Verifying Non-blocking Data Structures with Manual Memory Management

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
geboren am 22.03.1990
in Torgau

Eingereicht am: 03. März 2021
Disputation am: 25. Juni 2021
1. Referent: Prof. Roland Meyer
2. Referent: Prof. Rupak Majumdar
3. Referent: Prof. Constantin Enea

2021
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.
Zusammenfassung


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’Osualdo, Prakash Saivasan, Elisabeth Neumann, Sören van der Wall, Mike Becker, Thomas Haas, Elaine Anklam, and Johannes Mohr.
Preface

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

Relevant for: Chapter 6.

Relevant for: Chapters 1 to 5 and 7.

Relevant for: Chapters 1, 3, 8 and 9.

Further publications related to this thesis:


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


Relevant for: Appendices B and C.

Relevant for: Appendices A to C.

A web page accompanying this thesis is available at: https://wolff09.github.io/phd/
# Contents

1 Introduction ............................... 1  
   Contribution 1: SMR Specifications and Compositional Verification  .......... 3  
   Contribution 2: Ownership for Manual Memory Reclamation  ................. 4  
   Contribution 3: Avoiding Reallocations  ....................................... 5  
   Contribution 4: Verification under Garbage Collection  ...................... 6  
   Outlook ...................................... 7  

I Preliminaries .................................. 9  

2 Non-blocking Data Structures ............. 11  
   2.1 Linearizability ............................ 11  
   2.2 Fine-grained Synchronization .......... 13  
   2.3 Manual Memory Reclamation .......... 14  
       2.3.1 Free Lists ........................... 16  
       2.3.2 Epoch-Based Reclamation .......... 18  
       2.3.3 Hazard Pointers ..................... 20  
   2.4 Data Structure Implementations ...... 23  
       2.4.1 Stacks .................................. 24  
       2.4.2 Queues .................................. 26  
       2.4.3 Sets ................................... 28  

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 .................. 43  

II Contributions ............................... 45  

5 Compositional Verification ............... 47  
   5.1 SMR Automata ............................. 48
## Ownership and Reclamation

6.1 Reclamation breaks Ownership .................................................. 55
6.2 Regaining Ownership .................................................................. 60
6.3 Evaluation .................................................................................. 61
   6.3.1 Integrating Safe Memory Reclamation .................................. 62
   6.3.2 Linearizability Experiments .................................................. 62

## Pointer Races

7.1 Similarity of Computations .......................................................... 65
7.2 Preserving Similarity .................................................................... 69
7.3 Detecting ABAs .......................................................................... 73
7.4 Reduction Result ......................................................................... 76
7.5 Evaluation .................................................................................. 78
   7.5.1 Soundness checks ................................................................. 78
   7.5.2 Linearizability Experiments .................................................. 79
   7.5.3 Verifying SMR Implementations .............................................. 81

## Strong Pointer Races

8.1 Annotations ................................................................................. 85
8.2 Avoiding All Reallocations .......................................................... 86
8.3 A Type System to Prove Strong Pointer Race Freedom ................. 88
   8.3.1 Guarantees ........................................................................... 89
   8.3.2 Types .................................................................................. 90
   8.3.3 Type Rules .......................................................................... 92
   8.3.4 Soundness .......................................................................... 95
8.4 Example ..................................................................................... 97
   8.4.1 Type Transformer Relation .................................................. 98
   8.4.2 Angels ................................................................................. 99
   8.4.3 Typing ................................................................................ 100
   8.4.4 Annotations ....................................................................... 101
   8.4.5 Hazard Pointers .................................................................. 101
8.5 Invariant Checking ...................................................................... 103
8.6 Type Inference ............................................................................ 105
8.7 Avoiding Strong Pointer Races ................................................... 108
8.8 Evaluation .................................................................................. 111
# List of Figures and Tables

2.2 Pseudo implementations for compare-and-swap .......................... 13
2.3 A simple counter .................................................. 15
2.4 Implementation of free lists (FL) .................................. 16
2.5 A simple counter with FL ........................................ 17
2.6 A simple counter with EBR ....................................... 18
2.7 Implementation of epoch-based reclamation (EBR) ..................... 19
2.8 Implementation of hazard pointers (HP) ............................. 21
2.9 A simple counter with HP .......................................... 22
2.10 Node type for singly-linked data structures .......................... 23
2.11 Treiber’s stack ...................................................... 25
2.12 Optimized version of Treiber’s stack ................................. 25
2.13 Michael&Scott’s queue ............................................. 26
2.14 DGLM queue ............................................................. 27
2.15 Singly-linked set insertion .......................................... 28
2.16 Vechev&Yahav’s 2CAS set ......................................... 30
2.17 ORVYY set ............................................................... 31
2.18 Vechev&Yahav’s CAS set .......................................... 33
2.19 Michael’s set .......................................................... 34
2.20 Harris’ set ............................................................... 35
3.1 Semantics of commands .............................................. 39
3.2 SOS rules for the standard semantics ................................ 40
5.1 Typical system design and the interaction among components .... 47
5.4 SMR automata $O_{\text{Base}}$, $O_{\text{EBR}}$, and $O_{\text{HP}}^k$ ............ 50
5.9 SMR semantics ............................................................. 53
6.2 Spurious views encountered during thread-modular interference ... 57
6.4 Unsound ownership reasoning due to reallocations .................. 59
6.8 Experiments for verifying data structures using FL (state space exploration) .... 63
7.24 Experiments for verifying data structures using SMR (state space exploration) ... 80
7.25 Experiments for verifying SMR implementations (state space exploration) ..... 82
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 manual 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 rational 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 GC.

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
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 progress 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 two linearizable components is linearizable again [Herlihy and Shavit 2008, Section 3.5], allowing for the components to be verified 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 $evt = in: meth(t, \overline{v})$ where $meth$ is the invoked method, $t$ is the invoking thread, and $\overline{v}$ are the actual parameters. Responses take the form $evt = re: meth(t, \overline{v})$ where $meth$ is the returning method, $t$ is the executing thread, and $\overline{v}$ 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 $E'|_t$.

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 $E'$.  

---

12 Chapter 2 Non-blocking Data Structures
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 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

---

\(^1\) 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].

```c
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 }
7
8 bool CAS(T& dst1, T cmp1, T src1, T& dst2, T cmp2, T src2) { // double-word / two-word
9    atomic {
10       if (dst1 == cmp1 && dst2 == cmp2) { dst1 = src1; dst2 = src2; return true; }
11       else { return false; }
12    }
13 }
```
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 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`
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.

```c
12 struct Container { 22 int increment() { 13 int data; 23 Container* inc = new Container(); 14 }; 24 while (true) { 15 shared Container* Counter; 25 Container* curr = Counter; 16 atomic init() { 26 int out = curr->data; 19 Counter = new Container(); 27 inc->data = out+1; 20 Counter->data = 0; 28 if (CAS(Counter, curr, inc)) { 21 } } } } 
```

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].

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 function\(^2\) 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

\(^2\)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.
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 it is accessed by a single thread only.

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 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 $t'$. 

---

**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.

```c
32 threadlocal list<T*> freeList;
33
34 void retire(T* pointer) {
35    freeList.push(pointer);
36 }
37
38 T* reuse() {
39    if (freelist.empty()) return NULL;
40    T* result = freeList.pop();
41    return result;
42 }
```
Let thread \( t' \) increment the counter, installing value \( \text{out}+1 \) and retiring address \( a \). If \( t' \) performs another increment, it might reuse address \( a \) to install \( \text{out}+2 \). Now, the \text{CAS} of \( t \) succeeds although the counter has been updated: \( t \) erroneously decreases the counter’s value from \( \text{out}+2 \) to \( \text{out}+1 \) where an increase to \( \text{out}+3 \) 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 \text{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 \text{CAS} [Michael 2002a] or by \textit{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 \textit{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.
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 lag 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 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 epoch $e - 1$ 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 $e + 1$. 

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

```c
struct EbrRec {
  EbrRec* next;
  bool used;
  int epoch;
  List<T*> retired0, retired1, retired2;
};

shared int GEpoch;
shared EbrRec* LEpochs;
threadlocal EbrRec* myEpoch;

atomic init () {
  Epochs = NULL;
  GlobalEpoch = 0;
}

void join () {
  myEpoch = new EbrRec();
  myEpoch->used = true;
  myEpoch->epoch = GEpoch;
  while (true) {
    EbrRec* recs = LEpochs;
    myEpoch->next = recs;
    if (CAS(LEpochs, recs, myEpoch)) {
      break;
    }
  }
  int nextEpoch = (epoch + 1) % 3;
  if (!CAS(GEpoch, epoch, nextEpoch)) {
    return;
  }
  myEpoch->epoch = nextEpoch;
  for (T* ptr : myEpoch->retired2) {
    delete ptr;
  }
  retired2.clear();
  retired2.swap(retired1);
  retired1.swap(retired0);
}

void retire(T* ptr) {
  myEpoch->retired0.push(ptr);
}

void leave0 () {
  int epoch = GEpoch;
  myEpoch->epoch = epoch;
}

void part () {
  myEpoch->used = false;
}

void enterQ () {
}

void part () {
  myEpoch->used = false;
}

void enterQ () {
}
```

Section 2.3 Manual Memory Reclamation
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 97. 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 retire0, Line 121. Function leaveQ implements the epoch progression, Lines 128 to 138, and deferred deletion process, Lines 140 to 146, 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 106 to 114. Parting threads mark their epoch entry as inactive via the used flag of EbrRec, Line 118. 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 reclamation 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 hp_{t}[i] and may drop the thread subscript if clear...
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.

```c
struct HpRec {
    HpRec* next;
    Array<T*, K> hp; // 0-indexed
    List<T*> retired;
}

shared HpRec* HPtrs;
threadlocal HpRec* myHP;

atomic init() {
    HPtrs = NULL;
}

void join() {
    myHP = new HpRec();
    while (true) {
        HpRec* recs = HPtrs;
        myHP->next = recs;
        if (CAS(HPtrs, recs, myHP)) {
            break;
        }
    }
}

void part() {
    for (int i = 0; i < K; ++i) {
        hp[i] = NULL;
    }
}

atomic init() {
    HPtrs = NULL;
}

void join() {
    myHP = new HpRec();
    while (true) {
        HpRec* recs = HPtrs;
        myHP->next = recs;
        if (CAS(HPtrs, recs, myHP)) {
            break;
        }
    }
}

void retire(T* ptr, int index) {
    myHP->hp[index] = ptr;
}

void unprotect(int index) {
    protect(NULL, index);
}

void protect(T* ptr, int index) {
    assert(0 <= index < K);
    myHP->hp[index] = ptr;
}

void retire(T* ptr, int index) {
    myHP->retired.push(ptr);
    if (*) reclaim();
}

void reclaim() {
    for (T* ptr : myHP->retired) {
        if (defer.contains(ptr)) continue;
        myHP->retired.remove(ptr);
        delete ptr;
    }
}
```

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 179. Protections can be revoked by unprotect which takes a hazard pointer index and resets the corresponding hazard pointer by writing NULL to it, Line 183. Protections are respected as follows. Function retire stores the passed object into a thread-local list of retired objects, Line 187. Moreover, it periodically tries to reclaim the objects from that list, Line 188. To do so, it scans the hazard pointers of all threads, collecting all the objects that are currently protected, Lines 192 to 199. Then, a retired object is reclaimed if it is not in the list of protected objects, Lines 201 to 205. 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 \texttt{curr} because it is subsequently accessed. To that end, a protection is issued using hazard pointer \texttt{hp}. At this point, we cannot guarantee that a dereference of \texttt{curr} is safe. Between reading out \texttt{curr} in Line 219 and protecting it in Line 220, an interfering thread might have updated the counter and retired the object referenced by \texttt{curr}. Because the protection was not yet announced, \texttt{curr} might have already been reclaimed. Line 221 checks that this is not the case. It does so by ensuring that \texttt{curr} coincides with the shared \texttt{Counter}. It is worth pointing out that this check relies on the invariant that \texttt{Counter} is never retired. Only after both Lines 220 and 221 have been executed, \texttt{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 221, is prone to the ABA problem. Indeed, as noted above the object referenced by \texttt{curr} could have been reclaimed. Consequently, it could have been reused and installed as the shared \texttt{Counter} again. Those scenarios are not problematic since we just ensure that \texttt{curr} contains the current value of \texttt{Counter} and that is has been protected successfully. Put differently, Lines 219 to 221 appear as if they were executed atomically. Accesses to the content of \texttt{curr} happen only later.
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 192 to 199 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.

```c
struct Node {
    int data;
    bool mark;
    Node* next;

    Node(int value) { data = value; mark = false; next = NULL; } // constructor
}
```
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 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 implementation 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 244, linking the new node as a predecessor of top, Line 247, and installing node as the new ToS via a CAS, Line 248. 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 257. If top equals NULL, then the implementation signals that the stack is empty, Line 258. 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 261. 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 264. 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 245 and 246 in push as well as Lines 259 and 260 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 extension to HP is due to Michael [2002b].

```
236  shared Node* ToS;
237
238  atomic init() { ToS = NULL; }
239
240  void push(int input) {
241    leaveQ();
242    Node* node = new Node(input);
243    while (true) {
244      Node* top = ToS;
245      if (top != ToS) continue;
246      node->next = top;
247      if (CAS(&ToS, top, node)) break;
248    }
249    enterQ();
250  }
251
252  int pop() {
253    int output = EMPTY;
254    while (true) {
255      Node* top = ToS;
256      if (top == NULL) break; // empty
257      if (top != ToS) continue;
258      Node* next = top->next;
259      if (CAS(&ToS, top, next)) {
260        output = top->data;
261        retire(top);
262      } }
263    enterQ();
264    return output;
265  }
```

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.

```
270  shared Node* ToS;
271
272  atomic init() { ToS = NULL; }
273
274  void push(int input) {
275    // no SMR needed
276    Node* node = new Node(input);
277    while (true) {
278      Node* top = ToS;
279      node->next = top;
280      if (CAS(&ToS, top, node)) break;
281    }
282  }
283
284  int pop() {
285    int output = EMPTY;
286    while (true) {
287      Node* top = ToS;
288      if (top == NULL) break; // empty
289      if (top != ToS) continue;
290      Node* next = top->next;
291      if (CAS(&ToS, top, next)) {
292        output = top->data;
293        retire(top);
294      } }
295    return output;
296  }
```

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 280 (Line 248 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.
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 due to Michael and Scott [1996]. It is a practical example in that it is used for Java’s ConcurrentLinkedQueue [Oracle 2020] and C++ Boost’s lockfree::queue [Blechmann 2011], 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 enqueuer first moves Tail to the last node as it may lack behind, Line 312. Then, the new node is appended by pointing Tail->next to it, Line 315. Last, the enqueuer tries to move Tail to the new node, Line 316. This can fail as another thread may have moved Tail already to avoid waiting for the enqueuer. 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 337, and then moves the Head to that node, Line 338. Additionally, dequeue...
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.

346 shared Node* Head, Tail;
347
348 atomic init() { Head = Tail = new Node(_); }
349
350 void enqueue(int input) {
351   leaveQ;
352   Node* node = new Node(input);
353   while (true) {
354     Node* tail = Tail;
355     protect(tail, 0);
356     if (tail != Tail) continue;
357     Node* next = tail->next;
358     if (next != NULL) {
359       CAS(&Tail, tail, next);
360       continue;
361     }
362     if (CAS(&Tail->next, next, node)) {
363       Node* tail = Tail;
364       if (head == tail) {
365         CAS(Tail, tail, next);
366       }
367     } 
368     enterQ;
369   }
370   int dequeue() {
371     leaveQ;
372     int output = EMPTY;
373     while (true) {
374       Node* head = Head;
375       protect(head, 0);
376       if (head != Head) continue;
377       Node* next = head->next;
378       if (next == NULL) break; // empty
379       if (next != NULL) {
380         if (CAS(&Head, head, next)) {
381           Node* tail = Tail;
382           if (head == tail) {
383             CAS(Tail, tail, next);
384           }
385           if (next == NULL) break; // empty
386           if (next != NULL) {
387             if (next == head) {
388               break;
389             }
390             if (next != head) {
391               unprotect(0); unprotect(1);
392               return output;
393             }
394             } 
395           retize(head);
396           break;
397         }
398         unprotect(0); unprotect(1);
399         enterQ;
400       }
401     } 
402     return output;
403   }

maintains the property that Head does not overtake Tail by moving Tail towards the end of the list if necessary, Line 335.

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 339. The modifications required for FL and EBR are straightforward, 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 327. 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 330. Otherwise, the operation is restarted. Second, next is protected with hp[1], Line 331. If head and Head coincide, Line 332, 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.
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 $\text{CAS}(a->next, b, c)$ is prone to this problem.

Observe that $\text{dequeue}$ reads out the to-be-returned data value $\text{next->value}$, Line 337, before the actual dequeuing, Line 338. This is done because of FL. There, it is possible that immediately after the $\text{CAS}$ from Line 338 the node referenced by $\text{next}$ is dequeued, retired, and reused. The reuse leads to $\text{next->value}$ 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 $\text{CAS}$ is correct because the reuse of $\text{next}$ is prevented.

**DGLM Queue.** Doherty et al. [2004b] proposed a variation of Michael&Scott’s implementation, see Figure 2.14. Their $\text{dequeue}$ method avoids congestion on the Tail pointer. It does so 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, $\text{dequeue}$ 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 $\text{next}$ field of node $a$ from $b$ to $c$. However, a simple $\text{CAS}(a->next, b, c)$ 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 $b->\text{next}$ to $d$ (dashed line in Figure 2.15). Then, the above $\text{CAS}$ 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 454.

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 457. 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 418. Otherwise, a new node is linked in-between pred and curr, Lines 419 and 420. Method remove works similarly. To ensure a correct unlinking, two-word CAS is used, Line 435. 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.

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 449 to 456 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 454 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.

