{\sigma.act}(pexp) \neq m_{\sigma.act}(qexp)$ because we have $m_{\sigma.act}(qexp) \in \{seg, m_{\sigma}(qexp)\}$ by definition.

◇ **Case 4.3:** $pexp \equiv a.next$ and $pexp \equiv p$

By definition, $m_{\sigma.act}(pexp) \neq m_{\sigma.act}(qexp)$.

◇ **Case 4.4:** $pexp \equiv a.next$ and $pexp \not\equiv p$

We have $m_{\tau.act}(pexp) = m_{\sigma.act}(pexp) = seg$. Towards a contradiction, assume that $m_{\sigma.act}(qexp) = seg$ holds. By $qexp \in VExp(\tau.act)$, we have $qexp \in PVar$ or $qexp \equiv b.next \wedge b \in m_{\tau.act}(valid_{\tau.act})$. By (P1) this means that we have either $qexp \in PVar$ or $qexp \equiv b.next \wedge b \in m_{\sigma.act}(valid_{\sigma.act})$. From Lemma B.49 we get $qexp \in valid_{\sigma.act}$. So, $m_{\tau.act}(qexp) = seg$ by (P1). Since this contradicts the choice of $m_{\tau.act}(pexp) \neq m_{\tau.act}(qexp)$, we must have $m_{\sigma.act}(qexp) \neq seg$ as required.

◇ *Ad (G5).* We conclude by (P5), $retired_{\tau} = retired_{\tau.act}$, and $retired_{\sigma} = retired_{\sigma.act}$.

◇ *Ad (G6).* By definition, we have $valid_{\tau.act} = valid_{\tau} \cup \{p, a.next\}$. Consequently, we get $valid_{\tau.act} \cap Adr = (valid_{\tau} \cap Adr) \cup \{a\}$. From an invocation of Lemma B.47 we then get $adr(m_{\tau.act}|_{valid_{\tau.act}}) \subseteq adr(m_{\tau}|_{valid_{\tau}}) \cup \{a\}$. Combined with $freed_{\tau.act} = freed_{\tau} \setminus \{a\}$, we conclude by (P6).

◇ *Ad (G7).* We have $freed_{\tau.act} \subseteq freed_{\tau}$ and $retired_{\tau.act} = retired_{\tau}$. We conclude by (P7).

◇ *Ad (G8).* Follows from (P8) together with act not raising unsafe accesses.

◇ **Case 5:** $com \equiv p := malloc$ and $m_{\tau.act}(p) \notin fresh_{\sigma}$

Let $a = m_{\tau.act}(p)$. The update is $up = [p \mapsto a, a.next \mapsto seg, a.data \mapsto d]$ for some datum d . By definition, we have $a \in fresh_{\tau} \cup freed_{\tau}$. If $a \in freed_{\tau}$, then $a \notin adr(m_{\tau}|_{valid_{\tau}})$ by induction. Otherwise, $a \in fresh_{\tau}$ and we get $a \notin adr(m_{\tau}|_{valid_{\tau}})$ by Lemmas B.35 and B.43. Then, $\tau \sim \sigma$ together with Lemma B.38 yields $a \notin adr(m_{\sigma}|_{valid_{\sigma}})$. Now, we invoke Lemma B.66 for σ and a . This gives $\gamma \in \mathcal{O}[P]_{Addr}^{\emptyset}$ with $\sigma \sim \gamma$, $\sigma < \gamma$, $a \in fresh_{\gamma}$, and $retired_{\sigma} \subseteq retired_{\gamma} \cup \{a\}$. Furthermore, the lemma results in $m_{\sigma}(pexp) \neq m_{\sigma}(pexp) \implies m_{\gamma}(pexp) \neq m_{\gamma}(pexp)$, for all $pexp, qexp \in VExp(\sigma)$. Since $a \notin fresh_{\sigma}$ by assumption, we get $\mathcal{F}_{\mathcal{O}}(\sigma, b) = \mathcal{F}_{\mathcal{O}}(\gamma, b)$ for all $b \in fresh_{\gamma} \setminus \{a\}$.

◇ *Ad $\tau \sim \gamma$ and $\tau < \gamma$.* From the above, we have $\tau \sim \sigma \sim \gamma$ and $\tau < \sigma < \gamma$. By definition and Lemma B.38, we get $m_{\tau}(valid_{\tau}) \subseteq m_{\sigma}(valid_{\sigma})$ and $adr(m_{\tau}|_{valid_{\tau}}) = adr(m_{\sigma}|_{valid_{\sigma}})$. Lemmas B.32 to B.34 yield $\tau \sim \gamma$ and $\tau < \gamma$.

◇ *Ad $\mathcal{F}_{\mathcal{O}}(\tau, b) \subseteq \mathcal{F}_{\mathcal{O}}(\gamma, b)$ for all $b \in fresh_{\gamma}$.* Let $b \in fresh_{\gamma}$. If $b \neq a$ holds, we immediately get $\mathcal{F}_{\mathcal{O}}(\tau, b) \subseteq \mathcal{F}_{\mathcal{O}}(\sigma, b) = \mathcal{F}_{\mathcal{O}}(\gamma, b)$ where the inclusion is due to (P3) and the equality due to the correlation between σ and γ from above. Otherwise, $b = a$. Then, the desired inclusion follows from Lemma B.57 for τ and γ .

We need to show that Lemma B.57 is enabled. We already have $a \in \text{fresh}_\gamma$. Moreover, we have $\mathcal{F}_\mathcal{O}(\tau, c) \subseteq \mathcal{F}_\mathcal{O}(\sigma, c) = \mathcal{F}_\mathcal{O}(\gamma, c)$ for any $c \in \text{fresh}_\sigma \cap \text{fresh}_\gamma$ due to (P3) and the correlation of σ and γ . Lastly, we observe that $a \notin \text{retired}_\tau$. This follows from (P7) if $a \in \text{freed}_\tau$ and from Lemma B.42 if $a \in \text{fresh}_\tau$. Hence, $\mathcal{F}_\mathcal{O}(\tau, b) \subseteq \mathcal{F}_\mathcal{O}(\gamma, b)$.

- ◇ *Ad $m_\tau(\text{pexp}) \neq m_\tau(\text{qexp}) \implies m_\gamma(\text{pexp}) \neq m_\gamma(\text{qexp})$ for all $\text{pexp}, \text{qexp} \in \text{VExp}(\tau)$.* Let some $\text{pexp}, \text{qexp} \in \text{VExp}(\tau)$ with $m_\tau(\text{pexp}) \neq m_\tau(\text{qexp})$. By (P4), $m_\sigma(\text{pexp}) \neq m_\sigma(\text{qexp})$. So, $\text{pexp}, \text{qexp} \in \text{VExp}(\sigma)$ by (P1). Then, the properties of γ give $m_\gamma(\text{pexp}) \neq m_\gamma(\text{qexp})$.
- ◇ *Ad $\text{retired}_\tau \subseteq \text{retired}_\gamma$.* We have $\text{retired}_\tau \subseteq \text{retired}_\sigma \subseteq \text{retired}_\gamma \cup \{a\}$ where the first inclusion is due to (P5) and the second inclusion holds by the properties of γ . It remains to show that $a \notin \text{retired}_\tau$ holds. As before, this follows from (P7) together with $a \in \text{fresh}_\tau$ and Lemma B.42.

With the above properties in place, observe that the induction hypothesis could have given us γ instead of σ because γ satisfies the necessary properties (P1) to (P8). Further, $a \in \text{fresh}_\sigma$. Hence, we conclude as in the previous case.

◇ **Case 6:** $\text{com} \equiv \text{free}(a)$ and $\mathcal{H}(\sigma).\text{free}(a) \in \mathcal{S}(\mathcal{O})$

The update is $\text{up} = \emptyset$. By definition, $\mathcal{H}(\tau).\text{free}(a) \in \mathcal{S}(\mathcal{O})$. That is, $\text{free}(a) \in \mathcal{F}_\mathcal{O}(\tau, a)$. By assumption, $\mathcal{H}(\sigma).\text{free}(a) \in \mathcal{S}(\mathcal{O})$. That is, $\text{free}(a) \in \mathcal{F}_\mathcal{O}(\sigma, a)$. We choose $\hat{\sigma} = \sigma.\text{act}$.

- ◇ *Ad (G0) to (G2).* By $\mathcal{H}(\sigma).\text{free}(a) \in \mathcal{S}(\mathcal{O})$, we have $\sigma.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Adr}}^\emptyset$. Then, we conclude by Lemma B.68, which is enabled according to Lemma B.56.
- ◇ *Ad (G3).* Let $b \in \text{fresh}_{\sigma.\text{act}}$. By definition, $b \in \text{fresh}_\sigma \setminus \{a\}$. That is, $a \neq b$. Lemma B.56 yields $\mathcal{F}_\mathcal{O}(\tau.\text{act}, b) = \mathcal{F}_\mathcal{O}(\tau, b)$ and $\mathcal{F}_\mathcal{O}(\sigma.\text{act}, b) = \mathcal{F}_\mathcal{O}(\sigma, b)$. We conclude by (P3).
- ◇ *Ad (G4).* Let $\text{pexp}, \text{qexp} \in \text{VExp}(\tau.\text{act})$. By definition, $\text{valid}_{\tau.\text{act}} \subseteq \text{valid}_\tau$ and $m_{\tau.\text{act}} = m_\tau$. Hence, we have $\text{pexp}, \text{qexp} \in \text{VExp}(\tau)$. Moreover, $m_{\sigma.\text{act}} = m_\sigma$. We conclude by (G4).
- ◇ *Ad (G5).* By (P5) we get $\text{retired}_{\tau.\text{act}} = \text{retired}_\tau \setminus \{a\} = \text{retired}_\sigma \setminus \{a\} = \text{retired}_\sigma$.
- ◇ *Ad (G6).* We have $\text{freed}_{\tau.\text{act}} = \text{freed}_\tau \cup \{a\}$. Moreover, $\text{valid}_{\tau.\text{act}} \subseteq \text{valid}_\tau$ by definition. By Lemma B.47, $\text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) \subseteq \text{adr}(m_\tau|_{\text{valid}_\tau})$. In order to conclude by (P6), it remains to show $a \notin \text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}})$. Since $\sigma.\text{act}$ PRF and $a \in \text{freed}_{\sigma.\text{act}}$, Lemmas B.35 and B.48 yield $a \notin \text{adr}(m_{\sigma.\text{act}}|_{\text{valid}_{\sigma.\text{act}}})$. Then, the desired $a \notin \text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}})$ follows from (G1) together with Lemma B.38.
- ◇ *Ad (G7).* By definition, $\text{freed}_{\tau.\text{act}} = \text{freed}_\tau \cup \{a\}$ and $\text{retired}_{\tau.\text{act}} = \text{retired}_\tau \setminus \{a\}$. Then, we conclude by (P7).
- ◇ *Ad (G8).* Follows from (P8) together with act not raising unsafe accesses.

◇ **Case 7:** $com \equiv \text{free}(a)$ and $\mathcal{H}(\sigma).\text{free}(a) \notin \mathcal{S}(\mathcal{O})$

The update is $up = \emptyset$. By definition, $\mathcal{H}(\tau).\text{free}(a) \in \mathcal{S}(\mathcal{O})$. That is, $\text{free}(a) \in \mathcal{F}_{\mathcal{O}}(\tau, a)$. By assumption, we have $\text{free}(a) \notin \mathcal{F}_{\mathcal{O}}(\sigma, a)$. By $\tau < \sigma$ from (G2) we get $a \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$ as otherwise we had $\text{free}(a) \in \mathcal{F}_{\mathcal{O}}(\sigma, a)$. By (G1), this means $a \notin m_\sigma(\text{valid}_\sigma)$. We now show that $\hat{\sigma} = \sigma$ is an adequate choice.

◇ *Ad (G0).* By induction hypothesis, $\sigma \in \mathcal{O}[[P]]_{\text{Adr}}^\emptyset$.