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 500 to 505. 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.
Figure 2.16: Vechev and 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 435.

```cpp
shared Node* Head, Tail;
atomic init() {
  Head = new Node(-\infty);
  Tail = new Node(\infty);
  Head->next = Tail;
}

bool contains(int value) {
  Node* pred; Node* curr; int found;
  <pred, curr, found> = locate(value);
  H unprotected(0); unprotected(1);
  E enterQ();
  return found == value;
}

bool insert(int value) {
  Node* pred; Node* curr; int found;
  Node* entry = new Node(value);
  E leaveQ();
  bool success = false;
  while (!success) {
    <pred, curr, found> = locate(value);
    if (found == value) break;
    entry->next = curr;
    success = 2CAS(pred->next, curr, entry);
  }
  return success;
}

bool remove(int value) {
  Node* pred; Node* curr; int found;
  <pred, curr, found> = locate(value);
  if (found > value) break;
  Node* next = curr->next;
  success = 2CAS(pred->next, curr, next,
                 curr->next, next, NULL);
  if (!success) retire(entry);
  unprotected(0); unprotected(1);
  E enterQ();
  return success;
}

<Node*, Node*, int> locate(int value) {
  Node* pred; Node* curr; int found;
  assert(-\infty < value < \infty);
  retry: // jump label
  curr = Head;
  do {
    pred = curr;
    if (found == value) break;
    entry->next = curr;
    success = CAS(pred->next, curr, entry);
  } while (success);
  return success;
}
```

Chapter 2  Non-blocking Data Structures
Figure 2.17: The ORVYY set [O’Hearn et al. 2010] with SMR. The implementation of `remove` relies on a two-word CAS, Lines 500 to 505.

```c
shared Node* Head, Tail;

atomic init() {
    Head = new Node(-\infty);
    Tail = new Node(\infty);
    Head->next = Tail;
}

bool contains(int value) {
    Node* pred; Node* curr; int found;
    leaveQ();
    <pred, curr, found> = locate(value);
    H
    unprotect(0); unprotect(1);
    E
    enterQ();
    return found == value;
}

bool insert(int value) {
    Node* pred; Node* curr; int found;
    Node* entry = new Node(value);
    E
    leaveQ();
    bool success = false;
    while (!success) {
        <pred, curr, found> = locate(value);
        if (found == value) break;
        entry->next = curr;
        success = CAS(pred->mark, false, false,
                       pred->next, curr, entry);
    }
    F
    E
    H
    if (!success) retire(entry);
    H
    unprotect(0); unprotect(1);
    E
    enterQ();
    return success;
}

bool remove(int value) {
    Node* pred; Node* curr; int found;
    E
    leaveQ();
    bool success = false;
    while (!success) {
        <pred, curr, found> = locate(value);
        if (found > value) break;
        atomic { if (!pred->mark && pred->next == curr) {
            pred->next = curr->next;
            curr->mark = true;
            success = true;
        } }
        F
        E
        H
        if (success) retire(curr);
        H
        unprotect(0); unprotect(1);
        E
        enterQ();
        return success;
}

<Node*, Node*, int> locate(int value) {
    Node* pred; Node* curr; int found;
    retry: // jump label
    curr = Head;
    do {
        pred = curr;
        assert(-\infty < value < \infty);
        H
        protect(pred, 1);
        E
        if (success) retire(curr);
        H
        unprotect(0); unprotect(1);
        E
        enterQ();
        retry: // jump label
        if (found < value) {
            found = curr->data;
            do {
                pred = curr;
                assert(-\infty < value < \infty);
                H
                protect(pred, 1);
                E
                if (success) retire(curr);
                H
                unprotect(0); unprotect(1);
                E
                enterQ();
                retry: // jump label
                while (found < value);
            } while (found < value);
            return <pred, curr, found>;
        }
    } while (found < value);
```
Vechev&Yahav’s CAS Set. Towards both practical and non-blocking implementations, Vechev and Yahav [2008] showed that the aforementioned marking technique allows for removals with double-word CAS (or standard single-word CAS if the mark is implemented using bit stealing). Consider Figure 2.18 for the implementation. The removal of a node \( \text{curr} \) is performed in two steps. First, a double-word CAS sets the mark flag, Line 570. As for the ORVYY set, this prevents other threads from updating node \( \text{curr} \). Then, another double-word CAS unlinks \( \text{curr} \) by redirecting \( \text{pred} \rightarrow \text{next} \), Lines 573 and 574. The latter CAS goes through only if \( \text{pred} \) is unmarked, ensuring that the removal does not interfere with concurrent removals of \( \text{pred} \).

It is worth pointing out that the removal is considered successful only if \( \text{curr} \) 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.

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 \( \text{locate} \) eagerly performs physical removals of all logically removed nodes it encounters during its traversal, Lines 616 to 622, and method \( \text{remove} \) may return if it logically removed but failed to physically remove a node.

Harris’s Set. Harris [2001] proposed a lazy version of the \( \text{locate} \) method for Michael’s set: instead of removing individually all logically removed nodes, sequences of subsequent logically removed nodes are deleted. To that end, \( \text{locate} \) 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.
shared Node* Head, Tail;
atomic_init() {
Head = new Node(-∞);
Tail = new Node(∞);
Head->next = Tail;
}

bool contains(int value) {
Node* pred; Node* curr; int found;
leaveQ;
<pred, curr, found> = locate(value);
unprotect(0); unprotect(1);
enterQ;
return found == value;
}

bool insert(int value) {
Node* pred; Node* curr; int found;
Node* entry = new Node(value);
leaveQ();
bool success = false;
while (!success) {
<pred, curr, found> = locate(value);
if (found == value) break;
entry->next = curr;
success = CAS(pred->mark, false, false, pred->next, curr, entry);
}
unprotect(0); unprotect(1);
enterQ();
return success;
}

bool remove(int value) {
Node* pred; Node* curr; int found;
leaveQ();
Node* entry = new Node(value);
leaveQ();
bool success = false;
while (!success) {
<pred, curr, found> = locate(value);
if (found == value) break;
entry->next = curr;
success = CAS(pred->mark, false, false, pred->next, curr, entry);
}
unprotect(0); unprotect(1);
enterQ();
return success;
}

return found == value;
}

bool insert(int value) {
Node* pred; Node* curr; int found;
Node* entry = new Node(value);
leaveQ();
bool success = false;
while (!success) {
<pred, curr, found> = locate(value);
if (found == value) break;
entry->next = curr;
success = CAS(pred->mark, false, false, pred->next, curr, entry);
}
unprotect(0); unprotect(1);
enterQ();
return success;
}

bool remove(int value) {
Node* pred; Node* curr; int found;
leaveQ();
Node* entry = new Node(value);
leaveQ();
bool success = false;
while (!success) {
<pred, curr, found> = locate(value);
if (found == value) break;
entry->next = curr;
success = CAS(pred->mark, false, false, pred->next, curr, entry);
}
unprotect(0); unprotect(1);
enterQ();
return success;
}

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

```c
shared Node* Head;

atomic init() {
  Head = new Node(-\infty);
}

<Node*, Node*, int> locate(int value) {
  Node* pred; Node* curr; int found;
  assert(-\infty < value < \infty);
  retry: // jump label
  curr = Head;
  do {
    pred = curr;
    H protect(pred, 1);
    curr = pred->next;
    H protect(curr, 0);
    if (pred->mark) goto retry;
    if (pred->next != curr) goto retry;
    found = curr->data;
    if (curr->mark) {
      Node* next = curr->next;
      if (CAS(pred->mark, false, false, pred->next, curr, next)) {
        H E F retire(curr);
        goto retry;
      } }
  } while (found < value);
  return <pred, curr, found>;
}

bool contains(int value) {
  bool success = false;
  while (!success) {
    <pred, curr, found> = locate(value);
    if (found == value) break;
    entry->next = curr;
    success = CAS(pred->mark, false, true, curr->next, curr, entry);
  } return success;
}

bool insert(int value) {
  Node* pred; Node* curr; int found;
  Node* entry = new Node(value);
  & leaveQ();
  & unprotect(0); unprotect(1);
  & enterQ();
  return success;
}

bool remove(int value) {
  Node* pred; Node* curr; int found;
  leaveQ();
  bool success = false;
  while (!success) {
    <pred, curr, found> = locate(value);
    if (found > value) break;
    Node* next = curr->next;
    success = CAS(curr->mark, false, true, curr->next, curr, next);
  } if (success) {
    if (!CAS(pred->mark, false, false, pred->next, curr, next))
      locate(value);
    H E F else retire(curr);
  } return success;
}
```

Chapter 2  Non-blocking Data Structures
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.

```c
shared Node* Head, Tail;
atomic init() {
    Head = new Node(-∞);
    Tail = new Node(∞);
    Head->next = Tail;
}

bool unlink(Node* left, Node* lnext, Node* right) {
    if (lnext == right) return true;
    if (CAS(left->mark, false, false
              left->next, lnext, right)) {
        while (lnext != right) {
            retire(lnext);
            lnext = lnext->next;
        }
        return true;
    }
    return false;
}

<Node*, Node*, int> locate(int value) {
    Node* left; Node* lnext; int found;
    assert(-∞ < value < ∞);
    while (true) {
        Node* right = Head;
        bool rmark = Head->mark;
        Node* rnext = Head->next;
        do {
            if (!rmark) {
                left = right;
                lnext = rnext;
            } else {
                right = rnext;
                if (right == Tail) break;
                rmark = right->mark;
                rnext = right->next;
                if (rmark || found < value) break;
            }
            if (!CAS(left->mark, false, false
                      left->next, right, rnext))
                if (right == Tail || !right->mark)
                    locate(value);
            else retire(right);
        }
        return <left, right, found>;
    }
}

bool contains(int value) {
    Node* left; Node* right; int found;
    <left, right, found> locate(value);
    return found == value;
}

bool insert(int value) {
    Node* left; Node* right; int found;
    Node* entry = new Node(value);
    leaveQ();
    <left, right, found> locate(value);
    if (found == value) break;
    entry->next = right;
    bool success = CAS(left->mark, false, false
                         left->next, right, entry);
    if (!success) retire(entry);
    enterQ();
    return success;
}

bool remove(int value) {
    Node* left; Node* right; Node* rnext;
    int found; bool success = false;
    leaveQ();
    while (!success) {
        <left, right, found> locate(value);
        if (found != value) break;
        if (right->mark) continue;
        rnext = right->next;
        success = CAS(right->mark, false, true
                       right->next, rnext, rnext);
        if (success) {
            if (!CAS(left->mark, false, false
                      left->next, right, rnext))
                locate(value);
            else retire(right);
        } else retire(left);
    }
    return success;
}
```
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 $\bullet$ 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 $\textit{shared} \subseteq PVar$. Additionally, we allow pointer expressions to hold the special value $\textit{seg} \notin Adr$ denoting undefined/uninitialized pointers.

There is also an underlying data domain $Dom$ to which data expressions $DExp = DVar \cup DSel$ 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 $\textit{next}$ 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:

$$m : \{PExp \mapsto Adr \cup \{\text{seg}\}\} \cup \{DExp \mapsto Dom\}.$$  

The initial memory is $m_\epsilon$. Pointer variables $p$ are uninitialized, $m_\epsilon(p) = \text{seg}$. 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 $\tau$ which give rise to sequences of updates. We write $m_\tau$ for the memory resulting from the initial memory $m_\epsilon$ when applying the sequence of updates in $\tau$.
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:

\[
stmt ::= stmt; 
| stmt \oplus stmt 
| stmt^* 
| beginAtomic; stmt; endAtomic 
| com
\]

\[
com ::= p := q 
| p := q.next 
| p.next := q 
| u := op(\overline{u}) 
| u := q.data 
| p.data := u
\]

\[
cond ::= p = q 
| p \neq q 
| pred(\overline{u})
\]

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 \( \overline{r} \) is short for \( r_1, \ldots, r_n \) and similarly for \( \overline{u} \). 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 \( \langle P(R) \rangle^X \) of computations. It is defined relative to two sets \( Y \subseteq X \subseteq Adr \) of addresses allowed to be reallocated and freed, respectively. A computation is a sequence \( \tau \) of actions \( act \) that are of the form \( act = (t, com, up) \). The action indicates that thread \( t \) executes command \( com \) which results in the memory update \( up \). To make the semantics precise, let \( fresh_\tau \subseteq Adr \) be the set of addresses which have never been allocated in \( \tau \) and let \( freed_\tau \subseteq Adr \) 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, \( e \in \langle P(R) \rangle^X \). An action \( act \) can be appended to a computation \( \tau \in \langle P(R) \rangle^X \), denoted by \( \tau.act \in \langle P(R) \rangle^X \), 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.

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

### 3.4 Semantics of Programs

We give a small-step operational semantics (SOS) for programs [Plotkin 1981]. To than end, we define a transition relation \( \implies \) among pairs \( (pc, r) \) of control locations \( pc \) and computations \( r \).

Intuitively, \( \implies \) 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:

\[
st ::= stmt \mid \text{inatomic } st \mid st_1 \circ st_2 \mid \text{await func}
\]
Figure 3.2: SOS rules for the standard semantics, $\{P(R)\}_X$.

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

\[
\begin{align*}
\text{(sos-std-com)} & : \text{com} \xrightarrow{\text{com}} \text{skip} & \text{(sos-std-seq1)} & : \text{skip}; \text{st} \xrightarrow{\text{com}} \text{st} \\
\text{(sos-std-seq2)} & : \text{st}_1 \xrightarrow{\text{com}} \text{st}_1' & \text{(sos-std-choice)} & : \text{st}_1; \text{st}_2 \xrightarrow{\text{com}} \text{st}_1' \text{; } \text{st}_2 \\
\text{(sos-std-loop1)} & : \text{st}^* \xrightarrow{\text{com}} \text{skip} & \text{(sos-std-loop2)} & : \text{st}^* \xrightarrow{\text{com}} \text{st}^* \\
\text{(sos-std-atomic1)} & : \text{beginAtomic}; \text{st}; \text{endAtomic} \xrightarrow{\text{beginAtomic}} \text{inatomic st} & \text{(sos-std-atomic2)} & : \text{inatomic st} \xrightarrow{\text{com}} \text{inatomic st}' \\
\end{align*}
\]

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

\[
\begin{align*}
\text{(sos-std-call)} & : \text{st} \xrightarrow{\text{in}} \text{func}(\tau) \xrightarrow{\text{com}} \text{st}' & \text{(sos-std-return)} & : \text{st} \xrightarrow{\text{re}} \text{func} \xrightarrow{\text{com}} \text{st}' \\
\text{(sos-std-ds)} & : \text{st}_1 \xrightarrow{\text{com}} \text{st}_1' & \text{st} \xrightarrow{\text{in}} \text{func}(\tau) \xrightarrow{\text{com}} \text{st}^* \\
\text{(sos-std-smr)} & : \text{st}_2 \xrightarrow{\text{com}} \text{st}_2' & \text{st} \xrightarrow{\text{in}} \text{func}(\tau) \xrightarrow{\text{com}} \text{st}^* \\
\end{align*}
\]

(c) SOS transition relation $\vdash$. 

\[
\begin{align*}
\text{(sos-std-env)} & : \text{act} \in \text{Act}(r, \bot, \text{env}(a)) & \text{(sos-std-par)} & : \text{act} \in \text{Act}(r, t, \text{com}) \\
\text{(pc, \tau)} \xrightarrow{s} (\text{pc}, \tau, \text{act}) & \text{act} \in \text{Act}(r, t, \text{com}) \\
\text{st} \xrightarrow{\text{com}} \text{st}' & \text{act} \in \text{Act}(r, t, \text{com}) \\
\exists t'. t \neq t' & \land \text{locked}\langle pc(t') \rangle \\
(pc[t \mapsto \text{st}], \tau) \xrightarrow{s} (pc[t \mapsto \text{st}'], \tau, \text{act}) & \text{act} \in \text{Act}(r, t, \text{com})
\end{align*}
\]
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 $\vdash$ is based on a control-flow relation $\text{com}$ among statements $st$. More precisely, $\vdash$ $\text{com}$ $st'$ indicates that performing a step of $st$ executes command $\text{com}$ after which $st'$ remains to be executed. Formally, $\vdash$ and $\text{com}$ are the smallest relations that satisfies 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 $st_1 \triangleright st_2$. 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 $\text{skip}$ if no function is invoked at the moment. Rule $(\text{sos-std-call})$ looks up the code of the invoked function, $R.func$, and appends command $\text{await}$ $func$. We use $\text{await}$ $func$ to synchronize the callee with the caller, Rule $(\text{sos-std-return})$. This ensures that invocations $\text{in:func}(\tau)$ receive a matching response $\text{re:func}$. Rules $(\text{sos-std-ds})$ and $(\text{sos-std-smr})$ 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 $\text{com}$ executed by threads $t$ into actions, we write $\text{Act}(\tau, t, \text{com})$ to obtain the set of actions $\text{act} = \langle t, \text{com}, \text{up} \rangle$ such that $\tau.\text{act}$ satisfies the semantics of commands defined in Section 3.3 above. Rule $(\text{sos-std-env})$ updates unallocated memory non-deterministically, simulating the environment. Rule $(\text{sos-std-par})$ executes a step of thread $t$ if no other thread is currently within an atomic block, as defined by:

\[
\begin{align*}
\text{locked(inatomic } st) & := true \\
\text{locked}(st_1; st_2) & := \text{locked}(st_1) \\
\text{locked}(st_1 \triangleright st_2) & := \text{locked}(st_1) \lor \text{locked}(st_2) \\
\text{locked}(st) & := false \quad \text{otherwise}.
\end{align*}
\]

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

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

Here, $\vdash^*$ is the reflexive transitive closure of $\vdash$. The initial control location $pc_{\text{init}}$ 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 $(\text{sos-std-call})$. 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 $\tau$:

\[
\text{ctrl}(\tau) := \{ pc \mid (pc_{\text{init}}, \epsilon) \vdash^* (pc, \tau) \}.
\]
Proving a data structure correct for an arbitrary number of client threads requires a thread-modular analysis [Berdine et al. 2008; 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

$$V = V \cup \text{seq}(V) \cup \text{int}(V).$$

Function $\text{seq}$ computes a sequential step, the views obtained from letting each thread execute an action on its own views. Function $\text{int}$ 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, $\text{int}$ is computed by combining two views, letting one thread perform an action, and projecting the result to the other thread. More precisely, computing $\text{int}(V)$ requires for every pair of views $v_1, v_2 \in V$ to (i) compute a combined view $\omega$ of $v_1$ and $v_2$, (ii) perform for $\omega$ 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 used by 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
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 $O$ 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 $l \xrightarrow{f(r),g} l'$ with locations $l, l'$, event $f(R)$, and guard $g$. Events $f(R)$ consist of a type $r$ and parameters $R = r_1, \ldots, r_n$. The guard is a Boolean formula over equalities of variables and the parameters $R$. An SMR automaton state $s$ is a tuple $(l, \phi)$ where $l$ is a location and $\phi$ maps variables to values. Such a state is initial if $l$ is initial, and similarly accepting if $l$ is accepting. Then, $(l, \phi) \xrightarrow{f(R),g} (l', \phi)$ is an SMR automaton step, if $l \xrightarrow{f(R),g} l'$ is a transition and $\phi(g[R \mapsto \overline{r}])$ evaluates to true. With $\phi(g[R \mapsto \overline{r}])$ we mean $g$ where the formal parameters $R$ are replaced with the actual values $\overline{r}$ and where the variables are replaced by their $\phi$-mapped values. Initially, the valuation $\phi$ is chosen non-deterministically; it is not changed by steps.

A history $h = f_1(\overline{r}_1) \cdots f_n(\overline{r}_n)$ is a sequence of events. If there are steps $s \xrightarrow{f_1(\overline{r}_1), \ldots, f_n(\overline{r}_n)} s'$, then we write $s \xrightarrow{r} s'$. 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 $h \in S(s)$, if it is not accepted by $s$. Then, the specification of $O$, denoted by $S(O)$, is the set of histories that are not accepted by any initial state of $O$. Formally, we define:

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

The cross-product $O_1 \times O_2$ denotes an SMR automaton with $S(O_1 \times O_2) = S(O_1) \cap S(O_2)$.

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 $O$ 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 $S(s) \subseteq S(s')$ for SMR automata $O$. To accomplish this efficiently, we compute a simulation relation [Milner 1971] $\leq_O$ among the locations of $O$ which entails the desired inclusion. Technically, $\leq_O$ is the largest relation such that for all locations $l_1 \leq_O l_2$ the following conditions are met: (i) if $l_1$ is not accepting, then $l_2$ is not accepting, and (ii) for all transitions $l_1 \xrightarrow{ev,g} l'_1$ and $l_2 \xrightarrow{ev,g} l'_2$ with $g \land g'$ satisfiable we have $l'_1 \leq_O l'_2$. Relation $\leq_O$ can be computed by a greatest fixed point in the standard way [Baier and Katoen 2008, Section 7.6; Cleaveland and Steen 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 $l \leq_O l'$, then $S((l, \phi)) \subseteq S((l', \phi))$ for all $\phi$. 


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 func₁, . . . , funcₙ be the API functions offered by the SMR algorithm. The event types are (i) in:func₁, . . . , in:funcₙ, (ii) re:func₁, . . . , re:funcₙ, 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:funcᵢ, (ii) the executing thread for type re:funcᵢ, 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 Oᵦ_base × Oᵦ_rr from Figure 5.4. It consists of two SMR automata. First, Oᵦ_rr 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 enter_q for the first time after the retire. Second, Oᵦ_base specifies that no address must be freed that has not been retired yet. Further, Oᵦ_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 Oᵦ_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 ⥜ Adr be the addresses that have been retired in τ but not freed since.

**Definition 5.5 (Double Retire).** Computation τ.(t, in:retire(p), up) performs a double retire of address a if a ∈ retired and a = mₜ(p).

The specification of HP is a bit more involved. For two hazard pointers per thread, a first attempt is the automaton Oᵦ_base × Oᵦ_HP⁰ × Oᵦ_HP¹. Here, Oᵦ_HPᵏ 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
Figure 5.4: SMR automata specifications for EBR resp. HP:

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

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

(c) SMR automaton \( O_{HP} \) 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.
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 \( O^0_{HP} \times O^1_{HP} \) to account for transferring protections among hazard pointers. The
resulting SMR automaton is \( O^{0,1}_{HP} \). Due to the size of the automaton (it consists of 16 states) we do
not present it here, it can be found in Appendix A.2. It is worth noting that \( O^0_{HP} \times O^1_{HP} \) is useful
nevertheless: it is smaller than \( O^{0,1}_{HP} \) 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, we can use \( O_{Base} \)
as a specification for FL.

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

**Assumption 5.6.** SMR automata \( O \) are of the form \( O = O_{Base} \times O_{SMR} \) for some \( O_{SMR} \).

With an SMR specification in form of an SMR automaton \( O \) 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 \( \tau \) of \( R \) into its induced history \( \mathcal{H}(\tau) \) and check inclusion in \( S(O) \). The induced
history \( \mathcal{H}(\tau) \) is a projection of \( \tau \) 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 \( \tau \), denoted
by \( \mathcal{H}(\tau) \), is:

\[
\mathcal{H}(\epsilon) = \epsilon \\
\mathcal{H}(\tau.(t, \text{free}(p), \text{up})) = \mathcal{H}(\tau)\text{.free}(m_\epsilon(p)) \\
\mathcal{H}(\tau.(t, \text{in:func}(\bar{r}), \text{up})) = \mathcal{H}(\tau)\text{.in:func}(t, m_\epsilon(\bar{r})) \\
\mathcal{H}(\tau.(t, \text{re:func}, \text{up})) = \mathcal{H}(\tau)\text{.re:func}(t) \\
\mathcal{H}(\tau.\text{act}) = \mathcal{H}(\tau) \\
\text{otherwise.}
\]
Then, \( \tau \) satisfies \( \mathcal{O} \) if \( \mathcal{H}(\tau) \in S(\mathcal{O}) \). 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 \( R \vDash \mathcal{O} \), if for all \( \tau \in [\text{MGC}(R)]^{Adr} \) we have \( \mathcal{H}(\tau) \in S(\mathcal{O}) \).

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 \( \text{in} \) and \( \text{re} \) 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 \( \mathcal{O}[P]^X \) and corresponds to the standard semantics \( [P(\epsilon)]^X \) as defined in Section 3.3 except for an updated rule for frees from the environment.

**Free**

\[
\text{If} \quad \tau \in [P(\epsilon)]^X \quad \text{and} \quad a \in \text{Adr} \quad \text{can be freed, i.e.,} \quad a \in X \quad \text{and} \quad \mathcal{H}(\tau).\text{free}(a) \in S(\mathcal{O}),
\]

\[\text{then we have} \quad \tau.\text{act} \in \mathcal{O}[P]^X \quad \text{with} \quad \text{act} = (t, \text{free}(a), \emptyset).\]

The new semantics considers \( \mathcal{O} \) only when freeing memory, all other rules remain unaffected. With this definition, \( \mathcal{O}[P]^X \) performs more frees than \( [P(R)]^X \), provided \( R \vDash \mathcal{O} \). 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 \( R \)
Figure 5.9: Semantics of programs relative to an SMR automaton, $O[P]^Y_{X}$.

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

(Skip) \hspace{1em} If $act = \langle t, \text{skip}, \emptyset \rangle$.

(Assign1) \hspace{1em} If $act = \langle t, p.\text{next} := q, [a.\text{next} \mapsto b] \rangle$ then $m_c(p) = a$ and $m_c(q) = b$.

(Assign2) \hspace{1em} If $act = \langle t, p := q, [p \mapsto m_c(q)] \rangle$.

(Assign3) \hspace{1em} If $act = \langle t, p := q.\text{next}, [p \mapsto m_c(a.\text{next})] \rangle$ with $m_c(q) = a \in \text{Adr}$.

(Assign4) \hspace{1em} If $act = \langle t, u := \text{op}(u_1^1, \ldots, u_n^1), [u \mapsto d] \rangle$ with $d = \text{op}(m_c(u_1^1), \ldots, m_c(u_n^1))$.

(Assign5) \hspace{1em} If $act = \langle t, p.\text{data} := u, [a.\text{data} \mapsto m_c(u)] \rangle$ with $m_c(p) = a \in \text{Adr}$.

(Assign6) \hspace{1em} If $act = \langle t, u := q.\text{data}, [u \mapsto m_c(a.\text{data})] \rangle$ with $m_c(q) = a \in \text{Adr}$.

(Assume) \hspace{1em} If $act = \langle t, \text{assume exp} \# \exp', \emptyset \rangle$ then $m_c(\text{exp}) \# m_c(\text{exp}')$.

(Malloc) \hspace{1em} 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, $a \in \text{fresh}_c$ or $a \in \text{freed}_c \cap Y$.

(Free) \hspace{1em} If $act = \langle t, \text{free}(a), \emptyset \rangle$ then $a \in \text{Adr} \cap X$ and $H(\tau).\text{free}(a) \in S(O)$.

(Call) \hspace{1em} If $act = \langle t, \text{in:func}(\bar{\tau}), \emptyset \rangle$, then $m_c(r) \in \text{Adr} \cup \text{Dom}$ for every $r$ in $\bar{\tau}$.

(Return) \hspace{1em} If $act = \langle t, \text{re:func}, \emptyset \rangle$.

(Atomic) \hspace{1em} If $act = \langle \bot, \text{beginAtomic}, \emptyset \rangle$ or $act = \langle t, \text{endAtomic}, \emptyset \rangle$.

(Env) \hspace{1em} If $act = \langle \bot, \text{env}(a), [a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d] \rangle$ then $a \in \text{fresh}_c \cup \text{freed}_c$.

(b) SMR semantics of programs. We define a SOS transition relation $\rightarrow$ relative to a control-flow relation $\rightarrow^\text{cf}$ such that $\tau \in O[P]^Y_{X}$ if $\text{ctrl}(\tau) \neq \emptyset$ with $\text{ctrl}(\tau) = \{ pc | (\lambda t. P[t^r], e) \rightarrow^* (pc, \tau) \}$.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(sos-com)</td>
<td>$com \rightarrow \text{skip}$</td>
</tr>
<tr>
<td>(sos-SEQ1)</td>
<td>$\text{skip; st} \rightarrow \text{st}$</td>
</tr>
<tr>
<td>(sos-SEQ2)</td>
<td>$st_1 \overset{\text{com}}{\rightarrow} st'_1 \rightarrow \text{st}$</td>
</tr>
<tr>
<td>(sos-CHOICE)</td>
<td>$i \in { 1, 2 }$</td>
</tr>
<tr>
<td>(sos-LOOP1)</td>
<td>$st \rightarrow \text{skip}$</td>
</tr>
<tr>
<td>(sos-LOOP2)</td>
<td>$st \rightarrow \text{skip}$</td>
</tr>
<tr>
<td>(sos-ATOMIC)</td>
<td>$\text{endAtomic} \rightarrow \text{skip}$</td>
</tr>
<tr>
<td>(sos-ATOMIC2)</td>
<td>$st \overset{\text{com}}{\rightarrow} st'$</td>
</tr>
<tr>
<td>(sos-ENV)</td>
<td>$act \in \overline{Ac}(\tau, \bot, \text{env}(a))$</td>
</tr>
<tr>
<td>(sos-FREE)</td>
<td>$act \in \overline{Ac}(\tau, \bot, \text{free}(a))$</td>
</tr>
<tr>
<td>(sos-PAR)</td>
<td>$st \overset{\text{com}}{\rightarrow} st'$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(sos-environ)</td>
<td>$act \in \overline{Ac}(\tau, \bot, \text{env}(a))$</td>
</tr>
<tr>
<td>(sos-free)</td>
<td>$act \in \overline{Ac}(\tau, \bot, \text{free}(a))$</td>
</tr>
<tr>
<td>(sos-par)</td>
<td>$st \overset{\text{com}}{\rightarrow} st'$</td>
</tr>
</tbody>
</table>

Section 5.3 Verification Relative to SMR Automata
wrt. \( O \) and the correctness of \( P(e) \) under \( O \) 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].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 \( r \), we write \( \text{good}(r) \); 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 \( R \vDash O \) and \( \text{good}(O[] P[])^{Adr} \), then \( \text{good}(P(R)[])^{Adr} \).

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 \( O[] P[]^{Adr} \) 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 \( \text{good}(O[] P[])^{Adr} \), we will also establish the absence of double retires, as required for a reasonable application of \( O_{Base} \). As expected, compositionality allows us to rely on the simpler SMR semantics.

**Theorem 5.11.** If \( R \vDash O \), then \( P(R)[])^{Adr} \) is free from double retires if \( O[] P[])^{Adr} \) is.

---

1 Bouajjani et al. [2015b] show that linearizability reduces to control location reachability as well.
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 \texttt{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 \texttt{CAS} from Line 315 in order to insert the new node after \texttt{Tail}. Thread \( t_2 \) is executing \texttt{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 \texttt{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 \texttt{Tail}, breaking the shape invariant of the queue. Subsequent \texttt{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. \( \blacksquare \)

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.

\(^1\)The attentive reader of Chapter 2 might observe that, unlike presented in Figure 6.2 here, the next field \( c.\text{next} \) of the removed but not yet retired node \( c \) is never \texttt{NULL} in Michael&Scott’s queue. To obtain such a view where \( c.\text{next} \) is \texttt{NULL}, we require a preceding interference step which suffers from the same imprecision as the interference step presented here. For simplicity, we stick with \( c.\text{next} \) being \texttt{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 315. View $v_2$ captures thread $t_2$ which has removed node $c$ from the queue and is about to retire it, Line 339.

(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 339. The result is $v_4'$ where the retirement of $b$ has immediately freed it (marked with †). Next, we project away $t_2$ and let $t_1$ execute Line 315. 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 $v_4$. 

Section 6.1 Reclamation breaks Ownership
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 guarantee 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 neither reused nor 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.

**Example 6.3.** To see why traditional ownership reasoning breaks when memory is reclaimed and reused, consider the configurations of Micheal&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 $cfg^\prime$ 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 $cfg^\prime$. 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.
(a) Program configurations without view abstraction. In \( \text{cfg} \), 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 \( \text{cfg}' \).

(b) View abstraction for \( \text{cfg} \) gives views \( \nu_1 \) and \( \nu_2 \). Applying traditional ownership reasoning prevents a combined view where nodes \( b \) and \( c \) coincide, because \( b \) is owned. Hence, configuration \( \text{cfg}' \) is not guaranteed to be covered by any view. The analysis is unsound.
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.

Initially, all pointer variables are valid. 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.

The following definition makes this precise.

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

\[
\begin{align*}
\text{valid}_{\tau} &= PVar \\
\text{valid}_{\tau}.(t.p := q, up) &= \text{valid}_{\tau} \cup \{ p \} \quad \text{if } q \in \text{valid}_{\tau} \\
\text{valid}_{\tau}.(t.p := q, up) &= \text{valid}_{\tau} \setminus \{ p \} \quad \text{if } q \notin \text{valid}_{\tau} \\
\text{valid}_{\tau}.(t.p.next := q, up) &= \text{valid}_{\tau} \cup \{ a.next \} \quad \text{if } m_{\tau}(p) = a \in \text{Adr} \land q \in \text{valid}_{\tau} \\
\text{valid}_{\tau}.(t.p.next := q, up) &= \text{valid}_{\tau} \setminus \{ a.next \} \quad \text{if } m_{\tau}(p) = a \in \text{Adr} \land q \notin \text{valid}_{\tau} \\
\text{valid}_{\tau}.(t.p := q.next, up) &= \text{valid}_{\tau} \cup \{ p \} \quad \text{if } q \in \text{valid}_{\tau} \land m_{\tau}(q).next \in \text{valid}_{\tau} \\
\text{valid}_{\tau}.(t.p := q.next, up) &= \text{valid}_{\tau} \setminus \{ p \} \quad \text{if } q \notin \text{valid}_{\tau} \lor m_{\tau}(q).next \notin \text{valid}_{\tau} \\
\text{valid}_{\tau}.(t.free(a), up) &= \text{valid}_{\tau} \setminus \text{invalid}_a \\
\text{valid}_{\tau}.(t.p := \text{malloc}.up) &= \text{valid}_{\tau} \cup \{ p.a.next \} \quad \text{if } \{ p \mapsto a \} \in \text{up} \\
\text{valid}_{\tau}.(t.assume \ p = q, up) &= \text{valid}_{\tau} \cup \{ p, q \} \quad \text{if } \{ p, q \} \cap \text{valid}_{\tau} \neq \emptyset \\
\text{valid}_{\tau}.\text{act} &= \text{valid}_{\tau} \quad \text{otherwise} \\
\end{align*}
\]

with \( \text{invalid}_a = \{ p \mid m_{\tau}(p) = a \} \cup \{ b.next \mid m_{\tau}(b.next) = a \} \cup \{ a.next \} \).
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 \( \tau \), \( \text{owned}_\tau(t) \subseteq \text{Adr} \), are:

\[
\text{owned}_\tau(t) := \emptyset
\]

\[
\text{owned}_\tau(t', p := q[p\rightarrow a])(t) := \text{owned}_\tau(t) \setminus \{a\} \quad \text{if } p \in \text{shared} \land q \in \text{valid}_\tau
\]

\[
\text{owned}_\tau(t', p := q.next[p\rightarrow a])(t) := \text{owned}_\tau(t) \setminus \{a\} \quad \text{if } t \neq t' \land q.m_r(q).next \in \text{valid}_\tau
\]

\[
\text{owned}_\tau(t', p := q.next[p\rightarrow a])(t) := \text{owned}_\tau(t) \setminus \{a\} \quad \text{if } t = t' \land p \in \text{shared}
\]

\[
\text{owned}_\tau(t', p := \text{malloc}[p\rightarrow a])(t) := \text{owned}_\tau(t) \cup \{a\} \quad \text{if } t = t' \land p \notin \text{shared}
\]

\[
\text{owned}_{\tau, \text{free}(a), \varnothing}(t) := \text{owned}_\tau(t) \setminus \{a\}
\]

\[
\text{owned}_{\tau, \text{adr}}(t) := \text{owned}_\tau(t)
\]

otherwise.

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 \( \vartheta_4 \) and thus avoids the associated false alarm.

**Theorem 6.7 (Ownership Guarantee).** Consider \( \tau \in \mathcal{O}[P]^{\text{Adr}} \) with \( m_r(p) \in \text{owned}_\tau(t) \). 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 $O_{\text{Base}} \times O_{\text{EBR}}$, 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 $O_{\text{Base}} \times O_{\text{EBR}}$ 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 $O_{\text{Base}}$ is sound. We integrate an appropriate check: verification fails upon $\text{in}\text{-}\text{retire}(p)$ if $p$ points to $a$ and $a$ is currently retired, that is, if $O_{\text{Base}}$ is in state $(L_3, \varphi)$ 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 tmrex$^2$ and empirically evaluated it on Treiber’s stack, Michael&Scott’s queue, and the DGLM queue. For a baseline, 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 $O_{\text{Base}}$ 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 ✗ 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

---

$^2$ tmrex is freely available at: ☞ https://wolff09.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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Ownership</th>
<th>States</th>
<th>Interferences (pruned)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single lock stack</strong></td>
<td>GC</td>
<td>328</td>
<td>3.2k (10k)</td>
<td>0.006s</td>
</tr>
<tr>
<td></td>
<td>New</td>
<td>703</td>
<td>7k (22k)</td>
<td>0.21s</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>16k</td>
<td>183k (0)</td>
<td>5.34s</td>
</tr>
<tr>
<td><strong>Single lock queue</strong></td>
<td>GC</td>
<td>100</td>
<td>0.7k (5k)</td>
<td>0.04s</td>
</tr>
<tr>
<td></td>
<td>New</td>
<td>520</td>
<td>0.7k (31k)</td>
<td>0.56s</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>27k</td>
<td>442k (0)</td>
<td>32s</td>
</tr>
<tr>
<td><strong>Treiber’s stack</strong></td>
<td>GC</td>
<td>269</td>
<td>3.5k (16k)</td>
<td>0.06s</td>
</tr>
<tr>
<td></td>
<td>New</td>
<td>744</td>
<td>44k (96k)</td>
<td>2.36s</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>117k</td>
<td>7920k (0)</td>
<td>602s</td>
</tr>
<tr>
<td><strong>Michael&amp;Scott’s queue</strong></td>
<td>GC</td>
<td>3134</td>
<td>47k (1237k)</td>
<td>2.52s</td>
</tr>
<tr>
<td></td>
<td>New</td>
<td>19553</td>
<td>6678k (20748k)</td>
<td>3h</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>≥ 69000</td>
<td>−</td>
<td>2s ❌</td>
</tr>
<tr>
<td><strong>DGLM queue</strong></td>
<td>GC</td>
<td>3134</td>
<td>47k (1237k)</td>
<td>2.52s</td>
</tr>
<tr>
<td></td>
<td>New</td>
<td>≥ 6500</td>
<td>−</td>
<td>30s ❌</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>≥ 64000</td>
<td>−</td>
<td>2s ❌</td>
</tr>
</tbody>
</table>

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

False positive due to imprecision in the memory abstraction.