◇ *Ad (G1).* By definition, $\text{ctrl}(\tau) = \text{ctrl}(\tau.\text{act})$. So, $\text{ctrl}(\tau.\text{act}) = \text{ctrl}(\sigma)$ follows from (P1). To show $m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}} = m_\sigma|_{\text{valid}_\sigma}$ it suffices to show $m_\tau|_{\text{valid}_\tau} = m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}$ by (P1). To arrive there, we first show $\text{valid}_\tau = \text{valid}_{\tau.\text{act}}$. The inclusion $\text{valid}_{\tau.\text{act}} \subseteq \text{valid}_\tau$ holds by definition. To see $\text{valid}_\tau \subseteq \text{valid}_{\tau.\text{act}}$, consider $pexp \in \text{valid}_\tau$. Then, $pexp \neq a.\text{next}$ must hold as for otherwise we had $a \in \text{adr}(m_\tau|_{\text{valid}_\tau})$ by Lemma B.35 which does not hold as shown before. Moreover, we must have $m_\tau(pexp) \neq a$ as otherwise we would again get $a \in \text{adr}(m_\tau|_{\text{valid}_\tau})$ by Lemma B.35. Hence, by the definition of validity, we have $pexp \in \text{valid}_{\tau.\text{act}}$. Altogether, this means we have $\text{valid}_\tau = \text{valid}_{\tau.\text{act}}$ indeed. Then, the desired $m_\tau|_{\text{valid}_\tau} = m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}$ follows because $m_\tau = m_{\tau.\text{act}}$ due to $up = \emptyset$.

◇ *Ad (G2).* Let $b \in \text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}})$. We get $b \in \text{adr}(m_\tau|_{\text{valid}_\tau})$ along the same lines as in (G1). This means that $a \neq b$. Moreover, by $\tau < \sigma$ we have $\mathcal{F}_{\mathcal{O}}(\tau, b) \subseteq \mathcal{F}_{\mathcal{O}}(\sigma, b)$. So it suffices to show $\mathcal{F}_{\mathcal{O}}(\tau, b) = \mathcal{F}_{\mathcal{O}}(\tau.\text{act}, b)$. This follows from Lemma B.56.

◇ *Ad (G3).* Let $b \in \text{fresh}_\sigma$. If $a \neq b$, we get $\mathcal{F}_{\mathcal{O}}(\tau.\text{act}, b) \subseteq \mathcal{F}_{\mathcal{O}}(\tau, b)$ from Lemma B.56 and then conclude by (P3). Now, consider $a = b$. As argued above, $\text{free}(a) \in \mathcal{F}_{\mathcal{O}}(\tau, a)$. Moreover, (P3) gives $\mathcal{F}_{\mathcal{O}}(\tau, b) \subseteq \mathcal{F}_{\mathcal{O}}(\sigma, b)$. That is, we have $\text{free}(a) \in \mathcal{F}_{\mathcal{O}}(\sigma, a)$. Since this contradicts $\text{free}(a) \notin \mathcal{F}_{\mathcal{O}}(\sigma, a)$ from above, the case cannot apply.

◇ *Ad (G4).* Let $pexp, qexp \in \text{VExp}(\tau.\text{act})$. By definition, $\text{valid}_{\tau.\text{act}} \subseteq \text{valid}_\tau$ and $m_{\tau.\text{act}} = m_\tau$. Hence, we have $pexp, qexp \in \text{VExp}(\tau)$. Moreover, $m_{\sigma.\text{act}} = m_\sigma$. We conclude by (G4).

◇ *Ad (G5).* We conclude by (P5) and $\text{retired}_{\tau.\text{act}} = \text{retired}_\tau \setminus \{a\}$.

◇ *Ad (G6).* For (G1) we already showed $\text{valid}_\tau = \text{valid}_{\tau.\text{act}}$. Together with $m_{\tau.\text{act}} = m_\tau$, we obtain $\text{adr}(m_\tau|_{\text{valid}_\tau}) = \text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}})$ by Lemma B.35. We know $a \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$ from above. Moreover, $\text{freed}_{\tau.\text{act}} = \text{freed}_\tau \cup \{a\}$. Hence, we conclude by (G6).

◇ *Ad (G7).* By definition, $\text{freed}_{\tau.\text{act}} = \text{freed}_\tau \cup \{a\}$ and $\text{retired}_{\tau.\text{act}} = \text{retired}_\tau \setminus \{a\}$. Then, we conclude by (P7).

◇ *Ad (G8).* Follows from (P8) together with act not raising unsafe accesses.

◇ **Case 8:** $com \equiv \text{env}(a)$

The update is $up = [a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d]$ for some d . By definition, $a \in \text{fresh}_\tau \cup \text{freed}_\tau$. This implies

$a \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$: if $a \in \text{freed}_\tau$ it follows from (P6) and otherwise from Lemmas B.35 and B.43. We show that $\hat{\sigma} = \sigma$ is an adequate choice. We have:

$$\begin{aligned} \text{dom}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) &= \text{valid}_{\tau.\text{act}} \cup \text{DVar} \cup \{ b.\text{data} \mid b \in m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}}) \} \\ &= \text{valid}_\tau \cup \text{DVar} \cup \{ b.\text{data} \mid b \in m_{\tau.\text{act}}(\text{valid}_\tau) \} \\ &= \text{valid}_\tau \cup \text{DVar} \cup \{ b.\text{data} \mid b \in m_\tau(\text{valid}_\tau) \} = \text{dom}(m_\tau|_{\text{valid}_\tau}) \end{aligned}$$

where the first and last equality is by definition, the second equality by $\text{valid}_{\tau.\text{act}} = \text{valid}_\tau$, and the third equality because $a.\text{next} \notin \text{valid}_\tau$ by $a \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$ together with Lemma B.35. Similarly, $a.\text{data} \notin \text{valid}_\tau$ because $a \notin m_\tau(\text{valid}_\tau)$ by $a \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$ and Lemma B.35. Altogether, $m_\tau|_{\text{valid}_\tau} = m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}$ and $m_\tau(\text{valid}_\tau) = m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}})$ and $\text{adr}(m_\tau|_{\text{valid}_\tau}) = \text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}})$. Furthermore, we have $\text{ctrl}(\tau) = \text{ctrl}(\tau.\text{act})$ and $\text{valid}_\tau = \text{valid}_{\tau.\text{act}}$ as well as $\text{fresh}_\tau = \text{fresh}_{\tau.\text{act}}$ and $\text{freed}_\tau = \text{freed}_{\tau.\text{act}}$. Hence, (G1) to (G3) and (G5) to (G7) follow immediately from (P1) to (P3) and (P5) to (P7). Next, we establish (G4). To that end, consider $\text{pexp}, \text{qexp} \in \text{VExp}(\tau.\text{act})$ with $m_{\tau.\text{act}}(\text{pexp}) \neq m_{\tau.\text{act}}(\text{qexp})$. We have $\text{pexp}, \text{qexp} \in \text{VExp}(\tau)$ by definition. Note that we must have $\text{pexp} \neq a.\text{next} \neq \text{qexp}$ because of $a \notin \text{adr}(m_\tau|_{\text{valid}_\tau})$. Hence, we obtain $m_{\tau.\text{act}}(\text{pexp}) = m_\tau(\text{pexp})$. Then, (G4) follows from (P4). Finally, the remaining (G8) follow from (P8) since act does not raise unsafe accesses.

◇ **Case 9:** $\text{com} \equiv \text{assume } \text{cond}$ and $\sigma.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Adr}}^\emptyset$

The update is $up = \emptyset$. We show that $\hat{\sigma} = \sigma.\text{act}$ is an adequate choice. To that end, observe that we have $m_\tau(\text{valid}_\tau) = m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}})$ and $\text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) = \text{adr}(m_\tau|_{\text{valid}_\tau})$ by Lemma B.47.

- ◇ *Ad (G0).* Holds by assumption.
- ◇ *Ad (G1) and (G2).* Follow from Lemma B.68 together with (P1) and (P2).
- ◇ *Ad (G3).* Follows from (P3) together with $\text{fresh}_\sigma = \text{fresh}_{\sigma.\text{act}}$ as well as $\mathcal{H}(\tau) = \mathcal{H}(\tau.\text{act})$ and $\mathcal{H}(\sigma) = \mathcal{H}(\sigma.\text{act})$.
- ◇ *Ad (G4).* Follows by (G0) since $m_\tau = m_{\tau.\text{act}}$ and $m_\sigma = m_{\sigma.\text{act}}$ and $\text{VExp}(\tau.\text{act}) = \text{VExp}(\tau)$ by the above $m_\tau(\text{valid}_\tau) = m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}})$.
- ◇ *Ad (G5).* Follows from (G5), $\text{retired}_{\tau.\text{act}} = \text{retired}_\tau$, and $\text{retired}_{\sigma.\text{act}} = \text{retired}_\sigma$.
- ◇ *Ad (G6).* By definition, $\text{freed}_\tau = \text{freed}_{\tau.\text{act}}$ and $\text{adr}(m_\tau|_{\text{valid}_\tau}) = \text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}})$. Then, we conclude by (G6).
- ◇ *Ad (G7).* We conclude by (G7), $\text{freed}_\tau = \text{freed}_{\tau.\text{act}}$, and $\text{retired}_\tau = \text{retired}_{\tau.\text{act}}$.
- ◇ *Ad (G8).* Follows from (P8) together with act not raising unsafe accesses.

◇ **Case 10:** $\text{com} \equiv \text{assume } \text{cond}$ and $\sigma.\text{act} \notin \mathcal{O}[\![P]\!]_{\text{Adr}}^\emptyset$

The update is $up = \emptyset$. We have $m_\tau(u) = m_\sigma(u)$ for all $u \in \text{DVar}$ by (P1). Furthermore by (P4), $m_\tau(p) \neq m_\tau(q)$ implies $m_\sigma(p) \neq m_\sigma(q)$. Hence, for act to be not enabled after σ , the condition must be of the form $\text{cond} \equiv p = q$.

By the semantics, we have $m_\tau(p) = m_\tau(q)$. By assumption, $m_\sigma(p) \neq m_\sigma(q)$. Note that $\{p, q\} \notin \text{valid}_\tau$ must hold because (P1) would otherwise yield $m_\sigma(p) = m_\tau(p) = m_\tau(q) = m_\sigma(q)$ which contradicts the assumption. So we have $p \notin \text{valid}_\tau$ or $q \notin \text{valid}_\tau$. Wlog. assume $p \notin \text{valid}_\tau$, the other case is symmetric. Then, Lemma B.51 together with (P8) yields $m_\tau(p) \in \text{frees}_\tau$. Furthermore, $p \notin \text{valid}_\sigma$ by (P1) and thus $p \notin \text{CVar}$ by Lemma B.54. By Lemmas B.39 and B.40 together with (P1), we have $\text{com} \in \text{next-com}(\sigma)$. Since σ is MPRF as noted above, we must have $q \in \text{CVar}$. By Assumption A.9, $m_\tau(q) \notin \text{frees}_\tau$. So, $m_\tau(p) = m_\tau(q)$ results in $m_\tau(p) \notin \text{frees}_\tau$. Since this contradicts the previous $m_\tau(p) \in \text{frees}_\tau$, Case 10 cannot apply.

◇ **Case 11:** $\text{com} \in \{\text{skip}, \text{beginAtomic}, \text{endAtomic}, @\text{inv} \bullet\}$

We immediately obtain that $\hat{\sigma} = \sigma.\text{act}$ satisfies the claim.

The above case distinction is complete and thus concludes the induction. ■

Proof C.57 (Theorem B.72). We proceed by induction over the structure of τ . In the base case, we have $\tau = \epsilon$ and the claim follows immediately. For the induction step, consider $\tau.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Adr}}^\emptyset$ UAF and assume that we have already constructed σ with the following properties:

- (P1) $\sigma \in \mathcal{O}[\![P]\!]_\emptyset^\emptyset$
- (P2) $\text{ctrl}(\tau) = \text{ctrl}(\sigma)$
- (P3) $(\text{exp} \cap \text{Adr}) \cap (\text{fresh}_\tau \cup \text{freed}_\tau) = \emptyset \implies m_\tau(\text{exp}) = m_\sigma(\text{exp})$
- (P4) $\text{fresh}_\tau = \text{fresh}_\sigma$
- (P5) $\text{freed}_\tau \cup \text{retired}_\tau = \text{retired}_\sigma$
- (P6) $\text{inv}(\tau) \equiv \text{inv}(\sigma)$

We now construct $\hat{\sigma}$ such that:

- (G1) $\hat{\sigma} \in \mathcal{O}[\![P]\!]_\emptyset^\emptyset$
- (G2) $\text{ctrl}(\tau.\text{act}) = \text{ctrl}(\hat{\sigma})$
- (G3) $(\text{exp} \cap \text{Adr}) \cap (\text{fresh}_{\tau.\text{act}} \cup \text{freed}_{\tau.\text{act}}) = \emptyset \implies m_{\tau.\text{act}}(\text{exp}) = m_{\hat{\sigma}}(\text{exp})$
- (G4) $\text{fresh}_{\tau.\text{act}} = \text{fresh}_{\hat{\sigma}}$
- (G5) $\text{freed}_{\tau.\text{act}} \cup \text{retired}_{\tau.\text{act}} = \text{retired}_{\hat{\sigma}}$
- (G6) $\text{inv}(\tau.\text{act}) \equiv \text{inv}(\hat{\sigma})$

Let $\text{act} = \langle t, \text{com}, \text{up} \rangle$.