... 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.
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 $P(O)$ by considering only those computations where at most a single address is reused. This avoids the need for an exploration of full $O\left[\begin{array}{c} P \\ Adr \end{array}\right]$ 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 $O$ specifying the SMR algorithm. To achieve this, we establish that for every computation from $O\left[\begin{array}{c} P \\ Adr \end{array}\right]$ 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 $P(O)$ 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 $\tau$ where memory is reused arbitrarily with a computation $\sigma$ where memory reuse is restricted. As noted before, we want the threads in $\tau$ and $\sigma$ to reach the same control locations in order to verify safety properties of $\tau$ in $\sigma$. We introduce a similarity relation among computations such that $\tau$ and $\sigma$ 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 \( \sigma \) to mimic subsequent actions of \( \tau \), that is, to preserve similarity for subsequent actions. This is because most actions involve memory interaction. Since \( \sigma \) reuses memory differently than \( \tau \), 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 \( \tau_1 \) be a computation of a data structure \( P(O_{\text{Base}} \times O_{\text{HP}}^0 \times O_{\text{HP}}^1) \) using HP:

\[
\tau_1 = \langle t, p := \text{malloc}[p \mapsto a, \ldots] \rangle \cdot \langle t, \text{in:retire}(p), \emptyset \rangle \cdot \langle t, \text{free}(a), \emptyset \rangle \cdot \langle t, \text{re:retire}, \emptyset \rangle .
\]

\[
\langle t, q := \text{malloc}, [q \mapsto a, \ldots] \rangle
\]

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 \( \sigma_1 \) is a computation where \( a \) shall not be reused, then \( \sigma_1 \) is not able to execute the exact same sequence of actions as \( \tau_1 \). However, it can mimic \( \tau_1 \) as follows:

\[
\sigma_1 = \langle t, p := \text{malloc}, [p \mapsto b, \ldots] \rangle \cdot \langle t, \text{in:retire}(p), \emptyset \rangle \cdot \langle t, \text{free}(b), \emptyset \rangle \cdot \langle t, \text{re:retire}, \emptyset \rangle .
\]

\[
\langle t, q := \text{malloc}, [q \mapsto a, \ldots] \rangle
\]

where \( \sigma_1 \) coincides with \( \tau_1 \) up to replacing the first allocation of \( a \) with another address \( b \). We say that \( \sigma_1 \) elides the reuse of \( a \). The memories of \( \tau_1 \) and \( \sigma_1 \) 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 \( \tau_1 \) and \( \sigma_1 \) from the previous example, the last allocation renders valid pointer \( q \). On the other hand, the free to \( a \) in \( \tau_1 \) renders \( p \) invalid. The reallocation of \( a \) does not change the validity of \( p \), it remains invalid. In \( \sigma_1 \), address \( b \) is allocated and freed rendering \( p \) invalid. It remains invalid after the subsequent allocation of \( a \). That is, both \( \tau_1 \) and \( \sigma_1 \) agree on the validity of \( q \) and the invalidity of \( p \). Moreover, \( \tau_1 \) and \( \sigma_1 \) 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 \( m_e \mid \text{valid} \).

**Definition 7.3 (Restrictions).** A restriction of memory \( m \) to a set \( P \subseteq P\text{Exp} \), written \( m \mid_P \), is a new memory \( m' \) with domain \( \text{dom}(m') := P \cup \{ \text{DVar} \in D\text{Exp} \mid \text{a} \in m(P) \} \) 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 \( \tau \) and \( \sigma \) are similar, denoted by \( \tau \sim \sigma \), if we have \( \text{ctrl}(\tau) = \text{ctrl}(\sigma) \) and \( m_\tau \mid \text{valid} = m_\sigma \mid \text{valid} \).

If two computations \( \tau \) and \( \sigma \) are similar, then each action enabled after \( \tau \) can be mimicked in \( \sigma \).

More precisely, action \( \text{act} = \langle t, \text{com}, \text{up} \rangle \) after \( \tau \) can be mimicked by \( \text{act}' = \langle t, \text{com}, \text{up}' \rangle \) after \( \sigma \). 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 \( \sigma \) if \( \text{com} \) involves invalid pointers.

**Example 7.5 (Continued).** Consider the following continuation of \( \tau_1 \) and \( \sigma_1 \):

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

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

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 \( r_3 = r_2 \cdot y \) and \( \sigma_3 = \sigma_2 \cdot y \) with
\[
y = \langle t, \text{in:protect}_k(p), \varnothing \rangle, \langle t, \text{re:protect}_k, \varnothing \rangle, \langle t, \text{in:retire}(q), \varnothing \rangle, \langle t, \text{re:retire}, \varnothing \rangle
\]
where thread \( t \) issues a protection and a retirement using \( p \) and \( q \), respectively. The histories induced by those computations are:
\[
\mathcal{H}(r_3) = \mathcal{H}(r_2).\text{in:protect}_k(t,a).\text{re:protect}_k(t).\text{in:retire}(t,a).\text{re:retire}(t)
\]
and
\[
\mathcal{H}(\sigma_3) = \mathcal{H}(\sigma_2).\text{in:protect}_k(t,b).\text{re:protect}_k(t).\text{in:retire}(t,a).\text{re:retire}(t).
\]
Recall that \( r_2 \) and \( \sigma_2 \) are similar. Similarity guarantees that the events of the \( \text{retire} \) call coincide since \( q \) is valid. The events of the \( \text{protect} \) 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 \( r_3 \) and \( \sigma_3 \). More precisely, automaton \( O_{HP} \) from Figure 5.4 takes the following steps from the initial state \( (L_b, \varnothing) \) with \( \varnothing = \{ z_1 \mapsto t, z_a \mapsto a \} \):
\[
(L_b, \varnothing) \xrightarrow{\mathcal{H}(r_2)} (L_b, \varnothing) \xrightarrow{\text{in:protect}_k(t,a)} (L_b, \varnothing) \xrightarrow{\text{re:protect}_k(t)} (L_{10}, \varnothing)
\]
and
\[
(L_b, \varnothing) \xrightarrow{\mathcal{H}(\sigma_2)} (L_b, \varnothing) \xrightarrow{\text{in:protect}_k(t,b)} (L_b, \varnothing) \xrightarrow{\text{re:protect}_k(t)} (L_{10}, \varnothing).
\]
This prevents \( a \) from being freed after \( r_3 \), because a \( \text{free}(a) \) would lead to the final state \( (L_{12}, \varnothing) \) and is thus not enabled, but allows for freeing it after \( \sigma_3 \).

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 \( \sigma \) allows for more behavior on the valid addresses than the mimicked \( r \). 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 \( r_3 \) and \( \sigma_3 \). After \( \sigma_3 \) strictly more behavior is possible: \( \sigma_3 \) can free \( a \) while \( r_3 \) 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 \( \text{frees} \). Intuitively, the behavior allowed by \( O \) 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 \( O \) on address \( a \) after history \( h \) is the set \( \mathcal{F}_O(h, a) := \{ h' \mid h.\hat{h}' \in S(\hat{O}) \land \text{frees}_h \subseteq a \} \).
Note that $h^l \in \mathcal{F}_O(h, a)$ 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 $\sigma$ may have already freed such an address while $\tau$ has not. Hence, the free is no longer allowed after $\sigma$ but still possible after $\tau$. To prevent such invalid addresses from breaking the desired inclusion on valid addresses, we strip from $\mathcal{F}_O(h, a)$ 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 $\mathcal{F}_O(h, a)$. To be more precise, we have $\mathcal{F}_O(\mathcal{H}(\tau), a) = \mathcal{F}_O(\mathcal{H}(\tau, \text{act}), a)$ for all actions $\text{act}$ which do not emit an event.

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

**Definition 7.8 (SMR Behavior Inclusion).** Computation $\sigma$ includes the SMR behavior of $\tau$, denoted by $\tau \preceq \sigma$, if $\mathcal{F}_O(\tau, a) \subseteq \mathcal{F}_O(\sigma, a)$ 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 $\tau_3$ and $\sigma_3$:

$$\tau_4 = \tau_3 . \langle t, q.\text{next} := q, [a.\text{next} \mapsto a] \rangle . \langle t, p.\text{next} := p, [a.\text{next} \mapsto a] \rangle$$

and

$$\sigma_4 = \sigma_3 . \langle t, q.\text{next} := q, [a.\text{next} \mapsto a] \rangle . \langle t, p.\text{next} := p, [b.\text{next} \mapsto b] \rangle .$$

The first appended action updates $a.\text{next}$ in both computations to $a$. Since $q$ is valid after both $\tau_3$ and $\sigma_3$, this assignment renders valid $a.\text{next}$. The second action updates $a.\text{next}$ in $\tau_4$. This results in $a.\text{next}$ being invalid after $\tau_4$ because the right-hand side of the assignment is the invalid $p$. In $\sigma_4$ the second action updates $b.\text{next}$ which is why $a.\text{next}$ remains valid. That is, the valid memories of $\tau_4$ and $\sigma_4$ differ. We have executed an action that cannot be mimicked on the valid memory despite the computations being similar.

---

Section 7.2 Preserving Similarity
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 rational 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 $\tau.(t, \text{com}, \text{up})$ performs an unsafe access if $\text{com}$ contains $p.\text{data}$ or $p.\text{next}$ with $p \notin \text{valid}$. 

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 $\tau_1, \sigma_1$ from Example 7.1:

$$
\tau_5 = \tau_1.(t, \text{in;}\text{retire}(p), \varnothing) \quad \text{with} \quad \mathcal{H}(\tau_5) = \mathcal{H}(\tau_1).\text{in;}\text{retire}(t, a)
$$

and

$$
\sigma_5 = \sigma_1.(t, \text{in;}\text{retire}(p), \varnothing) \quad \text{with} \quad \mathcal{H}(\sigma_5) = \mathcal{H}(\sigma_1).\text{in;}\text{retire}(t, b)
$$

The SMR behavior of $\tau_1$ is included in $\sigma_1$, that is, $\tau_1 \preceq \sigma_1$. After $\tau_5$ a deletion of $a$ is possible because it is retired. After $\sigma_5$ a deletion of $a$ is prevented by $O_{\text{base}}$ because $a$ has not been retired. Formally, we have $\text{free}(a) \in \mathcal{F}_O(\tau_5, a)$ and $\text{free}(a) \notin \mathcal{F}_O(\sigma_5, a)$. However, $a$ is a valid address because it is referenced by the valid pointer $q$. That is, the behavior inclusion among $\tau_1$ and $\sigma_1$ 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 \textit{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.

\footnotesize
\begin{definition}[Racy Call] Computation \( r.(t, \text{in:func}(\tau), \emptyset) \) performs a racy call if the following holds for \( h = \mathcal{H}(\tau) \) and \( \mathcal{\overline{s}} = m_r(\mathcal{\overline{t}}) \):

\[
\exists a \exists w. \ (\forall i. (v_i = a \lor r_i \in \text{valid} \lor r_i \in \text{DExp}) \implies v_i = w_i) \\
\land \mathcal{F}_O(h.\text{in:func}(t, w), a) \notin \mathcal{F}_O(h.\text{in:func}(t, \mathcal{\overline{s}}), a)
\]
\end{definition}

It follows immediately that calls containing only valid pointers are not racy. Using the SMR automaton for HP, \( O_{\text{Base}} \times O_{\text{HP}}^0 \times O_{\text{HP}}^1 \), we deem racy the call to \texttt{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, \texttt{retire} is always called using valid pointers in practice. For \texttt{protect} calls one can show that they never race. We have already seen this in Example 7.6. There, a call to \texttt{protect} with invalid pointers did not break the SMR behavior inclusion. Instead, the mimicking computation \( \sigma_3 \) could perform strictly more frees than the computation it mimicked \( \tau_3 \).

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

\footnotesize
\begin{definition}[Pointer Race] A computation \( \tau.\text{act} \) is a pointer race if \( \text{act} \) performs (i) an unsafe access, or (ii) a racy SMR call.
\end{definition}

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.

\footnotesize\(^1\) For an instantiation of this pattern, consider Lines 326 to 328 of Michael&Scott’s queue from Figure 2.13.
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 \( O = O_{\text{Base}} \times O_{\text{SMR}} \) supports elision of memory reuse if for all \( h, h' \in S(O_{\text{Base}}) \) and \( a, b, c \in \text{Adr} \) the following conditions are met:

(i) \( a \neq c \neq b \) implies \( F_{O_{\text{SMR}}}(h, c) = F_{O_{\text{SMR}}}(h[a/b], c) \),
(ii) \( F_O(h, a) \subseteq F_O(h', a) \) and \( b \in \text{fresh}_h \) implies \( F_{O_{\text{SMR}}}(h, b) \subseteq F_{O_{\text{SMR}}}(h', b) \), and
(iii) \( a \neq b \) and \( h.\text{free}(a) \in S(O) \) implies \( F_{O_{\text{SMR}}}(h, b) = F_{O_{\text{SMR}}}(h.\text{free}(a), b) \).

Crucially, the above definition is concerned with the SMR-specific part \( O_{\text{SMR}} \) only. Automaton \( O_{\text{Base}} \) does not satisfy Property (ii) for arbitrary histories \( h, h' \). The problem is that an \( \text{in:retire}(t, b) \) in history \( h \) takes \( O_{\text{Base}} \) 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 \( O_{\text{Base}} \) is in \( L_2 \) where \( \text{free}(b) \) is not allowed. So, \( h' \) does not include all of \( h \)'s behavior. When constructing for \( \tau \) a mimicking computation \( \sigma \) that elides the reuse of \( b \), the problematic scenario occurs only if \( b \) is both freed and retired after \( \tau \). 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 \( O_{\text{Base}} \) would be in \( L_2 \) rather than \( L_3 \) after \( h = H(\tau) \).

For \( b \) to be freed, there must be a preceding retirement according to \( O_{\text{Base}} \). 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 \( O \) 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 $O_{\text{Base}} \times O_{\text{EBR}}$ and $O_{\text{Base}} \times O_{1}^{0} \times O_{1}^{1}$ from Figure 5.4 as well as $O_{\text{Base}} \times O_{1}^{0}$ 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 $O[P]_{\text{Adr}}$.

The semantics that we suggest to analyze is $O[P]_{\text{Adr}}^{\text{one}} := \bigcup_{a \in \text{Adr}} O[P]_{\text{Adr}}^{(a)}$. 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 $\text{assume } p = q$ 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:

```c
326  Node* head = Head;
327  protect_0(head);
328  if (head != Head) continue;
```

In Line 326 the value of the shared pointer `Head` is read into the local pointer `head`. Then, a hazard pointer is used in Line 327 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 328 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 327 is indeed protected after Line 328. With respect to the SMR automaton $O_{HP}$, the assumption guarantees that the protection was issued before a retirement (after the latest reallocation) so that $O_{HP}$ 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 $\tau \in O\left[ P \right]_{Adr}^{Adr}$ and $\sigma \in O\left[ P \right]_{Adr}^{(a)}$ be similar computations. Intuitively, $\sigma$ is a computation which elides the reuses from $\tau$ except for address $a$. Address $a$ can be used in $\sigma$ in exactly the same way as it is used in $\tau$. Let $act$ be an ABA prone assumption of the form $act = (t, assume p = q, \emptyset)$. Assume $act$ is enabled after $\tau$. To detect this ABA under $O\left[ P \right]_{Adr}^{(a)}$ we need $act$ to be enabled after $\sigma$. We seek to have $\sigma. act \in O\left[ P \right]_{Adr}^{(a)}$. 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 $\tau$ and $\sigma$. However, if $p$ points to $a$ in $\tau$, then it does so in $\sigma$ because $a$ is (re)used in $\sigma$ in the same way as in $\tau$. 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 $\tau$ and $\sigma$ are $a$-aligned, $\tau \preceq_a \sigma$, if:

- $\forall p \in \text{PVar}.\ m_\tau(p) = a \iff m_\sigma(p) = a$
- $\forall b \in m_\tau(\text{valid}_\tau).\ m_\tau(b.\text{next}) = a \iff m_\sigma(b.\text{next}) = a$
- $a \in \text{fresh}_\tau \cup \text{freed}_\tau \iff a \in \text{fresh}_\sigma \cup \text{freed}_\sigma$
- $F_{\sigma}(\tau, a) \subseteq F_{\sigma}(\sigma, a)$
- $a \in \text{retired}_\tau \iff a \in \text{retired}_\sigma$.

The first line in this definition states that the same pointer variables in $\tau$ and $\sigma$ 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 $\tau$ and $\sigma$ may differ on the in-use addresses due to eliding reuse. The third line states that $a$ can be allocated in $\tau$ iff it can be allocated in $\sigma$. The fourth line states that the SMR automaton allows for more behavior on $a$ in $\sigma$ than in $\tau$. These properties combined guarantee that $\sigma$ can mimic actions of $\tau$ involving $a$ no matter if invalid pointers are used. The last line requires that $a$ is retired in $\tau$ iff it is retired in $\sigma$. This property makes double retires performed after $\tau$ visible in the mimicking $\sigma$. 

74

Chapter 7 Pointer Races
The address alignment lets us detect ABAs in $O[\mathbb{P}]^{\text{one}}_{\text{Adr}}$. 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 $O[\mathbb{P}]^{\text{Adr}}_{\text{Adr}}$, 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 $O[\mathbb{P}]^{\text{none}}_{\text{Adr}}$ is free from harmful ABAs if:

$$\forall \sigma_a.\text{act} \in O[\mathbb{P}]^{\{a\}}_{\text{Adr}} \forall \sigma_b \in O[\mathbb{P}]^{\{b\}}_{\text{Adr}} \exists \sigma'_b \in O[\mathbb{P}]^{\{b\}}_{\text{Adr}}. \sigma_a \sim \sigma_b \land \text{act} = \langle \bullet, \text{assume } \bullet, \bullet \rangle \implies \sigma_a.\text{act} \sim \sigma'_b \land \sigma_b \not\sim \sigma'_b \land \sigma_a.\text{act} < \sigma'_b.$$

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

**Example 7.19 (Continued).** Consider the computation $r.\text{act}$ of Michael&Scott’s queue with:

$$r = r_a.\langle t, \text{head} := \text{Head, } [\text{head } \mapsto a] \rangle. r_7.\text{free}(a). r_b.$$  
$$\langle t, \text{in:protect}_0(\text{head), } \emptyset \rangle. \langle t, \text{re:protect}_{t_0}, \emptyset \rangle$$

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

This computation resembles a thread $t$ executing Lines 326 to 328 while an interferer frees address $a$ referenced by head, reallocates it, and makes it the head of the queue again; we assume that $r_a, r_7, r_b$ consist of the interferer’s actions the precise form of which does not matter here. The $\text{assume}$ in $\text{act}$ resembles the conditional from Line 328 and states that the condition evaluates to $\text{true}$. That is, $\text{act}$ is a potential ABA on address $a$.

Reusing address $a$ allows us to mimic $r$ with an $a$-aligned computation $\sigma_a \in O[\mathbb{P}]^{\{a\}}_{\text{Adr}}$. The ABA prone action $\text{act}$ is guaranteed to be enabled after $\sigma_a$, so $\sigma_a.\text{act}$ mimics $r.\text{act}$. Reusing another
address \(b\) yields a \(b\)-aligned \(\sigma_b \in \mathcal{O}[\mathcal{P}_b^{\{b\}}_{\text{Adr}}]\) mimicking \(\tau\). After \(\sigma_b\), \(act\) may not be enabled. The reason for this is that \(\sigma_b\) elides allocations of \(a\) to avoid it being reused. The interferer’s reallocation of \(a\) (in \(\tau_8\)) forces \(\sigma_b\) to elide its previous allocation. Hence, thread \(t\)’s \(\text{head}\) does not point to \(a\) while \(\text{head}\) still does. The ABA prone \(act\) is not enabled after \(\sigma_b\).

To see that the above ABA is harmless, consider the following rescheduling of the actions in \(\tau\):

\[
\tau' = \tau_6.\tau_7.\text{free}(a).\tau_8, \langle t, \text{head} := \text{head}, [\text{head} \mapsto a] \rangle.
\]

\[
\langle t, \text{in:protect}_0(\text{head}), \emptyset \rangle, \langle t, \text{re:protect}_b, \emptyset \rangle.
\]

Here, thread \(t\) reads the latest version of \(\text{head}\). This gives rise to a computation \(\sigma'_b \in \mathcal{O}[\mathcal{P}^{\{b\}}_{\text{Adr}}]\) mimicking \(\tau'\). Unlike \(\sigma_b\), however, \(\sigma'_b\) can execute \(act\) since the later read of \(\text{head}\) is not affected by the elision of reallocations. Finally, \(\sigma'_b.\text{act}\) mimics \(\tau.\text{act}\). Requiring the existence of such a \(\sigma'_b\) 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 \(\mathcal{O}[\mathcal{P}^{\text{one}}_{\text{Adr}}]\). Altogether, this means that we can rely on \(\mathcal{O}[\mathcal{P}^{\text{one}}_{\text{Adr}}]\) 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 \(\mathcal{O}[\mathcal{P}^{\text{one}}_{\text{Adr}}]\) instead of full \(\mathcal{O}[\mathcal{P}^{\text{Adr}}]\).

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

**Proof Sketch.** We construct \(\sigma\) inductively by mimicking every action from \(\tau\) and eliding reuses as needed. For the construction, consider \(\tau.\text{act} \in \mathcal{O}[\mathcal{P}^{\text{Adr}}]\) and assume we have already constructed, for every \(a \in \text{Adr}\), an appropriate \(\sigma_a \in \mathcal{O}[\mathcal{P}^{\{a\}}_{\text{Adr}}]\). Consider some address \(a \in \text{Adr}\). The task is to mimic \(\text{act}\) in \(\sigma_a\). If \(\text{act}\) is an assignment or an SMR call, then pointer race freedom guarantees that we can mimic \(\text{act}\) 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 \(\text{act}\) executes an ABA assumption \(\text{assume } p = q\). That the assumption is an ABA means that at least one of the pointers is invalid, say \(p\). Hence, \(\text{act}\) may not be enabled after \(\sigma_a\). Let \(p\) point to \(b\) in \(\tau\). By induction, we have already constructed \(\sigma_b\) for \(\tau\). The ABA is enabled after \(\sigma_b\). This is due to \(\tau \preceq_b \sigma_b\). It implies that \(p\) points to \(b\) in \(\tau\) iff \(p\) points to \(b\)
in \( \sigma_b \) (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 \( r.\text{act} \) for \( a \). Applying Definition 7.18 to \( \sigma_b.\text{act} \) and \( \sigma_a \) yields some \( \sigma'_a \) that satisfies the required properties.

Second, consider the case of \( \text{act} \) performing a \texttt{free}(\( b \)). If \( \text{act} \) is enabled after \( \sigma_a \) 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 \( r.\text{act} \sim \sigma_a \). By the definition of elision support, Definition 7.14iii, the \texttt{free} does not affect the behavior of the SMR automaton on other addresses. We get \( r.\text{act} \prec \sigma_a \). With the same arguments we conclude \( r.\text{act} \bowtie \sigma_a \). That is, we do not need to mimic frees of invalid addresses.

Last, consider \( \text{act} \) executing an allocation \( p := \text{malloc} \) of address \( b \). If \( b \) is fresh in \( \sigma_a \) or \( a = b \), then \( \text{act} \) is enabled after \( \sigma_a \). 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 \( O \) here because there are no double retires on \( b \) by assumption: a double retire on \( b \) in \( \tau \) would manifest as a double retire in some \( \sigma_b \in O[P]_{Adr}^{one} \) with \( \tau \bowtie_b \sigma_b \) which exists by induction. Consider now the remaining case where \( \text{act} \) is not enabled after \( \sigma_a \) because \( b \) is not fresh. We replace in \( \sigma_a \) 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 \( \prec \in \{ \sim, <, \bowtie_a \} \). Since \( b \) is fresh in \( \sigma_a[b/c] \), we conclude by enabledness of \( \text{act} \) as in the previous case. \( \blacksquare \)

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 \( O \) supports elision and \( O[P]_{Adr}^{one} \) is free from pointer races, double retires, and harmful ABAs, then \( \text{good}(O[P]_{Adr}^{one}) \iff \text{good}(O[P]_{Adr}^{one}) \).

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 \( O_{Base} \) in particular (cf. Section 5.2).

**Theorem 7.22.** If \( O \) supports elision and \( O[P]_{Adr}^{one} \) is free from pointer races, double retires, and harmful ABAs, then \( O[P]_{Adr}^{one} \) 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 \texttt{retire} with invalid pointers. That is, the pointer race check boils down to scanning dereferences and \texttt{retire} invocations for invalid pointers. A more general check for racy calls can be implemented by using the technique from Proposition 5.3.

\begin{proposition}
If a call is racy wrt. $O_{\text{Base}} \times O_{\text{EBR}}$ or $O_{\text{Base}} \times O_{\text{HP}}^0 \times O_{\text{HP}}^1$ or $O_{\text{Base}} \times O_{\text{HP}}^{0,1}$, then it is a call to function \texttt{retire} with an invalid pointer as its argument.
\end{proposition}

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 $\sigma_{\text{act}}$ contains a harmful ABA can be detected in the view $v_a$ for thread $t$ which performs $\text{act}$. Like for computations, the view abstraction $v_b$ of $\sigma_b$ 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 $\sigma_b$ is $\alpha(\sigma_b) = \{ v_b, v \}$ where $v_b$ is the view for thread $t$ which performs $\text{act}$. For just some $v_b'$, it is not guaranteed that there is a computation $\sigma_b'$ 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 $\sigma_b'$.

To overcome this problem, we use a method to search for a $v_b'$ that guarantees the existence of $\sigma_b'$; in terms of the above example, guarantees that there is $\sigma_b'$ with $\alpha(\sigma_b') = \{ v_b', v \}$. We take
the view $v_k$ that cannot perform the ABA. We apply sequential steps to $v_k$ 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 $v^l_k$. 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 $O_{Base} \times O_{EBR}$ and $O_{Base} \times O_{HP}^0 \times O_{HP}^1$, 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 $O_{Base}$ 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.

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

\(^2\) tmrexp is freely available at: https://wolff09.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.

<table>
<thead>
<tr>
<th>SMR</th>
<th>Program</th>
<th>States</th>
<th>ABAs</th>
<th>Linearizability</th>
<th>ABA Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBR</td>
<td>Treiber’s stack</td>
<td>1822</td>
<td>0</td>
<td>16s ✓</td>
<td>0s ✓</td>
</tr>
<tr>
<td></td>
<td>Michael&amp;Scott’s queue</td>
<td>7613</td>
<td>0</td>
<td>2630s ✓</td>
<td>0s ✓</td>
</tr>
<tr>
<td></td>
<td>DGLM queue</td>
<td>27132</td>
<td>0</td>
<td>3754s ✓</td>
<td>b</td>
</tr>
<tr>
<td>HP</td>
<td>Treiber’s stack</td>
<td>2606</td>
<td>186</td>
<td>19s ✓</td>
<td>0.06s ✓</td>
</tr>
<tr>
<td></td>
<td>Opt. Treiber’s stack</td>
<td>—</td>
<td>—</td>
<td>0.8s ✗</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Michael&amp;Scott’s queue</td>
<td>19028</td>
<td>536</td>
<td>7075s ✓</td>
<td>0.9s ✓</td>
</tr>
<tr>
<td></td>
<td>DGLM queue</td>
<td>41753</td>
<td>2824</td>
<td>7010s ✓</td>
<td>b</td>
</tr>
</tbody>
</table>

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

*b* Imprecision in the memory abstraction required hinting.

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:

```c
Node* top = ToS;
node->next = top;
if (CAS(&ToS, top, node))
    break
```

The operation reads the top-of-stack pointer into a local variable `top` in Line 278, links the newly allocated `node` to the top-of-stack in Line 279, and swings the top-of-stack pointer to the new node in Line 280. Between Line 278 and Line 280 the node referenced by `top` can be popped, reclaimed, reused, and reinserted by an interferer. Consequently, the `CAS` in Line 280 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 280 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 280 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 $O$. 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. $O$, then by definition there is $\tau \in O [ MGC(R) ]_{Adv}^{Adv}$ with $H(\tau) \notin S(O)$. Hence, there must be some state $s$ with $H(\tau) \notin S(s)$. Consider the specifications $O_{Base} \times O_{HP}^{0} \times O_{HP}^{1}$ and $O_{Base} \times O_{EBR}$. 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 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

\[^3\] `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.

<table>
<thead>
<tr>
<th>SMR Implementation</th>
<th>Specification</th>
<th>States</th>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hazard Pointers</td>
<td>$O_{\text{Base}} \times O_{\text{HP}}^0 \times O_{\text{HP}}^1$</td>
<td>5437</td>
<td>1.5s ✅</td>
</tr>
<tr>
<td>Hazard Pointers</td>
<td>$O_{\text{Base}} \times O_{\text{HP}}^{0,1}$</td>
<td>5304</td>
<td>1.5s ✅</td>
</tr>
<tr>
<td>Epoch-Based Reclamation</td>
<td>$O_{\text{Base}} \times O_{\text{EBR}}$</td>
<td>11528</td>
<td>11.2s ✅</td>
</tr>
</tbody>
</table>

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.
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 \( \mathcal{O}[P]_{Adr} \). The ABA check, however, mandates an analysis of \( \mathcal{O}[P]_{one} \). 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 \( \mathcal{O}[P]_o \), using off-the-shelf GC verifiers. To avoid an expensive state space exploration (a semantic analysis) of a semantics larger than \( \mathcal{O}[P]_o \), we present a type system a successful type check of which guarantees \( \mathcal{O}[P]_{Adr} \) 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 \( \mathcal{O}[P]_{Adr} \). 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

---

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

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 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

\[
\text{com ::= com } | \text{@inv angel } r \ | \text{@inv } p = q \ | \text{@inv } p \ \text{in } r \\
| \text{@inv active}(p) \ | \text{@inv active}(r).
\]

Angels are local variables \( r \) from the set \( \text{AVar} \). Invariant annotations include allocations of angels via the keyword \( \text{angel } r \). 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 \( O[P]_X \), the new commands do not lead to memory updates:

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

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 \( r \), we construct a formula \( \text{inv}(r) \). The invariant annotations are defined to hold for \( r \) iff \( \text{inv}(r) \) 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 \( \sigma \).\(^1\)

\(^1\)In Section 8.3 we will additionally require \( \text{active} \) 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.
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 \( \text{inv}(\tau) \) allows for. It remains to verify annotations. Section 8.5 shows how to automatically prove the correctness of invariant annotations for all computations.

### 8.2 Avoiding All Reallocations

Our goal is to strengthen the reduction from Chapter 7 such that verification of full \( \mathcal{O}[P]^{Adr} \) becomes possible under \( \mathcal{O}_\Delta^{Adr} \). 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 \( \tau \), 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 \( \mathcal{O}_\Delta^{Adr} \). Hence, we do not check the actually executed commands but those that the control-flow could choose next. Formally, we say that a command \( \text{com} \) is control-flow-enabled after \( \tau \), denoted by \( \text{com} \in \text{next-com}(\tau) \), if there is \( pc \in \text{ctrl}(\tau) \) with \( pc(\tau) \xrightarrow{\text{com}} \).
for some thread $t$. Then, $\tau$ is prone to an unsafe assumption if an ABA prone assumption is control-flow-enabled.

**Definition 8.3 (Unsafe Assumption).** A computation $\tau$ is prone to an unsafe assumption if there is an 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 $act$ 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 $O$ support elision and let $O[[P]]_{Adr}^{O}$ be free from strong pointer races and double retires. Then, for all $\tau \in O[[P]]_{Adr}^{O}$ there is some $\sigma \in O[[P]]_{Adr}^{O}$ 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 $O_{Base}$. The proof of the theorem is analogous to the one of Theorem 7.20.

The next step in our reduction is the removal of free commands. We simply strip them away from the computations of $O[[P]]_{Adr}^{O}$. The result are GC computations from $[[P]]_{O}^{O}$. 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 $\sigma \in O[[P]]_{Adr}^{O}$ is free from strong pointer races, then there is some $\gamma \in [[P]]_{O}^{O}$ such that $ctrl(\sigma) = ctrl(\gamma)$ and $m_{\sigma}|_{valid_{\sigma}} = m_{\gamma}|_{valid_{\gamma}}$ and $inv(\gamma) \implies inv(\sigma)$.

**Proof Sketch.** To see the theorem, we proceed in two steps. First, we remove env commands from $\sigma$. 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 $\sigma'$ that satisfies $\sigma \sim \sigma'$ and $inv(\sigma) \iff inv(\sigma')$. Next, we remove free commands from $\sigma'$ and arrive at $\gamma$. As no memory is reused in $O[[P]]_{Adr}^{O}$, all allocations remain enabled. The remaining commands ignore $O$, 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 $m_{\sigma} \mid_{\text{valid}, \sigma} = m_{\gamma} \mid_{\text{valid}, \gamma}$ where $m_{\gamma}$ is restricted to the valid expressions of $\sigma$ rather than $\gamma$. The latter establishes that invariant violations in $\gamma$ carry over to $\sigma$.

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 $O$ supports elision and $O(P)_{\text{Adr}}$ is free from strong pointer races and double retires, then $\text{good}(O(P)_{\text{Adr}}) \iff \text{good}(P)_{\text{o}}$ and $O(P)_{\text{Adr}}$ is free from double retires.

We turn towards checking $O(P)_{\text{Adr}}$ 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 $O$ relative to which we describe the type system. We assume that $O$ contains exactly two variables $z_t$ and $z_a$. Intuitively, $z_t$ stores the thread for which $O$ 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.

$L$: 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.

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

$S$: 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.

$E_L$: 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 $A$ and $S$. 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 $E$. 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 $A$ to $p$. We then deduce from the SMR automaton that $p$ can be accessed safely because the protection was successful. This adds guarantee $S$. 
### 8.3.2 Types

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

\[
T ::= \emptyset \mid L \mid A \mid S \mid E_L \mid T \wedge T .
\]

The meaning of guarantees \( L \) to \( E_L \) 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 \( T \cap \{ S, L, A \} \neq \emptyset \).

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 \( E_L \). To understand it, note that our type system infers guarantees about the protection of pointers thread-locally from the code, that is, 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 Owicky and Gries [1976]. A set of locations \( L \) in the SMR automaton \( O \) is *closed under interference from other threads*, if no SMR command issued by a thread different from \( z_t \) (the protections of which we track) can leave the locations. Formally, for every transition \( l \xrightarrow{f(z_{t}, \cdot), g} l' \) in \( O \) with \( l \in L \) and \( l' \notin L \), we require guard \( g \) to imply \( l' = z_t \). We only introduce guarantees \( E_L \) for sets of locations \( L \) that are closed under interference.

Type environments \( \Gamma \) 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 \( \Gamma \). We write \( \Gamma, x : T \) for \( \Gamma \cup \{ x : T \} \). If the type of \( x \) does not matter, we just write \( \Gamma, x \). The initial type environment \( \Gamma_{init} \) 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 \( \Gamma_{pre} \) but \( x : T' \) in the type environment \( \Gamma_{post} \) 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, \( O = O_{base} \times O_{SMR} \). Guarantee \( E_L \) already comes with a set of locations. Guarantee \( S \) grants safe access to the tracked address. In terms of locations, it should not be possible to free the address stored in \( z_a \). We define \( SafeLoc(O) \) to be the largest set of locations in \( O \) that is closed under interference from other threads and for which all transition \( l \xrightarrow{\text{free}(x), g} l' \)

---

**Chapter 8 Strong Pointer Races**
with \( I \in \text{SafeLoc}(O) \) and \( g \land a = z_a \) satisfiable lead to an accepting location \( I' \). Guarantee \( A \) is characterized by location \( L_2 \) in \( O_{\text{Base}} \). 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 \texttt{malloc}). Hence, we separately ensure that \( z_a \) is allocated, and thus active indeed, whenever \( A \) is assigned. Along the same lines, we also use location \( L_2 \) for \( L \).

The discussion yields the following definition.

\textbf{Definition 8.9 (Meaning of Types).} The locations associated with types \( T \), \( \text{Loc}(T) \), are:

\[
\begin{align*}
\text{Loc}(\emptyset) & := \text{Loc}(O) \\
\text{Loc}(A) & := \{ L_2 \} \times \text{Loc}(O_{\text{SMR}}) \\
\text{Loc}(L) & := \{ L_2 \} \times \text{Loc}(O_{\text{SMR}}) \\
\text{Loc}(\top) & := \text{Loc}(T_1 \land T_2) := \text{Loc}(T_1) \cap \text{Loc}(T_2).
\end{align*}
\]

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 \( \text{com} \) to transform \( x : T \) into \( x : T' \) if the locations associated with \( T' \) over-approximate the post-image under \( x \) and \( \text{com} \) of the locations associated with \( T \).

We capture those transformations with the \textit{type transformer relation} \( T, x, \text{com} \sim T' \). To make it precise, we first define the post-image \( \text{post}_{p,\text{com}}(L) \) on the locations of the SMR automaton. The post-image yields a set of locations \( L' \) reachable by taking a \( \text{com} \)-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 \( \text{com} \). Second, if \( p \) appears as a parameter in \( \text{com} \), then guard \( g \) must allow \( z_a \) to track \( p \). Formally, these requirements translate to the satisability of \( g \land t = z_t \) and \( g \land p = z_a \), respectively. The parameterization in \( p \) makes the post-image precise. To see this, consider \( O_{\text{Base}} \) and the command \( \text{com} = \text{in:retire}(p) \). We expect the post-image of \( L_2 \) wrt. \( \text{com} \) and \( p \) to be \( \text{post}_{p,\text{com}}(L_2) = \{ L_3 \} \). 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.

\textbf{Definition 8.10 (Type Transformer).} The type transformer relation \( T, x, \text{com} \sim T' \) is defined by the following conditions:

\[
\begin{align*}
\text{post}_{x,\text{com}}(\text{Loc}(T)) & \subseteq \text{Loc}(T') \\
\text{isValid}(T') & \Rightarrow \text{isValid}(T) \\
\{ L, A \} \cap T' & \subseteq \{ L, A \} \cap T.
\end{align*}
\]

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 (\( A \) or \( I \)) and the SMR command has been executed, that it is safe to access
the address (S). The last inclusion states that SMR commands cannot establish the guarantees \( L \) and \( A \). It is worth pointing out that the relation \( T, x, com \xrightarrow{} T' \) 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 \( \Gamma, com \xrightarrow{} \Gamma' \) if we have \( \Gamma(x), x, com \xrightarrow{} T'(x) \) for all pointers/angels \( x \). We write \( \Gamma \xrightarrow{} \Gamma' \) if we take the post-image to be the identity. For an example, refer to Section 8.4.1.

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

\[
\text{rm}(\Gamma) := \{ x : T \setminus \{ A \} | x : T \in \Gamma \land x \notin \text{shared} \} \cup \{ x : \emptyset | x \in \text{shared} \}.
\]

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 \( \vdash \{ \Gamma_{\text{init}} \} \text{stmt} \{ \Gamma \} \) to indicate that \( \{ \Gamma_{\text{init}} \} \text{stmt} \{ \Gamma \} \) is derivable with the given rules. We write \( \vdash \text{stmt} \) if there is a type environment \( \Gamma \) so that \( \vdash \{ \Gamma_{\text{init}} \} \text{stmt} \{ \Gamma \} \). 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 \( A \) 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:

\[
\text{stmt} ::= \text{stmt}; \text{stmt} \mid \text{stmt} \oplus \text{stmt} \mid \text{stmt}^* \mid \begin{array}{ll}
\text{beginAtomic}; \text{stmt}; \text{endAtomic} \\
\text{beginAtomic}; \text{com}; \text{endAtomic}
\end{array}.
\]
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 $L$ 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 $L$ 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 $L$ 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 $L$ is removed due to the alias. All other
assumptions have no effect on the type environment, Rule \((\text{assume}2)\). Similarly, Rule \((\text{equal})\) joins type information in the case of assertions. However, no validity check is performed and \(L\) is not removed. Rule \((\text{active})\) adds the \(\Lambda\) 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 \(L\), as they hold full sets of addresses, Rule \((\text{angel})\). We can also assert membership of an address held by a pointer in a set of addresses held by an angel, Rule \((\text{member})\).