◇ **Case 1:** $\text{com} \equiv p := q.\text{next}$

Let $a = m_\tau(q)$ and $b = m_\tau(a.\text{next})$. By definition, we have $a \neq \text{seg}$ and $\text{up} = [p \mapsto b]$. We choose $\hat{\sigma} = \sigma.\text{act}$.

◇ *Ad (G1).* Since $\tau.\text{act}$ is UAF, we have $q \in \text{valid}_\tau$. By definition, $q \cap \text{Adr} = \emptyset$. Moreover, the contrapositive of Lemmas B.43 and B.48 gives $a \notin \text{fresh}_\tau \cup \text{freed}_\tau$. Then, (P3) gives us $m_\sigma(p) = a$ and $m_\sigma(a.\text{next}) = b$. Hence, $\sigma.\text{act} \in \mathcal{O}[\![P]\!]_{\text{Adr}}^\emptyset$ as required.

◇ *Ad (G2).* Follows by (P2) together with executing the same action act after both τ and σ .

◇ *Ad (G3).* Consider exp such that $(exp \cap \text{Adr}) \cap (\text{fresh}_{\tau.act} \cup \text{freed}_{\tau.act}) = \emptyset$. If $exp \equiv p$, then $m_{\tau.act}(exp) = b = m_{\sigma.act}(exp)$ due to the form of the update up . Otherwise, we have $m_{\tau.act}(exp) = m_{\tau}(exp)$ and $m_{\sigma.act}(exp) = m_{\sigma}(exp)$. Because of $\text{fresh}_{\tau.act} = \text{fresh}_{\tau}$ and $\text{freed}_{\tau.act} = \text{freed}_{\tau}$, (G3) yields $m_{\tau}(exp) = m_{\sigma}(exp)$. Altogether, we arrive at the desired $m_{\tau.act}(exp) = m_{\sigma.act}(exp)$.

◇ *Ad (G4) to (G6).* The remaining properties follow immediately from (P4) to (P6) together with action act not affecting the fresh/freed/retired addresses nor the invariants.

◇ **Case 2:** $com \in \{ p := q, p.\text{next} := q, u := \text{op}(\bar{u}), u := q.\text{data}, p.\text{data} := u \}$

Analogous to previous case.

◇ **Case 3:** $com \in \{ \text{in:func}(\bar{v}), \text{re:func skip} \}$

We choose $\hat{\sigma} = \sigma.act$. By definition and (P2), $\sigma.act \in \llbracket P \rrbracket_{\emptyset}^{\emptyset}$ satisfying (G1). The remaining (G2) to (G6) follow from (P1) to (P6) together with the fact that act does not affect the memory nor the fresh/freed/retired addresses nor the invariants.

◇ **Case 4:** assume cond

We choose $\hat{\sigma} = \sigma.act$. Let $x \in PVar \cup DVar$. From $x \cap \text{Adr} = \emptyset$ and (P3), we obtain that $m_{\tau}(x) = m_{\sigma}(x)$. Consequently, $cond$ has the same truth value in τ and σ since it contains only variables x . By (P2), we obtain $\sigma.act \in \llbracket P \rrbracket_{\emptyset}^{\emptyset}$ which concludes (G1). The remaining (G2) to (G6) follow from (P1) to (P6) together with act not affecting the memory nor the fresh/freed/retired addresses nor the invariants.

◇ **Case 5:** $p := \text{malloc}$

We choose $\hat{\sigma} = \sigma.act$. Let $a = m_{\tau.act}(p)$. By $\tau.act \in \mathcal{O}\llbracket P \rrbracket_{\text{Adr}}^{\emptyset}$, we have $a \in \text{fresh}_{\tau}$.

◇ *Ad (G1).* We get $a \in \text{fresh}_{\sigma}$ from (P4). This means $\sigma.act \in \llbracket P \rrbracket_{\emptyset}^{\emptyset}$.

◇ *Ad (G3).* Consider exp with $(exp \cap \text{Adr}) \cap (\text{fresh}_{\tau.act} \cup \text{freed}_{\tau.act}) = \emptyset$. If $exp \in \{ p, a.\text{next} \}$, we have $m_{\tau.act}(exp) = m_{\sigma.act}(exp)$ by up updating exp . Otherwise, $m_{\tau.act}(exp) = m_{\tau}(exp)$ and $m_{\sigma.act}(exp) = m_{\sigma}(exp)$. Moreover, $\text{fresh}_{\tau.act} \cup \text{fresh}_{\tau.act} \subseteq \text{fresh}_{\tau} \cup \text{freed}_{\tau}$. Hence, we conclude by (P3).

◇ *Ad (G4).* We conclude by (P4): $\text{fresh}_{\tau.act} = \text{fresh}_{\tau} \setminus \{ a \} = \text{fresh}_{\sigma} \setminus \{ a \} = \text{fresh}_{\sigma.act}$.

◇ *Ad (G5).* By definition, we have:

$$\text{freed}_{\tau.act} \cup \text{retired}_{\tau.act} = (\text{freed}_{\tau} \setminus \{ a \}) \cup \text{retired}_{\tau} = \text{freed}_{\tau} \cup \text{retired}_{\tau}$$

where the last equality holds by $a \notin \text{freed}_{\tau}$, which follows from $a \in \text{fresh}_{\tau}$ and Lemma B.44. By definition, $\text{retired}_{\sigma.act} = \text{retired}_{\sigma}$. Moreover, $\text{freed}_{\tau} \cup \text{retired}_{\tau} = \text{retired}_{\sigma}$ by (G5). Hence, we conclude the desired $\text{freed}_{\tau.act} \cup \text{retired}_{\tau.act} = \text{retired}_{\sigma.act}$.

◇ *Ad (G2) and (G6).* Follows by (P2) and (P6) together with act not affecting the invariants.

◇ **Case 6:** $\text{free}(a)$

We choose $\hat{\sigma} = \sigma$.

◇ *Ad (G1)*. Follows from (P1)

◇ *Ad (G2)*. Follows from (G2) together with $ctrl(\tau) = ctrl(\tau.act)$ by definition.

◇ *Ad (G3)*. Follows from (G3) together with $m_{\tau.act} = m_\tau$ and $m_{\sigma.act} = m_\sigma$.

◇ *Ad (G4)*. By definition, we have $fresh_{\tau.act} = fresh_\tau \setminus \{a\}$. By Lemma B.46, $a \in retired_\tau$. By Lemma B.44, $a \notin fresh_\tau$. Hence, $fresh_{\tau.act} = fresh_\tau = fresh_\sigma$ where the last equality holds by (P4).

◇ *Ad (G5)*. By definition, we have:

$$\begin{aligned} freed_{\tau.act} \cup retired_{\tau.act} &= (freed_\tau \cup \{a\}) \cup (retired_\tau \setminus \{a\}) \\ &= freed_\tau \cup retired_\tau \cup \{a\} = freed_\tau \cup retired_\tau \end{aligned}$$

where the last equality holds by $a \in retired_\tau$ due to Lemma B.46. Finally, (G5) establishes the desired $freed_{\tau.act} \cup retired_{\tau.act} = retired_\sigma$.

◇ *Ad (G6)*. Follows from (P6) together with $inv(\tau.act) = inv(\tau)$.

◇ **Case 7:** $env(a)$

We choose $\hat{\sigma} = \sigma$. By definition, $a \in fresh_\tau \cup freed_\tau$.

◇ *Ad (G1)*. Follows from (P1).

◇ *Ad (G2)*. Follows from (G2) together with $ctrl(\tau) = ctrl(\tau.act)$ by definition.

◇ *Ad (G3)*. Consider some exp with $(exp \cap Addr) \cap (fresh_{\tau.act} \cup freed_{\tau.act}) = \emptyset$. This means we have $exp \notin \{a.next, a.data\}$. So $m_{\tau.act}(exp) = m_\tau(exp)$. Then, $fresh_{\tau.act} = fresh_\tau$ as well as $freed_{\tau.act} = freed_\tau$ together with (G3) yield $m_{\tau.act}(exp) = m_\tau(exp) = m_\sigma(exp)$.

◇ *Ad (G4) to (G6)*. The remaining properties follow immediately from (P4) to (P6) together with action act not affecting the fresh/freed/retired addresses nor the invariants.

◇ **Case 8:** $com \in \{\text{skip}, \text{beginAtomic}, \text{endAtomic}\}$

We immediately obtain that $\hat{\sigma} = \sigma.act$ satisfies the claim.

◇ **Case 9:** $com \equiv @inv \bullet$

We choose $\hat{\sigma} = \sigma.act$. By definition and (P2), we have $\sigma.act \in \llbracket P \rrbracket_\emptyset^\emptyset$, satisfying (G1). Because act does not affect the memory nor the fresh/freed/retired addresses, (G2) to (G5) follow from (P2) to (P5). It remains to establish (G6). To that end, it is sufficient to establish $inv_\tau(act) \equiv inv_\sigma(act)$ according to the definition of invariants together with (P6)..

◇ **Case 9.1:** $com \equiv @inv \text{ angel } r$

By definition, we immediately obtain $inv_\tau(act) \equiv \exists r.true \equiv inv_\sigma(act)$.

◇ **Case 9.2:** $com \equiv @inv\ p = q$

Let $a = m_\tau(p)$ and $b = m_\tau(q)$. By definition, $\{p, q\} \cap \text{Adr} = \emptyset$. Consequently, (G3) yields $m_\sigma(p) = a$ and $m_\sigma(q) = b$. Hence, $inv_\tau(act) \equiv a = b \wedge \text{true} \equiv inv_\sigma(act)$ as required.

◇ **Case 9.3:** $com \equiv @inv\ p\ \text{in}\ r$

As before, $m_\tau(p) = a = m_\sigma(p)$ by (G3). Then, $inv_\tau(act) \equiv a \in r \wedge \text{true} \equiv inv_\sigma(act)$.

◇ **Case 9.4:** $com \equiv @inv\ \text{active}(p)$

As before, $m_\tau(p) = a = m_\sigma(p)$ by (G3). By (G3), we have $active(\tau) = M = active(\sigma)$ for some set of addresses $M \subseteq \text{Adr}$. Then, $inv_\tau(act) \equiv a \in M \wedge \text{true} \equiv inv_\sigma(act)$ as required.

◇ **Case 9.5:** $com \equiv @inv\ \text{active}(r)$

As before, $m_\tau(p) = a = m_\sigma(p)$ by (G3) and $active(\tau) = M = active(\sigma)$ by (G3). Then, we arrive at the required $inv_\tau(act) \equiv r \subseteq M \wedge \text{true} \equiv inv_\sigma(act)$.

The above case distinction is complete and thus concludes the induction. ■

Proof C.58 (Theorem A.10). If $good(\mathcal{O}[P]_{\text{Adr}}^{\text{Adr}})$ then, $good(\llbracket P \rrbracket_\emptyset^\emptyset)$ follows by $\llbracket P \rrbracket_\emptyset^\emptyset \subseteq \mathcal{O}[P]_{\text{Adr}}^{\text{Adr}}$. For the reverse direction, assume $good(\llbracket P \rrbracket_\emptyset^\emptyset)$ holds. To the contrary, assume $good(\mathcal{O}[P]_{\text{Adr}}^{\text{Adr}})$ does not hold. There is $\tau \in \mathcal{O}[P]_{\text{Adr}}^{\text{Adr}}$ so that $good(\tau)$ is not satisfied. Hence, $ctrl(\tau) \cap \text{Fault} \neq \emptyset$ where Fault are the bad control locations. Theorem B.71 yields $\sigma \in \mathcal{O}[P]_{\text{Adr}}^\emptyset$ with $\tau \sim \sigma$. Note that σ is MPRF by assumption. Then, Theorem B.72 yields $\gamma \in \llbracket P \rrbracket_\emptyset^\emptyset$ with $ctrl(\sigma) = ctrl(\gamma)$. Altogether, we arrive at $ctrl(\tau) = ctrl(\gamma)$ and thus $ctrl(\gamma) \cap \text{Fault} \neq \emptyset$. Hence, $good(\llbracket P \rrbracket_\emptyset^\emptyset)$ does not hold. Since this contradicts the assumption, we obtain the desired $good(\mathcal{O}[P]_{\text{Adr}}^{\text{Adr}})$.

Now, we show that $\mathcal{O}[P]_{\text{Adr}}^{\text{Adr}}$ is free from double retires. To the contrary, assume there is a shortest computation $\tau.act \in \mathcal{O}[P]_{\text{Adr}}^{\text{Adr}}$ with $act = \langle t, \text{in:retire}(p), up \rangle$ and $m_\tau(p) \in \text{retire}(\tau)$. Theorem B.71 yields $\sigma \in \mathcal{O}[P]_{\text{Adr}}^\emptyset$ with $\tau \sim \sigma$ and $retired_\tau \subseteq retired_\sigma$. By assumption, σ is MPRF. If $p \notin \text{valid}_\sigma$, then $m_\sigma(p) \in \text{freed}_\sigma$ by Lemma B.52. So, $m_{\sigma.act}(p) \in \text{freed}_{\sigma.act} \cup retired_{\sigma.act}$. Then, Lemma B.70 yields a double retire in $\mathcal{O}[P]_{\text{Adr}}^\emptyset$. Since this contradicts the assumption, we must have $p \in \text{valid}_\sigma$. Hence, $m_\sigma(p) = m_\tau(p) \in retired_\tau \subseteq retired_\sigma$. This means $\sigma.act$ is a double retire in $\mathcal{O}[P]_{\text{Adr}}^\emptyset$. This contradicts the assumption. So, $\mathcal{O}[P]_{\text{Adr}}^{\text{Adr}}$ is free from double retires. ■

Proof C.59 (Theorem 8.5). Set $CVar = \emptyset$ so that $\mathcal{O}[P]_{\text{Adr}}^{\text{Adr}}$ satisfies Assumption A.9. Note that SPRF implies MPRF. Then, the claim follows from Theorem B.71. ■

Proof C.60 (Theorem 8.6). Set $CVar = \emptyset$ so that $\mathcal{O}[P]_{\text{Adr}}^{\text{Adr}}$ satisfies Assumption A.9. Note that SPRF implies MPRF. Then, the claim follows from Theorem B.72. ■

Proof C.61 (Theorem 8.7). Set $CVar = \emptyset$ so that $\mathcal{O}[P]_{\text{Adr}}^{\text{Adr}}$ satisfies Assumption A.9. Note that SPRF implies MPRF. Then, the claim follows from Theorem A.10. ■

C.4 Type System

Proof C.62 (Lemma B.73). By definition. ■

Proof C.63 (Lemma B.74). Consider $\vdash \{\Gamma_1\} \text{stmt} \{\Gamma_2\}$ and let $\text{stmt} \xrightarrow{\text{com}} \text{stmt}'$. We do an induction over the derivation of $\vdash \{\Gamma_1\} \text{stmt} \{\Gamma_2\}$. In the base case, the derivation is due to a single rule application. By definition, the derivation is not due Rule (INFER), (SEQ), (CHOICE), or (LOOP). For the remaining applicable rules we get $\text{stmt} \equiv \text{com}$ and $\text{stmt}' \equiv \text{skip}$. That is, we have $\vdash \{\Gamma_1\} \text{com} \{\Gamma_2\}$. By definition, we have $\vdash \{\Gamma_2\} \text{skip} \{\Gamma_2\}$. We choose $\Gamma = \Gamma_2$. For the induction step, consider a composed derivation $\vdash \{\Gamma_1\} \text{stmt} \{\Gamma_2\}$ and let $\text{stmt} \xrightarrow{\text{com}} \text{stmt}'$. We do a case distinction on the first rule.

◇ **Case 1:** Rule (SEQ), part 1

We have $\text{stmt} \equiv \text{stmt}_1; \text{stmt}_2$. There is Γ with $\vdash \{\Gamma_1\} \text{stmt}_1 \{\Gamma\}$ and $\vdash \{\Gamma\} \text{stmt}_2 \{\Gamma_2\}$ due to the rule definition.

◇ **Case 1.1:** $\text{stmt}_1 \equiv \text{skip}$

By definition, $\text{com} \equiv \text{skip}$ and $\text{stmt}' \equiv \text{stmt}_2$. We immediately obtain $\vdash \{\Gamma_1\} \text{com} \{\Gamma\}$ and $\vdash \{\Gamma\} \text{stmt}' \{\Gamma_2\}$

◇ **Case 1.2:** $\text{stmt}_1 \not\equiv \text{skip}$

By definition, we have $\text{stmt}' \equiv \text{stmt}_1'; \text{stmt}_2$ and $\text{stmt}_1 \xrightarrow{\text{com}} \text{stmt}_1'$. We invoke the induction hypothesis for $\vdash \{\Gamma_1\} \text{stmt}_1 \{\Gamma\}$ and $\text{stmt}_1 \xrightarrow{\text{com}} \text{stmt}_1'$. This yields some Γ' such that $\vdash \{\Gamma_1\} \text{com} \{\Gamma'\}$ and $\vdash \{\Gamma'\} \text{stmt}_1' \{\Gamma\}$. By Rule (SEQ) then, $\vdash \{\Gamma'\} \text{stmt}' \{\Gamma_2\}$.

◇ **Case 2:** Rule (CHOICE)

We have $\text{stmt} \equiv \text{stmt}_1 \oplus \text{stmt}_2$. By definition, we have $\text{stmt} \xrightarrow{\text{com}} \text{stmt}_i \equiv \text{stmt}'$ for $i \in \{1, 2\}$ and $\text{com} \equiv \text{skip}$. The type rules gives $\vdash \{\Gamma_1\} \text{stmt}' \{\Gamma_2\}$. Moreover, $\vdash \{\Gamma_2\} \text{com} \{\Gamma_2\}$.

◇ **Case 3:** Rule (LOOP)

We have $\text{stmt} \equiv \text{stmt}_1^*$ and $\Gamma_1 = \Gamma_2$. By definition, we have $\text{stmt}' \in \{\text{skip}, \text{stmt}_1; \text{stmt}\}$ and $\text{com} \equiv \text{skip}$. We immediately get $\vdash \{\Gamma_1\} \text{com} \{\Gamma_1\}$. Moreover, $\vdash \{\Gamma_1\} \text{stmt}' \{\Gamma_2\}$ follows from Rule (SKIP) in case of $\text{stmt}' \equiv \text{skip}$ and from Rule (SEQ) otherwise.

◇ **Case 4:** Rule (INFER)

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

The above case distinction concludes the induction. ■

Proof C.64 (Lemma B.75). Let $\vdash \{\Gamma_{\text{init}}\} P \{\Gamma\}$. We proceed by induction over the SOS transitions. In the base case, $(pc_{\text{init}}, \epsilon) \rightarrow^0 (pc, \tau)$. That is, $pc = pc_{\text{init}}$ and $\tau = \epsilon$. Let t be a thread. By definition, $\text{stmt}(\tau, t) = \text{skip}$ and

$pc(t) = P^{[t]}$. The former gives $\vdash \{ \Gamma_{init}^{[t]} \} stmt(\tau, t) \{ \Gamma_{init}^{[t]} \}$. The latter gives $\vdash \{ \Gamma_{init}^{[t]} \} pc(t) \{ \Gamma \}$ due to the premise. So we choose $\Gamma_1 = \Gamma_{init}^{[t]}$ and $\Gamma_2 = \Gamma$.

For the induction step, consider $(pc_{init}, \epsilon) \rightarrow^n (pc, \tau) \rightarrow_{t'} (pc', \tau.act)$. Let $t \neq \perp$ be some arbitrary thread. By induction, there are Γ_1', Γ_2' with:

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

First, assume $t \neq t'$. Then, $stmt(\tau.act, t) = stmt(\tau, t)$ and $pc'(t) = pc(t)$. Thus, the claim follows by induction for $\Gamma_1 = \Gamma_1'$. So consider $t = t'$ now. Then, $stmt(\tau.act, t) = stmt(\tau, t); com$. By definition, we have $pc(t) \xrightarrow{com} pc'(t)$ and $act \in \overline{Act}(\tau, t, com)$. Lemma B.74 yields Γ_1 with:

$$\vdash \{ \Gamma_1' \} com \{ \Gamma_1 \} \quad \text{and} \quad \vdash \{ \Gamma_1 \} pc'(t) \{ \Gamma_2' \}.$$

Altogether, we get $\vdash \{ \Gamma_{init}^{[t]} \} stmt(\tau.act, t) \{ \Gamma_1 \}$ and $\vdash \{ \Gamma_1 \} pc'(t) \{ \Gamma_2' \}$ as required. \blacksquare

Proof C.65 (Lemma B.76). Let $\tau.act \in \mathcal{O}[\![P]\!]_{Adr}^\emptyset$. Let t, t' be threads with $t \neq t' = thrd(act)$. By definition of the semantics, there is a step $(pc, \tau) \rightarrow_{t'} (pc', \tau.act)$ for some $pc \in ctrl(\tau)$. If we have $stmt(\tau, t) = skip$, nothing needs to be shown. So assume $stmt(\tau, t) \neq skip$. That is, t has contributed actions to τ . By Assumption 8.11, t must have entered an atomic block to do so. As $t \neq t'$, we have $\neg locked(pc(t))$ according to Rule (SOS-PAR). That is, t must have left the atomic block. By definition of Rule (SOS-ATOMIC3), this results in an action performing `endAtomic`. Applying the same argument inductively on τ , we obtain that the last primitive command of t must have been followed by an `endAtomic`. Hence, $stmt(\tau, t) = stmt; endAtomic$ for some sequence of statements $stmt$, as required. \blacksquare

Proof C.66 (Lemma B.77). Let $\tau.act \in \mathcal{O}[\![P]\!]_{Adr}^\emptyset$. Let t, t' be threads with $t \neq t' = thrd(act)$. Moreover, let $x \in PVar \cup AVar$. Assume $\vdash \{ \Gamma_{init}^{[t]} \} stmt(\tau, t) \{ \Gamma \}$. There are two cases.

◇ **Case 1:** $stmt(\tau, t) = skip$

By the type rules there is Γ' with:

$$\Gamma_{init}^{[t]} \rightsquigarrow \Gamma' \quad \text{and} \quad \vdash \{ \Gamma' \} skip \{ \Gamma' \} \quad \text{and} \quad \Gamma' \rightsquigarrow \Gamma.$$

By definition, we have $\Gamma_{init}^{[t]}(x) = \emptyset$. We obtain $\neg isValid(\Gamma_{init}^{[t]}(x))$. Hence, $\neg isValid(\Gamma'(x))$ and $\neg isValid(\Gamma(x))$ follow by type inference. We conclude $\Gamma(x) \cap \{ \mathbb{A}, \mathbb{L}, \mathbb{S} \} = \emptyset$.

◇ **Case 2:** $stmt(\tau, t) = stmt; endAtomic$

By the typing rules there are $\Gamma_1, \Gamma_2, \Gamma_3$ with:

$$\vdash \{ \Gamma_{init}^{[t]} \} stmt \{ \Gamma_1 \} \quad \text{and} \quad \Gamma_1 \rightsquigarrow \Gamma_2 \quad \text{and} \quad \vdash \{ \Gamma_2 \} endAtomic \{ \Gamma_3 \} \quad \text{and} \quad \Gamma_3 \rightsquigarrow \Gamma$$

where the derivation $\vdash \{ \Gamma_2 \} endAtomic \{ \Gamma_3 \}$ is due to Rule (END). This means, that we have $\Gamma_3 = rm(\Gamma_2)$. By definition, $\mathbb{A} \notin \Gamma_3(x)$. Hence, type inference provides $\mathbb{A} \notin \Gamma(x)$ as desired. Consider $x \notin local_t$ now. There are two cases. First, assume $x \in shared$. Then, we get $\Gamma_3(x) = \emptyset$ by definition of Rule (END). Second, assume

$x \notin \text{shared}$. That is, x is local to another thread $t'' \neq t$, i.e., $x \in \text{local}_{t''}$. 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 (type bindings are only updated by the type rules).