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, \(\text{enterQ}\) has an effect on all pointers). The post type environment of Rules \((\text{except})\) and \((\text{exit})\) simply infers guarantees wrt. the pre type environment and the emitted event. Note that this is the only way to infer SMR-specific guarantees \(E_L\), i.e., these guarantees solely depend on the SMR commands. Moreover, Rule \((\text{enter})\) performs a strong pointer race check. We define predicate \(\text{SafeCall}(\Gamma, \text{func}(\tau))\) to hold if the command \(\text{in:func}(\tau)\) is guaranteed not to be a racy call given the types in \(\Gamma\). The formalization coincides with the one of racy calls from before, Definition 7.12, except that it replaces the actual validity \(\text{valid}_\tau\) in a computation \(\tau\) by the under-approximation \(\text{isValid}_a\). A special case of Rule \((\text{enter})\) 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 \((\text{infer})\) allows for type transformations at any point, in particular, to establish the proper pre/post environments for Rules \((\text{choice})\) and \((\text{loop})\). Entering an atomic block, Rule \((\text{begin})\), has no effect on the type environment. Exiting an atomic block allows for interference. Hence, Rule \((\text{exit})\) removes any type information from the type environment that can be tampered with by other threads. Sequences of statements are straightforward, Rule \((\text{seq})\). Choices require a common pre and post type

---

**Figure 8.13**: Type rules for statements.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Pre-condition</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\text{infer}))</td>
<td>(\Gamma_1 \rightsquigarrow \Gamma_2) {(\Gamma_2) } (\text{stmt}) {(\Gamma_3)}</td>
<td>(\Gamma_1 \rightsquigarrow \Gamma_4) {(\Gamma_4)}</td>
</tr>
<tr>
<td>((\text{begin}))</td>
<td>(\Gamma) {beginAtomic}(\Gamma)</td>
<td>(\Gamma) {beginAtomic}(\Gamma)</td>
</tr>
<tr>
<td>((\text{end}))</td>
<td>({\Gamma}) {endAtomic}((\Gamma)) {(\text{rm}(\Gamma))}</td>
<td>{(\Gamma)} {(\text{stmt})}((\Gamma_1)) {(\text{stmt})}((\Gamma_2)) {(\text{stmt})}((\Gamma_3))</td>
</tr>
<tr>
<td>((\text{seq}))</td>
<td>{(\Gamma)} {(\text{stmt})}((\Gamma_1)) {(\text{stmt})}((\Gamma_2)) {(\text{stmt})}((\Gamma_3))</td>
<td>{(\Gamma)} {(\text{stmt})}((\Gamma_4)) {(\text{stmt})}((\Gamma_5))</td>
</tr>
<tr>
<td>((\text{choice}))</td>
<td>{(\Gamma)} {(\text{stmt})}((\Gamma_1)) {(\Gamma_2)} {(\text{stmt})}((\Gamma_3)) {(\Gamma_4)}</td>
<td>{(\Gamma)} {(\text{stmt})}((\Gamma_5)) {(\text{stmt})}((\Gamma_6))</td>
</tr>
<tr>
<td>((\text{loop}))</td>
<td>{(\Gamma)} {(\text{stmt})}((\Gamma_1)) {(\Gamma)} {(\text{stmt})}((\Gamma_2)) {(\Gamma)} {(\text{stmt})}((\Gamma_3))</td>
<td>{(\Gamma)} {(\text{stmt})}((\Gamma_4)) {(\Gamma)} {(\text{stmt})}((\Gamma_5))</td>
</tr>
</tbody>
</table>
environment, Rule (choice). Loops require a type environment that is stable under the loop body, Rule (loop).

8.3.4 Soundness

Our goal is to show that a successful type check ⊢ \( P \) 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 \( l_{\text{init}} \) be the initial location in \( O \). We write \( \tau \vDash_\varphi T \) if there is a location \( l \in \text{Loc}(T) \) associated with the type \( T \) so that \( (l_{\text{init}}, \varphi) \not\sim H(x) (l, \varphi) \). The definition is parameterized in the valuation \( \varphi \) determining the thread and the address to be tracked. We write \( \tau \vDash_\tau \delta \top \) if for every address \( a \), \( \tau \vDash_\tau \top \). 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 \( \tau \vDash_\tau \delta \top \) if we have \( \tau \vDash_\tau \delta \top \) 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(•) approximates validity. To make this precise, we define the straight-line version stmt(\( \tau \), \( t \)) of program \( P \) induced by \( \tau \) and \( t \). It is obtained by projecting \( \tau \) to the commands of thread \( t \). Furthermore, we define the relation \( \vDash_\Gamma \{ \Gamma_{\text{init}} \} \text{stmt}(\tau, t) \{ \Gamma \} \). It requires that (i) \( \tau, t \vdash_\Gamma \) 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 \( \vDash \).

Theorem 8.14 (Soundness). For all threads \( t \) and all \( \tau \in O[\| P \|_{\text{Adr}}^\varnothing \) with inv(\( \tau \)) we have:

\[ \vdash \{ \Gamma_{\text{init}} \} \text{stmt}(\tau, t) \{ \Gamma \} \implies \vDash_\Gamma \{ \Gamma_{\text{init}} \} \text{stmt}(\tau, t) \{ \Gamma \}. \]

Proof Sketch. We proceed by induction on the length of \( \tau \in O[\| P \|_{\text{Adr}}^\varnothing \). Let \( t \) be a thread with:

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

The induction hypothesis links the current type environment \( \Gamma \) 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 \( L \in T \), 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 \( L \) according to the type rules.

(iv) If \( A \in T \), 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, \( \tau \cdot \text{act} \). It can be summarized as follows. Property (i) continues to hold for \( \tau \cdot \text{act} \) because the type \( T \) of \( x \) is closed under interference; for \( L \) and \( A \) we argue separately in the following. If \( L \in T \), then \( \text{act} \) cannot use a valid alias of \( x \). In particular, it cannot retire \( x \) according to the premise of Rule (enter). If \( A \in T \), then thread \( t \) is in an atomic block and there is no chance to append action \( \text{act} \) of another thread. The case does not occur. Consider property (ii).

Assume \( \text{isValid}(T) \) holds. That is, \( T \) contains one of \( A \cdot L \cdot S \). If \( L \neq T \) or \( A \neq T \), then the above reasoning for (i) already implies (ii). Otherwise, we have \( S \in T \). It implies (ii) because \( S \) is closed under interference. Property (iii) follows from the fact that \( \text{act} \) 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 \( \text{inv}(O \llbracket P \rrbracket_{\text{Addr}}) \) and \( \vdash P \), then \( O \llbracket P \rrbracket_{\text{Addr}} \) is free from strong pointer races.

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

**Proof Sketch.** To see the theorem, consider \( \tau \cdot \text{act} \in O \llbracket P \rrbracket_{\text{Addr}} \). We focus on the case where the last action is a dereference, say due to command \( \text{com} \) 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 \( \text{stmt}(\tau \cdot \text{act}, t) = \text{stmt}; \text{com} \) 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:

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

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 \( \Gamma \) with \( \{ \Gamma_{\text{init}} \} \text{stmt}; \text{com} \{ \Gamma \} \). The only way to type the sequential composition \( \text{stmt}; \text{com} \) is Rule (seq). It requires a type environment \( \Gamma' \) so that both \( \{ \Gamma_{\text{init}} \} \text{stmt} \{ \Gamma' \} \) and \( \{ \Gamma' \} \text{com} \{ \Gamma \} \) 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 in general and SMR automaton $O_{Base}$ in particular.

Theorem 8.16. If $inv(O[P]^O_{Adr})$ and $\vdash P$, then $O[P]^Adr$ is free from double retires.

Proof Sketch. The argumentation is along the lines of Theorem 8.15. However, we have to deal with computations from full $O[P]^{Adr}$. To the contrary, assume there is $\tau.act \in O[P]^{Adr}$ which performs a double retire on address $a$. That is, $act$ executes command $\text{retire}(p)$ with $m_\tau(p) = a$ and $a \in \text{retired}_\tau$. By Theorem 8.15, there are no strong pointer races. Then, Theorem 8.5 yields another computation $\sigma.act \in O[P]^{O}_{Adr}$ such that $\tau \sim \sigma$ and $\text{retired}_\tau \subseteq \text{retired}_\sigma$. In order to retire $p$, Rule (enter) requires pointer $p$ to hold guarantee $A$. This, in turn, means $p$ is valid. We have $m_\sigma(p) = a$. Furthermore, $a \in \text{retired}_\tau$ implies $a \in \text{retired}_\sigma$ and thus we conclude that $O_{Base}$ is in location $L_3$. This, however, contradicts the activeness of $p$, which guarantees that $O_{Base}$ 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 $inv(O[P]^O_{Adr})$ 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 $E_{acc}$ is sufficient. We define $Loc(E_{acc})$ to be those locations where thread $z_t$ is guarantee to have returned from a call to $\text{leave}\overline{Q}$ but has not yet invoked $\text{enter}\overline{Q}$. That is, guarantee $E_{acc}$ captures when $z_t$ is accessing the data structure. The sets of locations represented by $A$, $S$, and $E_{acc}$ can be read of the cross-product SMR automaton $O_{Base} \times O_{EBR}$ in Figure 8.17. It is worth pointing out that $Loc(S)$ 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 $Loc(S)$ 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 \texttt{re:leaveQ} and the inference of guarantee \( S \).

First, we establish the type transformer relation \( \emptyset, x, \texttt{re:leaveQ} \leadsto \mathbb{E}_{\text{acc}} \). This boils down to checking the inclusion:

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

because the remaining properties of the type transformer relation are trivially satisfied (we do not add any of \{ \( A \), \( L \), \( S \) \}). 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 \( O_{\text{base}} \times O_{\text{EBR}} \) from Figure 8.17. We compute the post-image of \( L \) wrt. \( x \) and \( \texttt{re:leaveQ} \) in \( O_{\text{base}} \times O_{\text{EBR}} \). To this end, we consider all transitions labeled with \( \texttt{re:leaveQ}(t) \). The pointer or angel \( x \) does not play a role. We derive the desired inclusion as follows:

\[
\text{post}_{x, \texttt{re:leaveQ}}(\text{Loc}(\emptyset)) = \text{post}_{x, \texttt{re:leaveQ}}(L) = L \setminus \{(L_2, L_4), (L_3, L_4)\} = \text{Loc}(\mathbb{E}_{\text{acc}}).
\]
Second, we show how to infer $S$. From Figure 8.17 we know that $E_{acc}$ alone does not yield $S$ because of location $(L_5, L_3)$; we also need $A$. We establish $E_{acc} \land A \rightsquigarrow E_{acc} \land A \land S$. Since $E_{acc} \land A$ is valid and we do not add $L$, the key task is to establish

$$\text{Loc}(E_{acc} \land A) \subseteq \text{Loc}(E_{acc} \land A \land S).$$

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

$$\text{Loc}(E_{acc} \land A) = \text{Loc}(E_{acc}) \cap \text{Loc}(A) = \{ (L_2, L_5), L_{13} \} \subseteq \{ (L_2, L_5), (L_3, L_6), L_{13} \} = \text{Loc}(S).$$

### 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 $S$, 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 764 to 767. Later, upon accessing/dereferencing a pointer $p$, we make sure that $r$ captures the address pointed to by $p$, Lines 771 and 774.

Note that, conceptually, we want to execute Lines 766 and 767 atomically so that angel $r$ can precisely capture the addresses active when `leaveQ` returns, Line 766. However, there is no need to introduce this atomic block: the `dequeue` operation cannot acquire pointers to those addresses that become inactive between Lines 766 and 767 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 779 with type $\emptyset$ for all pointers and the angel $r$. The allocation of $r$ in Line 780 has no effect on the type assignment. Line 782 invokes leaveQ. Again, the types are not affected because the SMR automaton has no transitions labeled with $\text{in}: \text{leaveQ}$. Next, the invocation returns, Line 784. Following the discussion from Section 8.4.1, we obtain $\text{E}_{\text{acc}}$ for $r$, Line 785. 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 $\text{E}_{\text{acc}}$ of $r$ to $\mathbb{S}$. We annotate $r$ to hold a set of active addresses, Line 786. This yields type $\text{E}_{\text{acc}} \land \mathbb{A}$ for $r$, Line 787. As explained above, we can now lift this type to $\text{E}_{\text{acc}} \land \mathbb{A} \land \mathbb{S}$, Line 788. Recall that the allocation of $r$ in Line 780 is angelic. So the addresses held by $r$ will indeed be chosen to be active. In Line 789, 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 795 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 $\text{E}_{\text{acc}} \land \mathbb{S}$ from $r$ over to head. As a consequence, the dereference of head in Line 797 is safe. Similarly, we require next to be captured by $r$ in Line 800 such that the dereference in Line 802 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 772 (Line 330 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 $\texttt{@inv active(head)}$ annotation to Line 772. This makes head valid and the type inference go through for Line 772. 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 $\texttt{leaveQ}$ call. We use a template to transform the sequence $\texttt{leaveQ}; \texttt{re}$ to the code from Lines 764 to 767.\footnote{A subsequent use of this tactic will skip this step and reuse the existing angel.} Then, we fix Line 772 by prepending the annotation $\texttt{@inv head in r}$, as shown in Line 771. This makes head valid. Whether or not the annotation holds is again checked with the technique.

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 an example typing of Michael&Scott’s $\texttt{dequeue}$ method. There, we use $O_{\text{base}} \times O_{\text{hp}}^{0,1}$ as specification; for the types obtained from $O_{\text{base}} \times O_{\text{hp}}^{0,1}$ 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 rational 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 detail in Section 8.7.
Figure 8.20: An example of HP-specific types and an application to Micheal&Scott’s queue.

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

$$ F \ := \ \text{free}(a), \ a = z_a \quad R \ := \ \text{in:retire}(t,a), \ a = z_a $$

(b) A typing for an excerpt of the dequeue function from Michael&Scott’s queue with HP. Lines 326 to 337 from Figure 2.13. To guide the type check, we added annotations. The added lines are typeset in bold font. The atomic blocks, Lines 810 to 821 and Lines 832 to 840, can be obtained with the technique from Section 8.7 plus an HP-specific tactic for inserting active annotations.
8.5 Invariant Checking

The type system from Section 8.3 relies on invariant annotations in the program under scrutiny in order to incorporate runtime behavior that is typically not available to a type system. For the soundness of our approach, we require those annotations to be correct. More precisely, the premises of Theorems 8.15 and 8.16 require the annotations to be correct under \( O[\mathcal{P}]_{Adr} \). 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 \( \mathcal{P} \), suffices indeed. Technically, we extend Theorem 8.15 because the reduction from Theorem 8.6 requires strong pointer race freedom.

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

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

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 \( \text{assert } cond \) and non-deterministic assignments \( \text{havoc}(p) \) 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 \( \text{safe}(\cdot) \) evaluates to true iff the assertions hold, i.e., verification is successful. Recall that \( \mathcal{P} \) 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 \( inv(\mathcal{P}) \) iff \( \text{safe}(\text{inst}(P)) \).

The source-to-source translation is linear in size.
Invariants guaranteeing pointer equality yield assertions.

The purpose of invariants \(\text{inv active}(p)\) 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 \text{retire} call after which such an invariant will be checked. We instrument the program by an additional pointer \text{retire\_ptr} and a Boolean variable \text{retire\_flag}. Both are shared. An invocation of \text{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 \text{retire\_ptr} and \text{retire\_flag} is raised. Note that the instrumentation is tailored towards garbage collection. As long as \text{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 \(\text{inv active}(p)\) 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 \text{retire} invocations is not atomic. Hence, there may be an interleaving where a pointer has been stored in \text{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 \( \text{included}_r \) and \( \text{failed}_r \) that are local to the thread. When we allocate an angel using \( @\text{inv} \text{angel} \ r \), 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 \( \text{included}_r \). For (ii), we use \( \text{failed}_r \). If we are sure of both facts, we let verification fail. Note that we can derive the facts in arbitrary order.

An invariant \( @\text{inv} \ q \ \text{in} \ r \) 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 \( \text{included}_r \). Moreover, we check (ii). If the address has been retired already, then we report a bug.

Invariant \( @\text{inv} \ \text{active}(r) \) forces all addresses held by the angel to be active. In the instrumented program, \( r \) is a pointer that we compare to \( \text{retire}_\text{ptr} \) introduced above. If the address has been retired, we are sure about (ii) and document this by raising \( \text{failed}_r \). 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 \( O(|P|^2) \).

As common in type systems [Pierce 2002], our algorithm for type inference is constraint-based. We associate with program \( P \) a constraint system \( \Phi(\Gamma_{\text{init}}, P, X) \). 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 \( \Gamma \) can be assigned to \( X \) in order to solve the constraint system if and only if the derivation \( \{ \Gamma_{\text{init}} \} \ P \{ \Gamma \} \) 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 \( \text{Types} \) of all types, ordered by \( \rightsquigarrow \). The problem
Figure 8.25: Constraint system $\Phi(X, P, Y)$, defined inductively over the structure of $P$.

$\Phi(X, \text{com}, Y) : sp(X, \text{com}) \subseteq Y$

$\Phi(X, \text{stmt}_1; \text{stmt}_2, Y) : \Phi(X, \text{stmt}_1, Z) \land \Phi(Z, \text{stmt}_2, Y)$ with $Z$ fresh

$\Phi(X, \text{stmt}_1 \oplus \text{stmt}_2, Y) : \Phi(X, \text{stmt}_1, Y) \land \Phi(X, \text{stmt}_2, Y)$

$\Phi(X, \text{stmt}^*, Y) : \Phi(Y, \text{stmt}, Y) \land X \subseteq Y$

is that types $E_L$ and $E_{L'}$ with $L \subseteq L'$ are comparable, $E_L \leadsto E_L \land E_{L'}$ and $E_L \land E_{L'} \leadsto E_L$, yet distinct. This is not merely a theoretical issue of the domain being a quasi order instead of a partial order. It means we compute over too large a domain, namely a powerset lattice where we should have used a lattice of antichains [Wulf et al. 2006]. We factorize the set of all types along such equivalences $\leadsto \cap \leadsto^{-1}$. The resulting $\text{AntiChainTypes} := (\text{Types} / \leadsto \cap \leadsto^{-1}, \leadsto$) is a complete lattice [Birkhoff 1948, Theorem 3].

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

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

The constraint system $\Phi(\Gamma_{\text{init}}, P, X)$ is defined in Figure 8.25. We proceed by induction over the structure of the statements in $P$ and maintain triples $(X, \text{stmt}, Y)$. The idea is that statement $\text{stmt}$ 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 $sp(X, \text{com})$ to be the strongest enriched type environment resulting from the environment in $X$ when applying command $\text{com}$. The constraint $sp(X, \text{com}) \subseteq Y$ 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 $\text{stmt}_1$ from $X$. We then require $\text{stmt}_2$ to transform this environment into $Y$. For a choice, $Y$ should upper bound the effects of both $\text{stmt}_1$ and $\text{stmt}_2$ 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 $\text{stmt}$ to $X$. This means the environment in $Y$ is at least $X$ because the iteration may be skipped. Furthermore, if we apply $\text{stmt}$ to $Y$ then we should again obtain at most the environment in $Y$.