The above case distinction is complete according to Lemma B.76 and thus concludes the claim. ■

Proof C.67 (Lemma B.78). Let $\tau.act \in \mathcal{O}[\![P]\!]_{Adr}^\emptyset$. Let t, t' be threads with $t \neq t' = \text{thrd}(act) \neq \perp$. Consider $p \in PVar \cap \text{local}_t$. By definition, $\text{local}_t \cap \text{local}_{t'} = \emptyset$. Due to the semantics, p does not occur in $\text{com}(act)$. Hence, $p \in \text{valid}_\tau \iff p \in \text{valid}_{\tau.act}$. Further, $m_\tau(p) = m_{\tau.act}(p)$. So every valid alias created by act requires a valid alias in τ . This is not possible since $\text{noalias}_\tau(p)$. ■

Proof C.68 (Lemma B.79). Let $\tau.act \in \mathcal{O}[\![P]\!]_{Adr}^\emptyset$ UAF with $act = \langle t, @\text{inv } p = q, \emptyset \rangle$ and $\text{inv}(\tau.act)$. The latter gives $m_\tau(p) = m_\tau(q)$. Then, Lemma B.53 gives $p, q \in \text{valid}_\tau$ as required. ■

Proof C.69 (Lemma B.80). Let $\tau.act \in \mathcal{O}[\![P]\!]_{Adr}^\emptyset$ UAF such that $act = \langle t, @\text{inv active}(p), up \rangle$ and $\text{inv}(\tau.act)$. By definition, this means $m_\tau(p) \in \text{active}(\tau)$. So, $m_\tau(p) \notin \text{freed}_\tau$. Lemma B.52 yields $p \in \text{valid}_\tau$. We get $p \in \text{valid}_{\tau.act}$ by definition. Moreover, $m_{\tau.act}(p) \in \text{active}(\tau.act)$. That is, $m_{\tau.act}(p) \notin \text{retired}_{\tau.act}$. Hence, the remaining property follows from Lemma B.45. ■

Proof C.70 (Lemma B.81). Let $\tau.act \in \mathcal{O}[\![P]\!]_{Adr}^\emptyset$ such that $act = \langle t, @\text{inv active}(r), up \rangle$ and $\text{inv}(\tau.act)$. By definition, we have $\text{repr}_\tau(r) \subseteq \text{active}(\tau)$. Hence, $\text{repr}_\tau(r) \cap \text{freed}_{\tau.act} = \emptyset$ holds. Moreover, we also have $\text{repr}_{\tau.act}(r) \subseteq \text{active}(\tau.act)$ by definition. Let $a \in \text{repr}_{\tau.act}(r)$. This means $a \in \text{active}(\tau.act)$. That is, we have $a \notin \text{retired}_{\tau.act}$. Then, the remaining property follows from Lemma B.45. ■

Proof C.71 (Lemma B.82). Let $\tau \in \mathcal{O}[\![P]\!]_{Adr}^\emptyset$. Let Γ, Γ' be two type environments with $\Gamma \rightsquigarrow \Gamma'$. Let $p \in PVar$ be a pointer with $a = m_\tau(p)$ and let $r \in AVar$ be a ghost variable and $b \in \text{repr}_\tau(r)$.

- Assume $\text{isValid}(\Gamma(p)) \implies p \in \text{valid}_\tau$. By $\Gamma \rightsquigarrow \Gamma'$, we have $\text{isValid}(\Gamma'(p)) \implies \text{isValid}(\Gamma(p))$. Hence, we obtain the desired $\text{isValid}(\Gamma'(p)) \implies p \in \text{valid}_\tau$.
- Assume $\text{isValid}(\Gamma(r)) \implies b \notin \text{freed}_\tau$. By $\Gamma \rightsquigarrow \Gamma'$, we have $\text{isValid}(\Gamma'(r)) \implies \text{isValid}(\Gamma(r))$. Hence, we obtain the desired $\text{isValid}(\Gamma'(r)) \implies b \notin \text{freed}_\tau$.
- Assume $\mathbb{L} \in \Gamma(p) \implies \text{noalias}_\tau(p)$. By $\Gamma \rightsquigarrow \Gamma'$, we have $\mathbb{L} \in \Gamma'(p) \implies \mathbb{L} \in \Gamma(p)$. Hence, we obtain the desired $\mathbb{L} \in \Gamma'(p) \implies \text{noalias}_\tau(p)$.
- Assume $\text{reach}_{t,a}^\emptyset(\mathcal{H}(\tau)) \subseteq \text{Loc}(\Gamma(p))$. By $\Gamma \rightsquigarrow \Gamma'$, we have $\text{Loc}(\Gamma(p)) \subseteq \text{Loc}(\Gamma'(p))$. Hence, we obtain the desired $\text{reach}_{t,a}^\emptyset(\mathcal{H}(\tau)) \subseteq \text{Loc}(\Gamma'(p))$.
- Assume $\text{reach}_{t,b}^\emptyset(\mathcal{H}(\tau)) \subseteq \text{Loc}(\Gamma(r))$. By $\Gamma \rightsquigarrow \Gamma'$, we have $\text{Loc}(\Gamma(r)) \subseteq \text{Loc}(\Gamma'(r))$. Hence, we obtain the desired $\text{reach}_{t,b}^\emptyset(\mathcal{H}(\tau)) \subseteq \text{Loc}(\Gamma'(r))$.

This concludes the claim. ■

Proof C.72 (Theorem B.83). Let $\tau \in \mathcal{O}[[P]]_{Adr}^\emptyset$. We proceed by induction over the structure of τ in order to show the following:

$$\begin{aligned} & freed_\tau \cap retired_\tau = \emptyset \\ \text{and} \quad & \forall t \forall \Gamma. \vdash \{ \Gamma_{init}^{[t]} \} stmt(\tau, t) \{ \Gamma \} \\ & \implies \forall p, r. \left(\begin{array}{l} reach_{t, m_\tau(p)}^\mathcal{O}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma(p)) \\ \wedge isValid(\Gamma(p)) \implies p \in valid_\tau \\ \wedge \mathbb{L} \in \Gamma(p) \implies noalias_\tau(p) \\ \wedge \forall a \in repr_\tau(r). reach_{t, a}^\mathcal{O}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma(r)) \\ \wedge isValid(\Gamma(r)) \implies repr_\tau(r) \cap freed_\tau = \emptyset \end{array} \right) \end{aligned}$$

Base Case. We have $\tau = \epsilon$. Let t, Γ with $\vdash \{ \Gamma_{init}^{[t]} \} stmt(\tau, t) \{ \Gamma \}$. By definition, $stmt(\tau, t) = \text{skip}$ as well as $freed_\tau \cap retired_\tau = \emptyset$. Let $p \in PVar$ and $r \in AVar$. 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:

$$\Gamma_{init}^{[t]} \rightsquigarrow \Gamma_1 \quad \text{and} \quad \vdash \{ \Gamma_1 \} \text{skip} \{ \Gamma_1 \} \quad \text{and} \quad \Gamma_1 \rightsquigarrow \Gamma.$$

Since $\mathbb{L} \notin \Gamma_{init}^{[t]}(p)$, 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, $freed_\tau = \emptyset$. So, $isValid(\Gamma(r)) \implies a \notin freed_\tau$ holds for all $a \in Adr$. From Lemma B.73, we get $\Gamma_{init}^{[t]} \rightsquigarrow \Gamma$. That is $Loc(\Gamma_{init}^{[t]}(p)) \subseteq Loc(\Gamma(p))$. By definition, $\Gamma_{init}^{[t]}(p) = \emptyset$. That is, $reach_{t, m_\tau(p)}^\mathcal{O}(\mathcal{H}(\tau)) \subseteq Loc(\mathcal{O}) = Loc(\emptyset) = Loc(\Gamma_{init}^{[t]}(p))$. Similarly for r . Altogether, this concludes the base case.

Induction Step. Consider $\tau.act$ UAF¹ with $act = \langle t', com, up \rangle$ and $inv(\tau.act)$. Let t be some arbitrary thread. We establish the claim for t . To that end, we do a case distinction over thread t' executing act .

◇ **Case 1:** $t = t'$ and $t' \neq \perp$

By definition, we have $stmt(\tau.act, t) = stmt(\tau, t); com$ and $freed_{\tau.act} \subseteq freed_\tau$. First, we establish that $freed_{\tau.act} \cap retired_{\tau.act} = \emptyset$ holds. If $retired_{\tau.act} \subseteq retired_\tau$, then the claim follows by induction. Otherwise, we have $retired_{\tau.act} = retired_\tau \cup \{a\}$ and $com \equiv \text{in:retire}(q)$ with $m_\tau(q) = a$. Since $\vdash \{ \Gamma_1 \} com \{ \Gamma_2 \}$ holds, we know $q \in valid_\tau$. By the contrapositive of Lemma B.52, we get $a \notin freed_\tau$. So by induction, $freed_{\tau.act} \cap retired_{\tau.act} = \emptyset$.

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

$$\vdash \{ \Gamma_{init}^{[t]} \} stmt(\tau.act, t) \{ \Gamma_3 \}.$$

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

$$\vdash \{ \Gamma_{init}^{[t]} \} stmt(\tau, t) \{ \Gamma_0 \} \quad \text{and} \quad \Gamma_0 \rightsquigarrow \Gamma_1 \quad \text{and} \quad \vdash \{ \Gamma_1 \} com \{ \Gamma_2 \} \quad \text{and} \quad \Gamma_2 \rightsquigarrow \Gamma_3$$

¹ We use the UAF assumption implicitly when invoking previous results.

where $\vdash \{ \Gamma_1 \} \text{ com } \{ \Gamma_2 \}$ is derived by neither (SEQ), (CHOICE), (LOOP), nor (INFER). Induction yields the claim for Γ_0 . So by Lemma B.82 the claim also holds for Γ_1 . If the claim holds for Γ_2 , then the claim follow for Γ_3 from Lemma B.82 again. It remains to show that the claim holds for Γ_2 relying on Γ_1 . Let $p \in PVar$ and $r \in AVar$ be arbitrary. Let $m_{\tau.act}(p) = c_p$ and let $c_r \in repr_{\tau.act}(r)$. We show

- (G1) $reach_{t,c_p}^O(\mathcal{H}(\tau.act)) \subseteq Loc(\Gamma_2(p))$
- (G2) $reach_{t,c_r}^O(\mathcal{H}(\tau.act)) \subseteq Loc(\Gamma_2(r))$
- (G3) $isValid(\Gamma_2(p)) \implies p \in valid_{\tau.act}$
- (G4) $\mathbb{L} \in \Gamma_2(p) \implies noalias_{\tau.act}(p)$
- (G5) $isValid(\Gamma_2(r)) \implies c_r \notin freed_{\tau.act}$

We do a case distinction over the type rule applied to derive $\vdash \{ \Gamma_1 \} \text{ com } \{ \Gamma_2 \}$.

◇ **Case 1.1:** Rule (END)

The type rules gives $\Gamma_2 = rm(\Gamma_1)$. By definition, $\Gamma_2(p) \subseteq \Gamma_1(p)$ and $\Gamma_2(r) \subseteq \Gamma_1(r)$. This means $Loc(\Gamma_1(p)) \subseteq Loc(\Gamma_2(p))$ and $Loc(\Gamma_1(r)) \subseteq Loc(\Gamma_2(r))$. We obtain (G1) and (G2) by $\mathcal{H}(\tau) = \mathcal{H}(\tau.act)$. Now, assume $isValid(\Gamma_2(p))$. By $\Gamma_2(p) \subseteq \Gamma_1(p)$, we have $isValid(\Gamma_1(p))$. Induction yields $p \in valid_{\tau}$. Since $valid_{\tau} = valid_{\tau.act}$, we get (G3). Next, assume $isValid(\Gamma_2(r))$. By $\Gamma_2(r) \subseteq \Gamma_1(r)$, we get $isValid(\Gamma_1(r))$. Induction together with $repr_{\tau} = repr_{\tau.act}$ gives $c_r \notin freed_{\tau}$. Due to $freed_{\tau} = freed_{\tau.act}$, we arrive at (G5). If $\mathbb{L} \in \Gamma_2(p)$, then $\mathbb{L} \in \Gamma_1(p)$ and thus $noalias_{\tau}(p)$ by induction. Since $m_{\tau} = m_{\tau.act}$ by definition, we arrive at (G4).