---

3 Quasi orders are reflexive and transitive; partial orders are antisymmetric quasi orders [Birkhoff 1948, p. 4].
It remains to define $sp(X, \text{com})$, the strongest enriched type environment resulting from $X$ under command $\text{com}$. We refer to the typing rules in Figures 8.12 and 8.13 and extract $\text{pre}_\text{com}$ and $\text{up}_\text{com}$. The former is a predicate on environments capturing the premise of the rule associate with command $\text{com}$. 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:

$$
\begin{array}{c}
\Gamma(q) = T \quad \text{isValid}(T) \\
\{ \Gamma, p \} p := q.\text{next} \{ \Gamma, \rho : \emptyset \}
\end{array}
$$

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

$$
sp(\Gamma, \text{com}) := \begin{cases} 
\text{up}_\text{com}(\Gamma) & \text{if } \text{pre}_\text{com}(\Gamma) \\
T & \text{otherwise}
\end{cases}.
$$

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

We apply Kleene iteration to obtain the least solution to the constraint system $\Phi(\Gamma_{\text{init}}, P, X)$. The least solution is a function $\text{lsol}$ 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 $\text{lsol}(X)$ 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 $\Phi(\Gamma_{\text{init}}, P, X)$ we have $\text{lsol}(X) = \bigcap \{ \Gamma \} p \{ \Gamma \} \Gamma$. Hence, $\text{lsol}(X) \neq T$ 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 $|\text{Var}| \cdot |\{ A, L, S, E_{L_1}, \ldots, E_{L_n} \}| + 1$. 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 $sp(\bullet, \text{com})$ 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 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., $O\left[P\right]_{\text{Adr}} \not\subseteq O\left[Q\right]_{\text{Adr}}$. Instead, we just require that the correctness of $Q$ entails the correctness of $P$.

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

<table>
<thead>
<tr>
<th>(a) Protection scheme from Michael&amp;Scott’s queue, Lines 326 to 328. For the sake of presentation, the original condition from Line 328 is rewritten into the assumptions from Lines 849 and 852. Line 852 exhibits a strong pointer race; more precisely, an unsafe assumption.</th>
<th>(b) More atomic version of the protection mechanism from Michael&amp;Scott’s queue. The new version avoids the strong pointer race from Line 852, yet it retains all behaviors from the original implementation.</th>
</tr>
</thead>
</table>
| ```
845  unprotect\_0();
846  head = Head;
847  protect\_0(head);
848  if (*) {
849  assume(head != Head);
850  // restart procedure
851  }
852  assume(head == Head);
853  // continue procedure
``` | ```
854  unprotect\_0();
855  if (*) {
856  head = Head;
857  protect\_0(head);
858  assume(head != Head);
859  // restart procedure
860  }
861  atomic { head = Head;
862  protect\_0(head); }
863  // continue procedure
``` |
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 326 to 328 of Michael & Scott’s queue the code of which is repeated in Figure 8.27a (for simplicity, we added the latest unprotect in Line 845). There, (i) the shared Head is copied into a local pointer head, Line 846, (ii) a protection is issued for that local pointer head, Line 847, and then (iii) the procedure is restarted if Head has changed since it was read out, Line 849, and continued otherwise, Line 852. The comparison of Head and head in Line 852 raises a pointer race: head may be invalid despite the protection in Line 847. The reason is that Lines 846 and 847 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 852 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 849 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 $O$. 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 $P$. 

Section 8.7 Avoiding Strong Pointer Races
Definition 8.28 (Right Mover). Command \(com\) at label \(lab\) right-moves if, for all computations \(\tau.\text{act}_1, \text{act}_2 \in \mathcal{O}[\mathbb{P}]^{Adr}\) with \(\text{act}_i = \langle t_i, \text{com}_i, \text{up}_i \rangle\) and \(t_1 \neq t_2\) where \(\text{com}_1\) is command \(com\) at label \(lab\), there is \(\tau.\text{act}_2', \text{act}_1' \in \mathcal{O}[\mathbb{P}]^{Adr}\) with \(\text{act}_i' = \langle t_i, \text{com}_i, \text{up}_i \rangle\) so that:

\[
\begin{align*}
\text{m}_{\tau.\text{act}_1, \text{act}_2} &= \text{m}_{\tau.\text{act}_1', \text{act}_2'} \\
\text{and} & \quad \text{good}(\tau.\text{act}_2', \text{act}_1') \implies \text{good}(\tau.\text{act}_1) \land \text{good}(\tau.\text{act}_2) \\
\text{and} & \quad \forall a \in \text{Adr}. \mathcal{F}_O(\tau.\text{act}_1, \text{act}_2, a) \subseteq \mathcal{F}_O(\tau.\text{act}_2', \text{act}_1', a).
\end{align*}
\]

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 \(\text{act}_1\) to the right. It is worth pointing out that swapping the order of \(\text{act}_1\) and \(\text{act}_2\) might change the updates they perform. This is why we use actions \(\text{act}_1'\) and \(\text{act}_2'\) after moving \(\text{act}_1\) to the right: actions \(\text{act}_1\) and \(\text{act}_2'\) coincide up to a potentially different update.

What is remarkable about the definition of moverness is that SMR commands (\(\text{in}, \text{re}, \text{free}\)) 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 \(\text{free}\) of address \(a\) does not right-move over a \(\text{malloc}\) 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 \(\mathcal{O}[\mathbb{P}]^{Adr}\). 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 \(m_{\tau}\). 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 \(\text{act}_1\) and \(\text{act}_2\) execute SMR commands. Since neither of the actions updates the memory, we have \(\text{act}_i = \text{act}_i'\). The inclusion boils down to:

\[
\mathcal{F}_O(\tau.\text{act}_1, \text{act}_2, a) \subseteq \mathcal{F}_O(\tau.\text{act}_2, \text{act}_1, a).
\]
Let the histories of the involved computations be \( h_1^{evt_1}, h_2^{evt_2} \) resp. \( h_2^{evt_2}, h_1^{evt_1} \). Runs of the SMR automaton \( \mathcal{O} \) on those histories take the form

\[
\begin{align*}
& s_0 \xrightarrow{h} s_1 \xrightarrow{evt_1} s_2 \xrightarrow{evt_2} s_3 \\
& \text{and} \quad s_0 \xrightarrow{h} s_1 \xrightarrow{evt_2} s_2 \xrightarrow{evt_1} s_3.
\end{align*}
\]

Consequently, it suffices to show that state \( s_3 \) allows for more behavior than \( s_5 \). Formally, we require that \( \mathcal{F}_\mathcal{O}(s_3, a) \subseteq \mathcal{F}_\mathcal{O}(s_5, a) \) holds where \( \mathcal{F}_\mathcal{O}(s, a) := \{ h' \mid h' \in \mathcal{S}(s) \wedge \text{frees} h \subseteq a \} \).

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 s.e.a.l.\(^4\) 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 \( \mathcal{E}_L \) on the fly; there is no need for the user to manually specify them. To substantiate the usefulness of our approach, we empirically evaluated s.e.a.l 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 \( \mathcal{O}_{\text{Base}} \times \mathcal{O}_{\text{EBR}} \) and \( \mathcal{O}_{\text{Base}} \times \mathcal{O}_{\text{HP}}^{0.1} \), 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 \( \mathcal{O}[\bullet]_{\text{adr}}^{\text{one}} \), 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.

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.

\(^4\) s.e.a.l is freely available at: \( \mathcal{O} \) 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.

<table>
<thead>
<tr>
<th>SMR</th>
<th>Program</th>
<th>Type Inference</th>
<th>Annotations</th>
<th>Linearizability</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP</td>
<td>Treiber’s stack</td>
<td>0.7s ✓</td>
<td>12s ✓</td>
<td>1s ✓</td>
</tr>
<tr>
<td></td>
<td>Opt. Treiber’s stack</td>
<td>0.5s ✓</td>
<td>11s ✓</td>
<td>1s ✓</td>
</tr>
<tr>
<td></td>
<td>Michael&amp;Scott’s queue</td>
<td>0.6s ✓</td>
<td>12s ✓</td>
<td>4s ✓</td>
</tr>
<tr>
<td></td>
<td>DGLM queue</td>
<td>0.6s ✓</td>
<td>1s Xa</td>
<td>5s ✓</td>
</tr>
<tr>
<td></td>
<td>Vechev&amp;Yahav 2CAS set</td>
<td>1.2s ✓</td>
<td>13s ✓</td>
<td>98s ✓</td>
</tr>
<tr>
<td></td>
<td>Vechev&amp;Yahav CAS set</td>
<td>1.2s ✓</td>
<td>3.5h ✓</td>
<td>42m ✓</td>
</tr>
<tr>
<td></td>
<td>ORVYY set</td>
<td>1.2s ✓</td>
<td>3.2h ✓</td>
<td>47m ✓</td>
</tr>
<tr>
<td></td>
<td>Michael’s set</td>
<td>1.2s ✓</td>
<td>90s Xa</td>
<td>—</td>
</tr>
</tbody>
</table>

EBR

| Treiber’s stack           | 0.6s ✓         | 10s ✓       | 1s ✓            |
| Michael&Scott’s queue     | 0.7s ✓         | 16s ✓       | 5s ✓            |
| DGLM queue                | 0.7s ✓         | 1s Xa        | 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 Xa       | —               |
| Harris’ set               | 0.5s ✓         | 1s Xa       | —               |

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

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].


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 \texttt{ccured} [Necula et al. 2002] and \texttt{blast} [Henzinger et al. 2003] due to Beyer et al. [2005], \texttt{invader} [Yang et al. 2008], \texttt{xisa} [Laviron et al. 2010], \texttt{slayer} [Berdine et al. 2011], \texttt{infer} [Calcagno and Distefano 2011], \texttt{forester} [Holik et al. 2013], \texttt{predator} [Dudka et al. 2013; Holik et al. 2016], and \texttt{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 \texttt{cave} [Vafeiadis 2010a,b] as it can also prove linearizability.

Despite the differences, we point out that the combination of \texttt{blast} and \texttt{ccured} [Beyer et al. 2005] is closest to our approach in spirit. \texttt{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 \texttt{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 \texttt{invader}, \texttt{slayer}, \texttt{infer}, \texttt{forester}, \texttt{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 $A$ 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 [Svensen 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 graces 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 task. 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 where first introduced by Lipton [1975]. They were later generalized to arbitrary safety properties [Back 1989; Doepner 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 `read` 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 326 to 328 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 \(\mathcal{O}[\text{P}]^{\text{one}}_{\text{one}} = \bigcup_a \mathcal{O}[\text{P}]^{(a)}\) rather than \(\mathcal{O}[\text{P}]^{\text{one}}_{\text{Adr}} = \bigcup_a \mathcal{O}[\text{P}]^{(a)}_{\text{Adr}}\) would suffice.

\(^1\) The author is a coauthor of [Haziza et al. 2016] some results of which are part of Chapter 6.
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 \( \tau \) is tied to the syntactic information of typings \( x : T \) for threads \( t \), denoted by \( \tau, t \vDash x : T \) in Section 8.3.4. Currently, \( t \) and \( m_s(t) \) 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 \( m_s(t) \), 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 \( m_s(t) \), 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.

The evaluation from Section 8.8 demonstrated that checking invariants for correctness is often-times 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, could 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


Appendices
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

1. the program variables \( \text{Var} = \text{PVar} \cup \text{DVar} \) into the variables of \( P \) and \( R \), \( \text{Var} = \text{Var}^P \uplus \text{Var}^R \), and
2. the pointer selectors \( \text{Sel} = \text{PSel} \cup \text{DSel} \) into the selectors of \( P \) and \( R \), \( \text{Sel} = \text{Sel}^P \uplus \text{Sel}^R \), such that we have \( a \cdot \text{sel} \in \text{Sel}^Q \) if \( b \cdot \text{sel} \in \text{Sel}^Q \) for all \( a, b \in \text{Adr} \) and \( Q \in \{ P, R \} \).

We may write \( \text{sel} \in \text{Sel}^Q \) instead of \( a \cdot \text{sel} \in \text{Sel}^Q \) 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 \( m_r = m_r^P \uplus m_r^R \) defined by

\[
m_r^P = m_r \downarrow \text{Var}^P \cup \text{Sel}^P \quad \text{and} \quad m_r^R = m_r \downarrow \text{Var}^R \cup \text{Sel}^R .
\]

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 \( \vdash \tau Q, t \) to indicate that thread \( t \) is taking a step due to \( Q \in \{ P, R \} \). Technically, we have \( \vdash \tau R, t \) if Rule (sos-std-smr) is involved in the derivation of the program step and \( \vdash \tau P, t \) 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 \( IVar \subseteq Var^R \setminus \text{shared} \) and assume that invocations \( \text{in}:\text{func}(r_1, \ldots, r_n) \) contain interface variables only, \( r_i \in IVar \) for all \( 1 \leq i \leq n \).

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

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:

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

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 \( \sigma \in \llbracket MGC(R) \rrbracket_{Adr}^\lambda \), there is \( \tau = \langle t, \text{com}, \text{up} \rangle \) such that \( \sigma.\text{act} \in \llbracket MGC(R) \rrbracket_{Adr}^\lambda \), provided one of the following applies:

(i) \( \text{com} \equiv \text{in}:\text{func}(r_1, \ldots, r_n) \) and \( r_i \in IVar \) and \( m_\sigma(r_i) \neq \text{seg} \) and \( \text{skip} \in ctrl^R(\sigma)(t) \),

(ii) \( \text{com} \equiv \text{re}:\text{func} \) and \( \text{await} \ \text{re}:\text{func} \in ctrl^R(\sigma)(t) \),

(iii) \( \text{com} \equiv p \equiv \text{malloc} \), or

(iv) \( \text{up} = \lfloor p \mapsto a \rfloor \) 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:

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

We use a single action \( \text{act} \) for simplicity; a generalization to sequences of actions that mimic \( \text{act} \) on the interface variables is straightforward. We now show that the simpler \( MGC \) over-approximates \( P \)’s usage of \( R \). Hence, \( R \vdash O \) guarantees that \( R \) adheres to the specification \( O \) when used by \( P \).
Theorem A.2 (Proof C.5). Let \( \tau \in \llbracket P(R) \rrbracket_{Adr} \) be free from separation violations. Then, there is \( \sigma \in \llbracket MGC(R) \rrbracket_{Adr} \) such that (i) \( \text{ctrl}^P(\tau) \subseteq \text{ctrl}(\sigma) \), (ii) \( m^R_\tau = m^R_\sigma \), (iii) \( m^p_\tau = m^p_\sigma \), (iv) \( H(\tau) = H(\sigma) \), (v) \( \text{fresh}_\tau \subseteq \text{fresh}_\sigma \), (vi) \( \text{freed}_\tau \subseteq \text{freed}_\sigma \), and (vii) \( \text{used}(\tau) \subseteq \text{used}(\sigma) \). \( \Box \)

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

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

Theorem A.4 (Proof C.7). Let \( \tau \in \llbracket P(R) \rrbracket_{Adr} \) be free from separation violations. If \( R \models O \), then there is \( \sigma \in \llbracket P \rrbracket_{Adr} \) such that (i) \( \text{ctrl}^P(\tau) \subseteq \text{ctrl}(\sigma) \), (ii) \( m^P_\tau = m^P_\sigma \), (iii) \( H(\tau) = H(\sigma) \), (iv) \( \text{fresh}_\tau \subseteq \text{fresh}_\sigma \), (v) \( \text{freed}_\tau \subseteq \text{freed}_\sigma \), and (vi) \( \text{retired}_\tau \subseteq \text{retired}_\sigma \). \( \Box \)

In Chapter 5 we introduced the predicate \( \text{good}(\tau) \) that tests whether or not a computation \( \tau \) reaches a bad program location which we assumed to be in \( P \). Formally, we let \( \text{Fault} \) be the bad program locations and define \( \text{good}(\tau) : \iff \text{ctrl}^P(\tau) \cap \text{Fault} = \emptyset \). 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: \( O_{\text{Base}} \times O^0_{\text{HP}} \times O^1_{\text{HP}} \) and \( O_{\text{Base}} \times O^0_{\text{HP}} \times O^1_{\text{HP}} \). We used the former in the example from Figure 8.20 for typing Micheal\&Scott’s queue. However, \( O_{\text{Base}} \times O^0_{\text{HP}} \times O^1_{\text{HP}} \) 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 \( O_{\text{Base}} \times O^0_{\text{HP}} \times O^1_{\text{HP}} \). The definition of \( O^0_{\text{HP}} \) is given Figure A.5. Intuitively, \( O^0_{\text{HP}} \) is the cross-product \( O^0_{\text{HP}} \times O^1_{\text{HP}} \) 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 \( O^0_{\text{HP}} \). If the same address is additionally protected with the 1-st hazard pointer, then \( O^0_{\text{HP}} \) moves to locations \( L_{19} \) to \( L_{21} \) after \( \text{protect}_1 \) is invoked and to locations \( L_{22} \) and \( L_{23} \) after \( \text{protect}_1 \) returns. When the protection of the 0-th hazard pointer is revoked, the protection is transferred to the 1-st hazard pointer. In \( O^1_{\text{HP}} \), 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 \( O^1_{\text{HP}} \). 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 $E_1, E_2, E_3, E_4$ 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 $E_1$ encodes that an invocation of protect_0 has been issued. Guarantee $E_2$ encodes that the invocation has returned. Formally, we have:

$$\emptyset, x, \text{in:protect}_0(x) \sim E_1 \quad \text{and} \quad E_1, x, \text{re:protect}_0 \sim E_2.$$  

Similarly, guarantees $E_3, E_4$ deal with protect_1. The protection is successful if it is issued while the address is active:

$$E_2 \land A \sim S \quad \text{and} \quad E_4 \land A \sim S.$$  

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 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 454 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 $\tau$ is a relaxed unsafe assumption if there is some assume $p = q \in \text{next-com}(\tau)$ such that (i) $p \notin valid$, $q \notin CExp$ or (ii) $p \notin CExp \land q \notin valid$.
Figure A.5: Hazard pointer specification $O_{\text{Base}} \times O_{\text{HP}}^{0.1}$ for two hazard pointer per thread supporting the transfer of protections. The HP-specific guarantees $E_0, E_1, E_2, E_3$ are needed when typing.

Section A.3  Relaxation of Strong Pointer Races

153
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 \( \tau.\text{act} \) is a moderate pointer race if (i) \( \text{act} \) is an ordinary pointer race, or (ii) \( \tau.\text{act} \) 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 \( \tau \) is a constant violation if there is a constant \( C \in \text{CVar} \) such that \( m_{\tau}(C) \in \text{frees}_\tau \cup \text{retired}_\tau \). \( \square \)

For simplicity, we assume that \( O[\llbracket P \rrbracket_{\text{Adr}}] \) 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 \( O[\llbracket P \rrbracket_{\text{Adr}}] \). \( \square \)

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 (\( \otimes \) Proof C.58).** If \( O \) supports elision and \( O[\llbracket P \rrbracket_{\text{Adr}}] \) is free from moderate pointer races and double retires, then we have \( \text{good}(O[\llbracket P \rrbracket_{\text{Adr}}]) \iff \text{good}(O[\llbracket P \rrbracket_{\text{Adr}}]) \) and \( O[\llbracket P \rrbracket_{\text{Adr}}] \) 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:

\[
\begin{align*}
& \text{(assume 1-constant)} \\
& \tau'' = (\tau \land \tau') \setminus \{\bot\} \quad \{ p, q \} \cap \text{CVar} \neq \emptyset \\
& \{ \Gamma, p : T, q : T' \} \quad \text{assume } p = q \\
& \{ \Gamma, p : T, q : T'' \}
\end{align*}
\]

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

**Theorem A.11 (\( \otimes \) Proof C.84).** If \( \text{inv}(O[\llbracket P \rrbracket_{\text{Adr}}]) \) and \( \vdash P \), then \( \text{inv}(O[\llbracket P \rrbracket_{\text{Adr}}]) \) and \( O[\llbracket P \rrbracket_{\text{Adr}}] \) 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 (\( \otimes \) Proof C.85).** If \( \tau \in O[\llbracket P \rrbracket_{\text{Adr}}] \), then \( m_{\tau}(\text{CVar}) \subseteq \text{active}(\tau) \). \( \square \)
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 $\tau$:

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

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

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

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

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

and similarly for sets of computations.