◇ **Case 1.2:** Rule (BEGIN) or Rule (SKIP)

The type rule gives $\Gamma_2 = \Gamma_1$. By definition, we have $m_{\tau} = m_{\tau.act}$ and $valid_{\tau} = valid_{\tau.act}$. Moreover, we have $freed_{\tau} = freed_{\tau.act}$ and $repr_{\tau} = repr_{\tau.act}$ as well as $\mathcal{H}(\tau) = \mathcal{H}(\tau.act)$. Hence, (G1) to (G5) follow by induction.

◇ **Case 1.3:** Rule (ASSIGN1)

We have $freed_{\tau} = freed_{\tau.act}$ and $repr_{\tau} \equiv repr_{\tau.act}$. Hence, (G5) holds by induction because of $\Gamma_2(r) = \Gamma_1(r)$. If $\mathbb{L} \in \Gamma_2(p)$, then p cannot appear in com since Rule (ASSIGN1) would have removed \mathbb{L} . Hence, no alias of p is created by com , (G4) continues to hold by induction. Assume now $isValid(\Gamma_2(p))$. There are two cases. First, $com \equiv p := q$. Then, we have $\Gamma_2(p) = \Gamma_1(q) \setminus \{ \mathbb{L} \}$. So, induction gives $q \in valid_{\tau}$. We conclude $p \in valid_{\tau.act}$ as required. Second, p is not assigned to by com . Then, $\Gamma_2(p) \subseteq \Gamma_1(p)$. Hence, $p \in valid_{\tau}$ by induction and thus $p \in valid_{\tau.act}$. This concludes (G3). Note that $\mathcal{H}(\tau.act) = \mathcal{H}(\tau)$. Then, (G2) continues to hold by induction since r is not affected by com . It remains to establish (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, we get $\Gamma_2(p) = \Gamma_1(p) \setminus \{ \mathbb{L} \}$. That is, $Loc(\Gamma_1(p)) \subseteq Loc(\Gamma_2(p))$. So, we get (G1) by induction. Otherwise, com takes the form $com \equiv p := q$ with some q . Then, the type rules give $\Gamma_2(p) = \Gamma_1(q) \setminus \{ \mathbb{L} \}$. Furthermore, $m_{\tau.act}(p) = m_{\tau}(q) = c_p$. By induction, we have $reach_{t,c_p}^O(\mathcal{H}(\tau)) \subseteq Loc(\Gamma_1(q))$. Hence, $reach_{t,c_p}^O(\mathcal{H}(\tau)) \subseteq Loc(\Gamma_1(q) \setminus \{ \mathbb{L} \})$. We get the desired $reach_{t,c_p}^O(\mathcal{H}(\tau.act)) \subseteq Loc(\Gamma_2(q))$ by definition. This concludes (G1).

◇ **Case 1.4:** Rule (ASSIGN2) or Rule (ASSIGN3)

Analogous to the previous case for (ASSIGN1).

◇ **Case 1.5:** Rule (ASSIGN4), Rule (ASSIGN5), Rule (ASSIGN6), or Rule (ASSUME2)

We have $\Gamma_2 = \Gamma_1$ as well as $repr_\tau = repr_{\tau.act}$, $valid_\tau = valid_{\tau.act}$, and $freed_\tau = freed_{\tau.act}$ as well as $\mathcal{H}(\tau) = \mathcal{H}(\tau.act)$. Moreover, $m_\tau(p) = m_{\tau.act}(p)$. So (G1) to (G5) follow by induction.

◇ **Case 1.6:** Rule (ASSUME1-CONSTANT)

Wlog. $com \equiv \text{assume } C = q$ with $C \in CVar$. By the semantics, we have $m_\tau(C) = m_\tau(q)$. Lemma B.54 yields $q \in valid_\tau$. Lemma B.51 and Assumption A.9 gives $C \in valid_\tau$. Hence, we get $valid_\tau = valid_{\tau.act}$. Moreover, we have $freed_\tau = freed_{\tau.act}$ and $repr_\tau \equiv repr_{\tau.act}$. We show the required properties. By definition, we have $\Gamma_2(r) = \Gamma_1(r)$. Hence, (G5) follows by induction. If $\mathbb{L} \in \Gamma_2(p)$, then $\mathbb{L} \in \Gamma_1(p)$ due to the type rule. This means $noalias_\tau(p)$ by induction. Note that $m_\tau = m_{\tau.act}$. So we get $noalias_{\tau.act}(p)$ by definition together with the above $valid_\tau = valid_{\tau.act}$. This establishes (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 , i.e., $p \in \{C, q\}$. So, $p \in valid_\tau$ as before. This gives (G3). Since $\mathcal{H}(\tau.act) = \mathcal{H}(\tau)$ and $repr_\tau = repr_{\tau.act}$, (G2) continues to hold by induction since r is not affected. It remains to establish (G1). If p does not occur in com nothing needs to be show. So assume p appears in com . By the the type rule, we have $\Gamma_2(p) = (\Gamma_1(C) \wedge \Gamma_1(q)) \setminus \{\mathbb{L}\}$. By the semantics, we get:

$$m_{\tau.act}(C) = m_\tau(C) = m_\tau(q) = m_{\tau.act}(q) \quad \text{and} \quad m_\tau(p) = c_p = m_{\tau.act}(p).$$

We conclude (G1) by induction as follows:

$$reach_{t,c_p}^\mathcal{O}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma_1(p)) \cap Loc(\Gamma_1(q)) = Loc(\Gamma_1(p) \wedge \Gamma_2(p)) = Loc(\Gamma_2(p)).$$

◇ **Case 1.7:** Rule (ASSUME1)

Analogous to the previous case.

◇ **Case 1.8:** Rule (EQUAL)

If $isValid(\Gamma_2(p))$, then there are two cases. First, $isValid(\Gamma_1(p))$ holds. By induction, this means $p \in valid_\tau$. Then, we get $p \in valid_{\tau.act}$ 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 $com \equiv @inv \ p = q$ with $isValid(q)$. By induction, we have $q \in valid_\tau$. Then, B.79 gives $p \in valid_\tau$. As before, we get $p \in valid_{\tau.act}$. This concludes (G3). The remaining properties follow analogously to the previous case for Rule (ASSUME1). For (G1) note that $inv(\tau.act)$ gives $m_\tau(p) = m_\tau(q)$ and thus $inv(\tau) \iff inv(\tau.act)$. That is, $repr_\tau = repr_{\tau.act}$.

◇ **Case 1.9:** Rule (ACTIVE) for pointers

If $isValid(\Gamma_2(p))$, then there are two cases. First, $isValid(\Gamma_1(p))$ holds. By induction, this means $p \in valid_\tau$. Then, we get $p \in valid_{\tau.act}$ because $valid_\tau = valid_{\tau.act}$ by definition. Second, $\neg isValid(\Gamma_1(p))$ holds. That is, p is validated by com . Hence, com must be of the form $com \equiv @inv \ active(p)$. Since the invariants hold by assumption, $inv(\tau.act)$, we can invoke Lemma B.80. It gives $p \in valid_{\tau.act}$. Altogether, this concludes (G3). If we have $\mathbb{L} \in \Gamma_2(p)$, then $\mathbb{L} \in \Gamma_1(p)$ due to the type rule. Hence, the induction hypothesis together with $m_\tau = m_{\tau.act}$ and $valid_\tau = valid_{\tau.act}$ gives (G4). For (G5) observe that we have $\Gamma_2(r) = \Gamma_1(r)$ and

$repr_\tau(r) = repr_{\tau.act}(r)$ because $inv(\tau.act)$. So (G5) follows by induction. By definition, $\mathcal{H}(\tau.act) = \mathcal{H}(\tau)$. Since r is not affected, (G2) follows by induction. We show (G1). If com does not contain p , nothing needs to be shown. Otherwise, com is of the form $com \equiv @inv \text{ active}(p)$. From Lemma B.80 we get $reach_{t,c_p}^\mathcal{O}(\mathcal{H}(\tau.act)) \subseteq Loc(\mathbb{A})$. Induction yields $reach_{t,c_p}^\mathcal{O}(\mathcal{H}(\tau.act)) \subseteq Loc(\Gamma_1(p))$. Hence, we conclude (G1) by definition as follows:

$$reach_{t,c_p}^\mathcal{O}(\mathcal{H}(\tau.act)) \subseteq Loc(\Gamma_1(p)) \cap Loc(\mathbb{A}) \subseteq Loc(\Gamma_1(p) \wedge \mathbb{A}) = Loc(\Gamma_2(p)) .$$

◇ **Case 1.10:** Rule (ACTIVE) for angels

Using Lemma B.81, this case is analogous to the previous case for pointer variables.

◇ **Case 1.11:** Rule (MALLOC)

Recall that $\tau.act \in \mathcal{O}[P]_{Adr}^\mathcal{O}$. So act allocates a fresh address. Hence, $freed_\tau = freed_{\tau.act}$. Moreover, $repr_\tau \equiv repr_{\tau.act}$. Then, (G5) holds by induction. (G2) continues to hold by induction since r is not affected. Consider (G1). If p does not appear in com , nothing needs to be shown. Otherwise, $com \equiv p := \text{malloc}$. By Lemma B.44, $c_p \notin retired_\tau$ and thus $c_p \notin retired_{\tau.act}$. Then, $reach_{t,c_p}^\mathcal{O}(\mathcal{H}(\tau.act)) \subseteq \{L_2\} \times Loc(\mathcal{O}_{SMR}) = Loc(\mathbb{L})$ follows by Lemma B.45. By definition, $\Gamma_2(p) = \{\mathbb{L}\}$. This concludes (G1).

For the remaining properties, let com be $com \equiv q := \text{malloc}$. First, consider $p \neq q$. Then, $\Gamma_2(p) = \Gamma_1(p)$ and $p \in valid_{\tau.act} \iff p \in valid_\tau$. Then, (G3) follows by induction. Now, assume $\mathbb{L} \in \Gamma_2(p)$. By induction, we have $noalias_\tau(p)$. This means $m_\tau(p) \neq \text{seg}$ and $m_\tau(p) \notin m_\tau(valid_\tau \setminus \{p\})$. Lemma B.47 yields $m_{\tau.act}(valid_{\tau.act}) \subseteq m_\tau(valid_\tau)$. Hence, we arrive at $m_{\tau.act}(p) \notin m_{\tau.act}(valid_{\tau.act} \setminus \{p\})$. Moreover, $m_\tau(p) = m_{\tau.act}(p)$. Altogether, this means $noalias_{\tau.act}(p)$. This establishes (G4).

Second, consider the case $p = q$. Then, $\Gamma_2(p) = \{\mathbb{L}\}$. By Lemma B.42, $a \notin m_\tau(valid_\tau)$. Hence, we get $a \notin m_{\tau.act}(valid_{\tau.act} \setminus \{p\})$. This means $noalias_{\tau.act}(p)$ holds by definition. This establishes (G4). Moreover, $p \in valid_{\tau.act}$ by definition. So (G3) holds as well.

◇ **Case 1.12:** Rule (ENTER)

By definition, $m_\tau = m_{\tau.act}$ and $repr_\tau = repr_{\tau.act}$. We have $reach_{t,c_p}^\mathcal{O}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma_1(p))$ by induction. Type inference $\Gamma_1, com \rightsquigarrow \Gamma_2$ results in $post_{p,com}(Loc(\Gamma_1(p))) \subseteq Loc(\Gamma_2(p))$. Hence, we get the desired $reach_{t,c_p}^\mathcal{O}(\mathcal{H}(\tau.act)) \subseteq Loc(\Gamma_2(p))$. This concludes (G1). We conclude (G2) along the same lines. Now, assume $isValid(\Gamma_2(p))$. Then, $isValid(\Gamma_1(p))$ by the definition of type inference. So, (G3) follows by induction and $valid_\tau = valid_{\tau.act}$. Similarly, $isValid(\Gamma_2(r))$ implies $isValid(\Gamma_1(r))$. Then, (G5) follows by induction together with $freed_\tau = freed_{\tau.act}$ and $repr_\tau = repr_{\tau.act}$. Lastly, assume $\mathbb{L} \in \Gamma_2(p)$. Then $\mathbb{L} \in \Gamma_1(p)$ by definition. We obtain the required $noalias_{\tau.act}(p)$ by induction and $m_\tau = m_{\tau.act}$. This concludes (G4).

◇ **Case 1.13:** Rule (EXIT)

Analogously to the previous case for Rule (ENTER).

◇ **Case 1.14:** Rule (ANGEL)

By definition, $m_\tau = m_{\tau.act}$, $valid_\tau = valid_{\tau.act}$, and $freed_\tau = freed_{\tau.act}$. Moreover, the type rule gives $\Gamma_2(p) = \Gamma_1(p)$. So (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, (G5) follows by induction. Since p is not affected, we get (G1) by induction. It remains to show (G2). If r does not appear in com , nothing needs to be show because $repr_{\tau.act}(r) = repr_{\tau}(r)$ holds due to the fact that annotations cannot correlate different angels. Otherwise, $\Gamma_2(r) = \emptyset$. This concludes (G2).

◇ **Case 1.15:** Rule (MEMBER)

By definition, $m_{\tau} = m_{\tau.act}$ and $valid_{\tau} = valid_{\tau.act}$. If $isValid(\Gamma_2(r))$, then $isValid(\Gamma_1(r))$ holds since $\Gamma_2(r) = \Gamma_1(r)$. Moreover, we have $inv(\tau.act) \implies inv(\tau)$ by construction of $inv(\bullet)$. This means $repr_{\tau.act}(r) \subseteq repr_{\tau}(r)$. So, we get $c_r \notin freed_{\tau}$ by induction. This concludes (G5) because of $freed_{\tau} = freed_{\tau.act}$. 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, (G4) follows by induction. Now, assume we have $isValid(\Gamma_2(p))$. There are two cases. First, consider $isValid(\Gamma_1(p))$. Then, $p \in valid_{\tau}$ by induction and thus $p \in valid_{\tau.act}$. Second, consider $\neg isValid(\Gamma_1(p))$. Then, com validates p . We must have $com \equiv @inv\ p\ in\ r'$ with $isValid(\Gamma_1(r'))$. Then, we get $m_{\tau}(p) \in repr_{\tau.act}(r')$ from $inv(\tau.act)$. Further, $isValid(\Gamma_1(r'))$ gives $isValid(\Gamma_2(r'))$. Hence, $m_{\tau}(p) \notin freed_{\tau.act}$ follows from the already established (G5). The contrapositive of Lemma B.52 yields $p \in valid_{\tau.act}$. This establishes (G3). By definition, we have $\mathcal{H}(\tau.act) = \mathcal{H}(\tau)$ and $\Gamma_2(r) = \Gamma_1(r)$. Hence, (G2) follows by induction. It remains to show (G1). If p does not occur in com , nothing needs to be shown. Otherwise, we have $com \equiv @inv\ p\ in\ r'$. As above, we get $c_p \in repr_{\tau}(r')$. Induction gives:

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

This concludes (G1) because $\mathcal{H}(\tau.act) = \mathcal{H}(\tau)$.

◇ **Case 2:** $t \neq t'$ and $t' \neq \perp$

We conclude $freed_{\tau.act} \cap retired_{\tau.act} = \emptyset$ as in the previous case. Consider now some t, Γ_4 such that $\vdash \{ \Gamma_{init}^{[t]} \} stmt(\tau, t) \{ \Gamma_4 \}$. By definition, we have:

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

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

Consider $p \in PVar$. If $\mathbb{L} \in \Gamma_4(p)$, then $p \in local_t$ by the contrapositive of Lemma B.77. By induction, we have $noalias_{\tau}(p)$. Then, Lemma B.78 gives $noalias_{\tau.act}(p)$. If $isValid(\Gamma_4(p))$, then $p \in local_t$ by the contrapositive of Lemma B.77 and $p \in valid_{\tau}$ by induction. Lemma B.78 gives $p \in valid_{\tau.act}$.

Consider $r \in AVar$. If $isValid(\Gamma_4(r))$, then $r \in local_t$ by the contrapositive of Lemma B.77. By induction, we get $repr_{\tau}(r) \cap freed_{\tau} = \emptyset$. By $t \neq t'$, we know that r cannot occur in com . Consequently, we have $repr_{\tau.act}(r) = repr_{\tau}(r)$. Observe that we have $freed_{\tau.act} \subseteq freed_{\tau}$ due to $t' \neq \perp$. Hence, we obtain $repr_{\tau.act}(r) \cap freed_{\tau.act} = \emptyset$ as required.

Let $x \in dom(\Gamma_4)$. If $x \in shared$, Lemma B.76 yields $\Gamma_4(x) = \emptyset$. So, $Loc(\Gamma_4(x)) = Loc(\mathcal{O})$ entails the remaining properties. Assume $x \notin shared$ hereafter. This means $x \in local_t$. Consider $a \in m_{\tau.act}(x)$; to be precise, we

mean $a = m_{\tau.act}(x)$ if $x \in PVar$ and $a \in repr_{\tau.act}(x)$ if $x \in AVar$. Because x is local to t , it cannot appear in com due to the semantics. Hence, we have $a \in m_\tau(x)$. By induction, we have $reach_{t,a}^\mathcal{O}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma_4(x))$. We establish the required $reach_{t,a}^\mathcal{O}(\mathcal{H}(\tau.act)) \subseteq Loc(\Gamma_4(x))$. If $\mathcal{H}(\tau.act) = \mathcal{H}(\tau)$, then the claim follows by induction. Otherwise, $\mathcal{H}(\tau.act)$ is of the form $\mathcal{H}(\tau.act) = h.evt$ with $\mathcal{H}(\tau) = h$. Since $t \neq t'$, we know that evt is not an event of t , that is, $evt \downarrow_t = \epsilon$. By the definition of closedness under interference, we have for \mathbb{S} and all \mathbb{E}_L :

$$\begin{aligned} reach_{t,a}^\mathcal{O}(h) \subseteq Loc(\mathbb{S}) &\implies reach_{t,a}^\mathcal{O}(h.evt) \subseteq Loc(\mathbb{S}) \\ \text{and } reach_{t,a}^\mathcal{O}(h) \subseteq Loc(\mathbb{E}_L) &\implies reach_{t,a}^\mathcal{O}(h.evt) \subseteq Loc(\mathbb{E}_L). \end{aligned}$$

By induction, this means that $\mathbb{G} \in \Gamma_4(x)$ implies $reach_{t,a}^\mathcal{O}(h.evt) \subseteq Loc(\mathbb{G})$ for all guarantees $\mathbb{G} \in \{\mathbb{S}, \mathbb{E}_{L_1}, \dots, \mathbb{E}_{L_k}\}$. By Lemma B.77, we have $\mathbb{A} \notin \Gamma_4(x)$. It remains to show: $\mathbb{L} \in \Gamma_4(x)$ implies $reach_{t,a}^\mathcal{O}(h.evt) \subseteq Loc(\mathbb{L})$. So assume $\mathbb{L} \in \Gamma_4(x)$. Then we have $x \in PVar$ since the type rules do not allow angels to carry \mathbb{L} . By induction, we have $reach_{t,a}^\mathcal{O}(h) \subseteq Loc(\mathbb{L})$. Towards a contradiction, assume $reach_{t,a}^\mathcal{O}(h.evt) \not\subseteq Loc(\mathbb{L})$. By definition, this means that evt makes \mathcal{O}_{Base} leave its initial location. To do that, evt must be of the form $evt = in:retire(t', a)$. That is, $com \equiv in:retire(q)$ with $m_\tau(q) = a$. From the already established $freed_{\tau.act} \cap retired_{\tau.act} = \emptyset$, we conclude that $a \notin freed_{\tau.act}$. Hence, we get $a \notin freed_\tau$. The contrapositive of Lemma B.52 gives $q \in valid_\tau$. We get $\neg noalias_\tau(x)$ because of $x \neq q$. However, induction together with $\mathbb{L} \in \Gamma_4(x)$ gives $noalias_\tau(x)$. Since this resembles a contradiction, we must have $reach_{t,a}^\mathcal{O}(h.evt) \subseteq Loc(\mathbb{L})$ as required.

Combining the above results for individual guarantees yields: $reach_{t,a}^\mathcal{O}(\mathcal{H}(\tau)) \subseteq Loc(\Gamma_4(x))$ implies $reach_{t,a}^\mathcal{O}(\mathcal{H}(\tau.act)) \subseteq Loc(\Gamma_4(x))$. Hence, we get $reach_{t,a}^\mathcal{O}(\mathcal{H}(\tau.act)) \subseteq Loc(\Gamma_4(x))$ by induction, as required.

◇ **Case 3:** $t' = \perp$

We have $com \equiv free(a)$ or $com \equiv env(a)$. In the latter case, we get $m_\tau(p) = m_{\tau.act}(p)$ for all pointers $p \in PVar$, $valid_\tau = valid_{\tau.act}$, $\mathcal{H}(\tau) = \mathcal{H}(\tau.act)$, $freed_\tau = freed_{\tau.act}$, and $repr_\tau = repr_{\tau.act}$. Moreover, $m_\tau(valid_\tau) \subseteq m_{\tau.act}(valid_{\tau.act})$ by Lemma B.47 results in $noalias_\tau(p) \implies noalias_{\tau.act}(p)$ for all pointers $p \in PVar$. Hence, the claim follows by induction.

Consider now $com \equiv free(a)$. The update is $up = \emptyset$. We have $freed_{\tau.act} = freed_\tau \cup \{a\}$ as well as $retired_{\tau.act} = retired_\tau \setminus \{a\}$. Hence, we obtain $freed_{\tau.act} \cap retired_{\tau.act} = \emptyset$ as required. Let t, Γ with $\vdash \{\Gamma_{init}^{[t]}\} stmt(\tau, t) \{\Gamma\}$. By definition, we have $stmt(\tau.act, t) = stmt(\tau, t)$. That is, $\vdash \{\Gamma_{init}^{[t]}\} stmt(\tau.act, t) \{\Gamma\}$. We show that Γ satisfies the claim. Let $\mathcal{H}(\tau) = h$. This means $\mathcal{H}(\tau.act) = h.free(a)$. By the semantics, we have $h.free(a) \in \mathcal{S}(\mathcal{O})$.

Consider $p \in PVar$. If $\mathbb{L} \in \Gamma(p)$, then $noalias_\tau(p)$. Since $m_\tau = m_{\tau.act}$ and $valid_{\tau.act} \subseteq valid_\tau$, we obtain $noalias_{\tau.act}(p)$. If $isValid(\Gamma(p))$, then $\{\mathbb{A}, \mathbb{L}, \mathbb{S}\} \cap \Gamma(p) \neq \emptyset$. By induction, we have $p \in valid_\tau$ and $reach_{t,m_\tau(p)}^\mathcal{O}(h) \subseteq Loc(\Gamma(p))$. Consider $\mathbb{A} \in \Gamma(p)$ or $\mathbb{L} \in \Gamma(p)$. By definition of the meaning of types, $reach_{t,m_\tau(p)}^\mathcal{O}(h) \subseteq \{L_2\} \times Loc(\mathcal{O}_{SMR})$. Hence, $m_\tau(p)$ cannot be freed according to \mathcal{O}_{Base} as otherwise it would reach its accepting location and thus contradict $h.free(a) \in \mathcal{S}(\mathcal{O})$. We get $p \in valid_{\tau.act}$. Consider now $\mathbb{S} \in \Gamma(p)$. By definition, $reach_{t,m_\tau(p)}^\mathcal{O}(h) \subseteq SafeLoc(\mathcal{O})$. As before, this means $m_\tau(p)$ cannot be freed. Altogether, we get $p \in valid_{\tau.act}$ as required.

Consider $r \in AVar$. If $isValid(\Gamma(r))$, then $repr_\tau(r) \cap freed_\tau = \emptyset$ by induction. By definition, we get $repr_\tau = repr_{\tau.act}$ and $freed_{\tau.act} = freed_\tau \cup \{a\}$. To arrive at $repr_{\tau.act}(r) \cap freed_{\tau.act} = \emptyset$, it suffices to establish $a \notin repr_\tau(r)$. Towards a contradiction, assume $a \in repr_\tau(r)$. Then, induction gives $reach_{t,a}^\mathcal{O}(h) \subseteq Loc(\Gamma(r))$. As before, however, we get $h.free(a) \notin \mathcal{S}(\mathcal{O})$ from $isValid(\Gamma(r))$. Hence, $a \notin repr_\tau(r)$ must hold as desired.

For the remaining properties, consider some $x \in dom(\Gamma_4)$. Let $b = m_{\tau.act}(x) = m_\tau(x)$ if $x \in PVar$ and $b \in repr_{\tau.act}(x) = repr_\tau(x)$ if $x \in AVar$. By the definition of \mathcal{O}_{Base} and guarantees \mathbb{A}, \mathbb{L} , we have:

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

By the definition of interference freedom and guarantees \mathbb{S}, \mathbb{E}_L , we have:

$$\begin{aligned} reach_{t,b}^\mathcal{O}(h) \subseteq Loc(\mathbb{E}_L) &\implies reach_{t,b}^\mathcal{O}(h.free(a)) \subseteq Loc(\mathbb{E}_L) \\ \text{and } reach_{t,b}^\mathcal{O}(h) \subseteq Loc(\mathbb{S}) &\implies reach_{t,b}^\mathcal{O}(h.free(a)) \subseteq Loc(\mathbb{S}). \end{aligned}$$

Recall from before that $\{\mathbb{A}, \mathbb{L}, \mathbb{S}\} \cap \Gamma(x) \neq \emptyset$ implies $a \neq b$. Hence, the above properties entail the desired $reach_{t,b}^\mathcal{O}(h.free(a)) \subseteq Loc(\Gamma(x))$ because we have $reach_{t,b}^\mathcal{O}(h) \subseteq Loc(\Gamma(x))$ by induction together with $m_\tau = m_{\tau.act}$ and $repr_\tau = repr_{\tau.act}$.

The above case distinction is complete and thus concludes the induction. ■

Proof C.73 (Corollary B.84). Let $\vdash \{\Gamma_{init}^{[t]}\} stmt(\tau, t) \{\Gamma\}$ and $inv(\llbracket P \rrbracket_\emptyset^\mathcal{O})$. Towards a contradiction, assume the claim does not hold. That is, there is a shorted prefix $\sigma.act \in \mathcal{O}\llbracket P \rrbracket_{Adr}^\mathcal{O}$ of τ such that $\sigma.act$ raises a pointer race. By minimality, σ is PRF. By definition, we have $inv(\sigma)$. Let $act = \langle t, com, up \rangle$. By minimality, com is neither `beginAtomic` nor `endAtomic` as otherwise σ would be a pointer race. Lemma B.75 yields Γ_3 with $\vdash \{\Gamma_{init}^{[t]}\} stmt(\sigma.act, t) \{\Gamma_3\}$. We have $stmt(\sigma.act, t) = stmt(\sigma, t); com$. So by the type rules there are $\Gamma_0, \Gamma_1, \Gamma_2$

$$\vdash \{\Gamma_{init}^{[t]}\} stmt(\sigma, t) \{\Gamma_0\} \quad \text{and} \quad \Gamma_0 \rightsquigarrow \Gamma_1 \quad \text{and} \quad \vdash \{\Gamma_1\} com \{\Gamma_2\} \quad \text{and} \quad \Gamma_2 \rightsquigarrow \Gamma_3.$$

We show that act does not raise a pointer race.

◇ **Case 1:** act is an unsafe access

Then, com contains $p.next$ or $p.data$ with $p \notin valid_\sigma$. That is, $\vdash \{\Gamma_1\} com \{\Gamma_2\}$ is derived using one 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 Theorem B.83, we have $p \in valid_\sigma$. Since this contradicts the assumption of act raising an unsafe access, this case cannot apply.

◇ **Case 2:** act is a racy call

Then, $com \equiv in:func(\bar{r})$. That is, $\vdash \{\Gamma_1\} com \{\Gamma_2\}$ is derived using Rule (ENTER). Consider $m_\sigma(\bar{r}) = \bar{v}$ with

$\bar{r} = r_1, \dots, r_k, \dots, r_n$ and $\bar{v} = v_1, \dots, v_n$. Wlog. let $r_1, \dots, r_k \in PExp$ and $r_{k+1}, \dots, r_n \in DExp$. That act is a racy call means that there are $\bar{w} = w_1, \dots, w_n$ and c with:

$$\begin{aligned} & \forall i. (v_i = c \vee r_i \in valid_\sigma \vee r_i \in DExp) \implies v_i = w_i \\ \text{and} \quad & \mathcal{F}_O(h.in:func(t, \bar{w}), c) \notin \mathcal{F}_O(h.in:func, \bar{v}, c). \end{aligned}$$

Let $T_i = \Gamma(r_i)$ for $1 \leq i \leq k$. From Theorem B.83 we get that $r_i \notin valid_\sigma \implies \neg isValid(T_i)$. Hence:

$$\begin{aligned} & \forall i. (v_i = c \vee r_i \in valid_\sigma \vee r_i \in DExp) \implies v_i = w_i \\ \text{implies} \quad & \forall i. v_i \neq w_i \implies (v_i \neq c \wedge r_i \notin valid_\sigma \wedge r_i \notin DExp) \\ \text{implies} \quad & \forall i. v_i \neq w_i \implies (v_i \neq c \wedge \neg isValid(T_i) \wedge r_i \notin DExp) \\ \text{implies} \quad & \forall i. (v_i = c \vee isValid(T_i) \vee r_i \in DExp) \implies v_i = w_i. \end{aligned}$$

Moreover, Theorem B.83 gives $reach_{t,v_i}^O(\sigma) \subseteq Loc(\Gamma(r_i))$ for all $r_i \notin DExp$. Hence, by definition, we arrive at $SafeCall(\Gamma_1, func(\bar{r})) = false$. As this contradicts $\vdash \{\Gamma_1\} com \{\Gamma_2\}$, this case cannot apply.

The above case distinction is complete. Since it yields a contradiction in each case, τ must be PRF. ■

Proof C.74 (Theorem B.85). Let $\vdash \{\Gamma_{init}\} P \{\Gamma_P\}$ and $inv(\llbracket P \rrbracket_\emptyset^\emptyset)$. Towards a contradiction, assume the claim does not hold. That is, there is a shortest computation $\tau.act \in \mathcal{O}[\llbracket P \rrbracket_{Adr}^\emptyset]$ such that $\tau.act$ contains moderate pointer race. By minimality, τ is MPRF. Hence, there is some $\sigma \in \llbracket P \rrbracket_\emptyset^\emptyset$ with $inv(\sigma) \implies inv(\tau)$ by Theorem B.72. Assumption $inv(\llbracket P \rrbracket_\emptyset^\emptyset)$ thus implies $inv(\tau)$. Let $act = \langle t, com, up \rangle$. If $com \equiv @inv \bullet$, then act cannot raise a pointer race and we have $\tau.act$ PRF. Otherwise, we have $inv(\tau.act)$ by definition and thus obtain $\tau.act$ PRF by Corollary B.84. Consequently, $\tau.act$ must be an unsafe assumption. That is, there is some $com' \in next-com(\tau.act)$ with $com' \equiv \text{assume } p = q$ and $\{p, q\} \notin valid_{\tau.act}$. By definition, this means there is $pc \in ctrl(\tau.act)$ with $pc(t') \xrightarrow{com'} \bullet$ for some thread t' . By Lemma B.75, there are Γ_1, Γ_3 with $\vdash \{\Gamma_1\} pc(t') \{\Gamma_3\}$. Then, Lemma B.74 yields Γ_2 such that $\vdash \{\Gamma_1\} com' \{\Gamma_2\}$. This derivation is due to Rule (ASSUME1) or (ASSUME1-CONSTANT). In the latter case, we have $p \in CVar$ or $q \in CVar$ by definition of the type rule. By definition, this means com' is not an unsafe assumption. The case cannot apply. The derivation must be due to Rule (ASSUME1). By definition of the type rule, $\Gamma_1(p) = T$ with $isValid(T)$ and $\Gamma_1(q) = T'$ with $isValid(T')$. Theorem B.83, which is enabled because $\tau.act$ is PRF, gives $\{p, q\} \subseteq valid_\tau$. That is, $\tau.act$ is not prone to an unsafe assumption. Overall, this contradicts $\tau.act$ being a moderate pointer race. Hence, we conclude that $\mathcal{O}[\llbracket P \rrbracket_{Adr}^\emptyset]$ is MPRF as required. ■

Proof C.75 (Theorem B.86). Furthermore, assume $\vdash P$ and $inv(\llbracket P \rrbracket_\emptyset^\emptyset)$. To the contrary, assume that the overall claim does not hold. That is, there is $\tau.act \in \mathcal{O}[\llbracket P \rrbracket_{Adr}^\emptyset]$ with $act = \langle t, com, up \rangle$ such that $com \equiv in:retire(p)$ and $m_\tau(p) = a$ and $a \in retired_\tau$. Lemma B.75 for $\tau.act$ now yields some Γ_3 with $\vdash \{\Gamma_{init}^{[t]}\} stmt(\tau.act, t) \{\Gamma_3\}$. By definition, $stmt(\tau.act, t) = stmt(\tau, t); com$. The type rules give:

$$\vdash \{\Gamma_{init}^{[t]}\} stmt(\tau, t) \{\Gamma_0\} \quad \text{and} \quad \Gamma_0 \rightsquigarrow \Gamma_1 \quad \text{and} \quad \vdash \{\Gamma_1\} com \{\Gamma_2\} \quad \text{and} \quad \Gamma_2 \rightsquigarrow \Gamma_3$$

for some $\Gamma_0, \Gamma_1, \Gamma_2$ where the derivation $\vdash \{\Gamma_1\} com \{\Gamma_2\}$ is due to Rule (ENTER). By definition, this means we have $\mathbb{A} \in \Gamma_1(p)$. Note that $\tau.act$ is MPRF and thus UAF by Theorem B.85. Then, Theorem B.83 yields $reach_{t,a}^O(\tau) \subseteq$

$Loc(\Gamma_1(p))$. In particular, $reach_{t,a}^O(\tau) \subseteq Loc(\mathbb{A})$. Hence, we arrive at $(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau)} (L_2, \varphi)$ for $\varphi = \{z_a \mapsto a\}$. Now, Lemma B.45 gives $a \notin retired_\tau$. This, however, contradicts the assumed $a \in retired_\tau$. ■

Proof C.76 (Theorem B.87). Furthermore, assume $\vdash P$ and $inv(\llbracket P \rrbracket_\emptyset^\emptyset)$. To the contrary, assume the overall claim does not hold. Then, there is a shortest $\tau.act \in \mathcal{O}\llbracket P \rrbracket_{Adr}^\emptyset$ with $\neg inv(\tau.act)$. By minimality, we have $inv(\tau)$. By Theorem B.85, $\tau.act$ is MPRF. Then, Theorem B.72 for $\tau.act$ yields $\sigma \in \llbracket P \rrbracket_\emptyset^\emptyset$ with $inv(\sigma) \implies inv(\tau.act)$. The contrapositive implication and $\neg inv(\tau.act)$ gives $\neg inv(\sigma)$. Hence, $\neg inv(\llbracket P \rrbracket_\emptyset^\emptyset)$. This contradicts the assumption. ■

Proof C.77 (Theorem 8.14). Set $CVar = \emptyset$ so that Assumption A.9 is satisfied. From Corollary B.84 we know that τ is PRF. Consider some thread t and $x \in PVar \cup AVar$. Let $a \in m_\tau(x)$ and $T = \Gamma(x)$. We need to show that $reach_{t,a}^O(\tau) \subseteq Loc(T)$ and $isValid(T) \implies x \in valid_\tau$. Both properties follow by Theorem B.83. ■

Proof C.78 (Theorem 8.15). Set $CVar = \emptyset$ so that $\mathcal{O}\llbracket P \rrbracket_{Adr}^{Adr}$ satisfies Assumption A.9. Theorem B.85 yields $\mathcal{O}\llbracket P \rrbracket_{Adr}^\emptyset$ is MPRF. By $CVar = \emptyset$, this means $\mathcal{O}\llbracket P \rrbracket_{Adr}^\emptyset$ SPRF. ■

Proof C.79 (Theorem 8.16). Set $CVar = \emptyset$ so that $\mathcal{O}\llbracket P \rrbracket_{Adr}^{Adr}$ satisfies Assumption A.9. Theorem B.86 yields $\mathcal{O}\llbracket P \rrbracket_{Adr}^\emptyset$ is DRF. ■

Proof C.80 (Theorem 8.21). Follows from Theorems 8.15 and B.87. ■

Proof C.81 (Theorem 8.22). Proof given in Section 8.5. ■

Proof C.82 (Theorem 8.24). Proof given in Section 8.6. ■

Proof C.83 (Theorem 8.26). Proof given in Section 8.6. ■

Proof C.84 (Theorem A.11). Follows from Theorems B.85 to B.87. ■

Proof C.85 (Proposition A.12). By definition of $active(\bullet)$ and Assumption A.9. ■