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

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

### 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 $\text{com}(\text{act}) = \text{com}$ to access the command $\text{com}$ executed by action $\text{act} = \langle t, \text{com}, \text{up} \rangle$.

#### B.1.1 Computations

**Definition B.1.** We define $\text{var} \cap \text{Adr} := \emptyset$ for $\text{var} \in P\text{Var} \cup D\text{Var}$ and $\text{a.sel} \cap \text{Adr} := \{ a \}$. □

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

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

**Definition B.4.** The set of thread-local variables of $t$ is $\text{local}_t = \{ p_t \mid p \notin \text{shared} \}$ of non-shared variables indexed by $t$. □
Definition B.5. The fresh addresses after a computation \( \tau \), denoted \( \text{fresh}_\tau \), are defined by:

\[
\begin{align*}
\text{fresh}_\tau & \triangleq \text{Adr} \\
\text{fresh}_{\tau, \text{act}} & \triangleq \text{fresh}_\tau \setminus \{ a \} \quad \text{if } \text{com}(\text{act}) \equiv \text{free}(a) \\
\text{fresh}_{\tau, \text{act}} & \triangleq \text{fresh}_\tau \setminus \{ a \} \quad \text{if } \text{com}(\text{act}) \equiv p := \text{malloc} \land m_{\tau, \text{act}}(p) = a \\
\text{fresh}_{\tau, \text{act}} & \triangleq \text{fresh}_\tau \quad \text{otherwise}
\end{align*}
\]

Definition B.6. The freed addresses after a computation \( \tau \), denoted \( \text{freed}_\tau \), are defined by:

\[
\begin{align*}
\text{freed}_\epsilon & \triangleq \emptyset \\
\text{freed}_{\tau, \text{act}} & \triangleq \text{freed}_\tau \cup \{ a \} \quad \text{if } \text{com}(\text{act}) \equiv \text{free}(a) \\
\text{freed}_{\tau, \text{act}} & \triangleq \text{freed}_\tau \setminus \{ a \} \quad \text{if } \text{com}(\text{act}) \equiv p := \text{malloc} \land m_{\tau, \text{act}}(p) = a \\
\text{freed}_{\tau, \text{act}} & \triangleq \text{freed}_\tau \quad \text{otherwise}
\end{align*}
\]

Definition B.7. The addresses freed in \( \tau \) are:

\[
\text{frees}_\tau \triangleq \{ a \mid \exists \tau_1, \tau_2. \ \tau = \tau_1 \cdot \tau_2 \land a \in \text{freed}_{\tau_1} \}.
\]

Definition B.8. The retired addresses after a computation \( \tau \), denoted \( \text{retired}_\tau \), are defined by:

\[
\begin{align*}
\text{retired}_\epsilon & \triangleq \emptyset \\
\text{retired}_{\tau, \text{act}} & \triangleq \text{retired}_\tau \cup \{ a \} \quad \text{if } \text{com}(\text{act}) \equiv \text{in:retire}(p) \land m_{\tau}(p) = a \\
\text{retired}_{\tau, \text{act}} & \triangleq \text{retired}_\tau \setminus \{ a \} \quad \text{if } \text{com}(\text{act}) \equiv \text{free}(a) \\
\text{retired}_{\tau, \text{act}} & \triangleq \text{retired}_\tau \quad \text{otherwise}
\end{align*}
\]

Definition B.9. The active addresses after \( \tau \) are:

\[
\text{active}(\tau) \triangleq \text{Adr} \setminus (\text{freed}_\tau \cup \text{retired}_\tau).
\]

Definition B.10. The set of control-flow-enabled commands after \( \tau \) are:

\[
\text{next-com}(\tau) \triangleq \{ \text{com} \mid \exists t, pc, stmt. \ pc \in \text{ctrl}(\tau) \land pc(t) \xrightarrow{\text{com}} stmt \}.
\]

Definition B.11. We define:

\[
VExp(\tau) \triangleq P\text{Var} \cup \{ a.\text{next} \mid a \in m_{\tau}(\text{valid}_\tau) \}.
\]

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

Definition B.13. The initial program counter \( \text{pc}_{\text{init}} \) is \( \text{pc}_{\text{init}} = \lambda t. P^t \circ \text{skip} \) for the standard semantics and \( \text{pc}_{\text{init}} = \lambda t. P^t \) for the SMR semantics.
B.1.2 SMR Automata

Definition B.14. The fresh addresses after a history \( h \), denoted by \( \text{fresh}_h \), are defined by:

\[
\begin{align*}
\text{fresh}_\varepsilon & := \text{Adr} \\
\text{fresh}_h.\text{free}(a) & := \text{fresh}_h \setminus \{ a \} \\
\text{fresh}_h.\text{in} : \text{func}(t, v_1, \ldots, v_k) & := \text{fresh}_h \setminus \{ v_1, \ldots, v_k \} \\
\text{fresh}_h.\text{re} : \text{func}(t) & := \text{fresh}_h
\end{align*}
\]

\( \square \)

Definition B.15. The addresses freed in \( h \) are:

\[
\text{frees}_h := \{ a \mid \exists h_1, h_2. \ h = h_1.\text{free}(a).h_2 \} .
\]

\( \square \)

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

\[
\forall l, l', f, t, \bar{r}, g. \quad (l \in L \land l \xrightarrow{f(t, \bar{r}), g} l' \land l' \notin L) \implies g \models t = z_t \\
\land (l \in L \land l \xrightarrow{\text{free}(\bullet), g} l') \implies l' \in L
\]

where \( \models \) denotes entailment among logic formulas.

\( \square \)

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

\[
\text{SafeLoc}(\mathcal{O}) := \bigcup \{ L \subseteq L_{\text{safe}} \mid L \text{ closed under interference} \}
\]

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

\( \square \)

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

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

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

\( \square \)

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

\[
\text{reach}_t^\mathcal{O}(h) := \{ l \mid \exists \varphi. (l_{\text{init}}, \varphi) \xrightarrow{*} (l, \varphi) \land \varphi(z_t) = t \land \varphi(z_a) = a \}
\]

where \( l_{\text{init}} \) is the initial location in \( \mathcal{O} \). For \( \text{seg} \) 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.

\( \square \)
B.1.3 Elision

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

\[
\begin{align*}
\text{swap}_{\text{exp}}(p) &= p & \text{swap}_{\text{exp}}(a.\text{next}) &= \text{swap}_{\text{adr}}(a).\text{next} \\
\text{swap}_{\text{exp}}(u) &= u & \text{swap}_{\text{exp}}(a.\text{data}) &= \text{swap}_{\text{adr}}(a).\text{data}
\end{align*}
\]

and a history mapping $\text{swap}_{\text{hist}}$ with

\[
\begin{align*}
\text{swap}_{\text{hist}}(\varepsilon) &= \varepsilon \\
\text{swap}_{\text{hist}}(h.\text{free}(a)) &= \text{swap}_{\text{hist}}(h).\text{free}(\text{swap}_{\text{adr}}(a)) \\
\text{swap}_{\text{hist}}(h.\text{in}:\text{func}(t, \overline{v})) &= \text{swap}_{\text{hist}}(h).\text{func}(t, \text{swap}_{\text{adr}}(\overline{v})) \\
\text{swap}_{\text{hist}}(h.\text{re}:\text{func}(t)) &= \text{swap}_{\text{hist}}(h).\text{re}:\text{func}(t)
\end{align*}
\]

for any SMR function $\text{func}$. For $\overline{v} = v_1, \ldots, v_k$ we use $\text{swap}_{\text{adr}}(\overline{v}) = \text{swap}_{\text{adr}}(v_1), \ldots, \text{swap}_{\text{adr}}(v_k)$.

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

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

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

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

158 Appendix B Meta Theory
section B.1  Formal Definitions

B.1.4 Races

Repeated Definition 6.5 (Valid Expressions). The valid pointer expressions in \( r \in O[p]_{Adr} \)
denoted by \( valid_r \subseteq PExp \), are defined by:

\[
valid_r := PVar
\]

\[
valid_{r, (t, p := q, up)} := valid_r \cup \{ p \} \quad \text{if } q \in valid_r
\]

\[
valid_{r, (t, p := q, up)} := valid_r \setminus \{ p \} \quad \text{if } q \notin valid_r
\]

\[
valid_{r, (t, p := q, next, up)} := valid_r \cup \{ a, next \} \quad \text{if } m_r(p) = a \in Adr \land q \in valid_r
\]

\[
valid_{r, (t, p := q, next, up)} := valid_r \setminus \{ a, next \} \quad \text{if } m_r(p) = a \in Adr \land q \notin valid_r
\]

\[
valid_{r, (t, p := q, next, up)} := valid_r \cup \{ p \} \quad \text{if } q \in valid_r \land m_r(q).next \in valid_r
\]

\[
valid_{r, (t, p := q, next, up)} := valid_r \setminus \{ p \} \quad \text{if } q \notin valid_r \lor m_r(q).next \notin valid_r
\]

\[
valid_{r, (t, free(a), up)} := valid_r \setminus invalid_a
\]

\[
valid_{r, (t, p := malloc, up)} := valid_r \cup \{ p, a, next \} \quad \text{if } [p \mapsto a] \in up
\]

\[
valid_{r, (t, assume p=q, up)} := valid_r \cup \{ p, q \} \quad \text{if } \{ p, q \} \cap valid_r \neq \emptyset
\]

\[
valid_{r, act} := valid_r \quad \text{otherwise}
\]

with \( invalid_a := \{ p \mid m_r(p) = a \} \cup \{ b, next \mid m_r(b, next) = a \} \cup \{ a, next \} \).

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

Repeated Definition 7.12 (Racy SMR Calls). A computation \( \tau, (t, \text{in:func}(r), \varnothing) \) performs a racy call if for \( H(\tau) = h \) and \( m_r(\overline{r}) = \overline{h} \) we have:

\[
\exists a \exists \overline{w}. \quad \left( \forall i. \left( v_i = a \lor r_i \in valid_r \lor r_i \in DExp \implies v_i = w_i \right) \right)
\]

\[
\wedge F_0(h.in:func(t, \overline{w}), a) \notin F_0(h.in:func(t, \overline{b}), a)
\]

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

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

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

Repeated Definition A.6 (Relaxed Unsafe Assumption). Computation \( \tau \) is prone to a relaxed unsafe assumption if there is assume \( p = q \in next-com(\tau) \) with (i) \( p, q \notin valid_r \), (ii) \( p \notin valid_r \land q \notin CExp \), or (iii) \( p \notin CExp \land q \notin valid_r \).

Repeated Definition A.7 (Moderate Pointer Race). A computation \( \tau, \text{act} \) is a moderate pointer race if (i) \( \text{act} \) is an ordinary pointer race, or (ii) \( \tau, \text{act} \) is a relaxed unsafe assumption.
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 $\text{dom}(m') := P \cup DVar \cup \{ a.data \in DExp \mid a \in m(P) \}$ and $m(e) = m'(e)$ for all $e \in \text{dom}(m')$.

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

Repeated Definition 7.7 (SMR Behavior). The behavior allowed by automaton $O$ on address $a$ after history $h$ is the set $F_O(h, a) := \{ h', h | h'.h' \in S(O) \land \text{frees}_h \subseteq a \}$.

Repeated Definition 7.8 (SMR Behavior Inclusion). Computation $\sigma$ includes the SMR behavior of $\tau$, denoted by $\tau \bowtie \sigma$, if $F_O(\tau, a) \subseteq F_O(\sigma, a)$ holds for all $a \in \text{adr}(m_{\tau|\text{valid}_\tau})$.

Repeated Definition 7.17 (Address Alignment). Two computations $\tau$ and $\sigma$ are $a$-aligned, denoted by $\tau \equiv_a \sigma$, if
\[
\forall p \in PVar. \ m_\tau(p) = a \iff m_\sigma(p) = a
\]
and
\[
\forall b \in m_\tau(\text{valid}_\tau). \ m_\tau(b.next) = a \iff m_\sigma(b.next) = a
\]
and
\[
a \in \text{fresh}_\tau \cup \text{freen}_\tau \iff a \in \text{fresh}_\sigma \cup \text{freen}_\sigma
\]
and
\[
F_O(\tau, a) \subseteq F_O(\sigma, a)
\]
and
\[
a \in \text{retired}_\tau \iff a \in \text{retired}_\sigma.
\]

Repeated Definition 7.18 (Harmful ABA). $O[F_{\text{PAdr}}^{\text{enc}}]$ is free from harmful ABAs if:
\[
\forall \sigma_a, act \in O[F_{\text{PAdr}}^{\text{enc}}] \forall \sigma_b \in O[F_{\text{PAdr}}^{\text{enc}}] \exists \sigma'_b \in O[F_{\text{PAdr}}^{\text{enc}}].
\]
\[
\sigma_a \sim \sigma_b \land \text{act} = \langle \bullet, \text{assume} \bullet, \bullet \rangle \implies \sigma_a, act \sim \sigma'_b \land \sigma_b \bowtie \sigma'_b \land \sigma_a, act \bowtie \sigma'_b.
\]

B.1.6 Types

Definition B.22. The initial type environment for $P$ is $\Gamma_{\text{init}}$. For $P^{[r]}$ it is $\Gamma_{\text{init}}^{[r]}$. Formally:
\[
\Gamma_{\text{init}} := \{ x : \emptyset \mid x \in PVar \cup AVar \}
\]
\[
\Gamma_{\text{init}}^{[r]} := \{ p : \emptyset \mid p \in PVar \cap \text{shared} \} \cup \{ p_t : \emptyset \mid p \in PVar \setminus \text{shared} \}
\]
\[
\cup \{ r_t : \emptyset \mid r \in AVar \}
\]

Definition B.23. A type $T$ is valid, denoted by $\text{isValid}(T)$, if:
\[
\text{isValid}(T) : \iff T \cap \{ S, A, L \} \neq \emptyset.
\]
Definition B.24. A computation \( \tau \) induces a straight-line program \( stmt(\tau, t) \) for thread \( t \) by:

\[
\begin{align*}
stmt(\epsilon, t) & := \text{skip} \\
stmt(\tau.\text{act}, t) & := stmt(\tau, t); \text{com} & & \text{if } \text{act} = \langle t, \text{com}, \text{up} \rangle \\
stmt(\tau.\text{act}, t) & := stmt(\tau, t) & & \text{if } \text{act} = \langle t', \text{com}, \text{up} \rangle \land t \neq t' \\
\end{align*}
\]

\[\square\]

Definition B.25. A pointer \( p \) has no valid alias in a computation \( \tau \), denoted by \( \text{noalias}_c(\tau) \), if the following holds: \( \text{seg} = m_\tau(p) \notin m_\tau(\text{valid}_c \setminus \{ p \}) \).

\[\square\]

Definition B.26. Let \( \tau \in \mathcal{O}[P]_{\text{Addr}} \). Let \( \text{inv}(\tau) \) have the prenex normal form \( \exists r_1 \ldots \exists r_n \phi \), where \( \phi \) is quantifier-free. Let \( r_n \) be the instance of angel \( r \) resulting from the last allocation in \( \tau \). The set of addresses possibly represented by angel \( r \) after computation \( \tau \) is

\[
\text{repr}_\tau(r) = \{ a \in \text{Addr} \mid \exists A_1, \ldots, A_n \subseteq \text{Addr}, a \in A_n \land (A_1, \ldots, A_n) \models \phi \}.
\]

We define \( m_\tau(r) = \text{repr}_\tau(r) \).

\[\square\]

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

\[
\text{SafeCall}(\Gamma, \text{func}(\tau)) \\
\begin{align*}
: & \iff \nexists h \exists t, a, \overline{w}. \ (\forall i. r_i \notin \text{DExp} \implies \text{reach}_{\Gamma, \overline{w}}(h) \subseteq \text{Loc}(\Gamma(r_i))) \\
& \land (\forall i. (w_i = a \lor \text{isValid}(\Gamma(r_i)) \lor r_i \in \text{DExp}) \implies a = w_i) \\
& \land \mathcal{F}_{\Omega}(h.\text{in}:\text{func}(t, \overline{w}), a) \notin \mathcal{F}_{\Omega}(h.\text{in}:\text{func}(t, \overline{w}), a).
\end{align*}
\]

\[\square\]

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

\[
\begin{align*}
\text{Loc}(\emptyset) & := \text{Loc}(\emptyset) & \text{Loc}(\mathbb{L}_t) & := L \\
\text{Loc}(\mathbb{A}) & := \{ L_2 \} \times \text{Loc}(\mathbb{SMR}) & \text{Loc}(\mathbb{S}) & := \text{SafeLoc}(\mathbb{O}) \\
\text{Loc}(\mathbb{L}) & := \{ L_2 \} \times \text{Loc}(\mathbb{SMR}) & \text{Loc}(T_1 \land T_2) & := \text{Loc}(T_1) \cap \text{Loc}(T_2).
\end{align*}
\]

where \( \text{Loc}(\mathbb{O}) \) and \( \text{Loc}(\mathbb{SMR}) \) refer to all locations of \( \mathbb{O} \) and \( \mathbb{SMR} \) respectively.

\[\square\]

Repeated Definition 8.10 (Type Transformer). The type transformer relation \( T, x, \text{com} \leadsto T' \) is defined by:

\[
T, x, \text{com} \leadsto T' : \iff \text{post}_{x, \text{com}}(\text{Loc}(T)) \subseteq \text{Loc}(T') \\
\land \text{isValid}(T') \Rightarrow \text{isValid}(T) \\
\land \{ L, A \} \land T' \subseteq \{ L, A \} \land T .
\]

Moreover, we define the following abbreviations:

\[
\begin{align*}
\Gamma, \text{com} & \leadsto \Gamma' : \iff \forall x. \ \Gamma(x), x \text{com} \leadsto \Gamma'(x) \\
\Gamma & \leadsto \Gamma' : \iff \Gamma, \text{skip} \leadsto \Gamma'.
\end{align*}
\]

\[\square\]
B.2 Compositionality

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

Lemma B.28 (Proof C.1). The simulation relation $\leq_{\text{SMR}}$ for SMR automaton $O_{\text{ERR}}$ is:

$$L_7 \leq_{\text{ERR}} L_6 \leq_{\text{ERR}} L_5 \leq_{\text{ERR}} L_4$$

□

Lemma B.29 (Proof C.2). The simulation relation $\leq_{\text{META}}$ for SMR automaton $O^k_{\text{HP}}$ is:

$$L_{12} \leq_{\text{META}} L_{11} \leq_{\text{META}} L_{10} \leq_{\text{META}} L_9 \leq_{\text{META}} L_8$$

□

Lemma B.30 (Proof C.3). The simulation relation $\leq_{\text{META}}$ for SMR automaton $O_{\text{HP}}^{0,1}$ is:

$$L_{15} \leq_{\text{META}} L_{16} \leq_{\text{META}} L_{17} \leq_{\text{META}} L_{18} \leq_{\text{META}} L_{19} \leq_{\text{META}} L_{20} \leq_{\text{META}} L_{21} \leq_{\text{META}} L_{22} \leq_{\text{META}} L_{23} \leq_{\text{META}} L_{24} \leq_{\text{META}} L_{25} \leq_{\text{META}} L_{26} \leq_{\text{META}} L_{27} \leq_{\text{META}} L_{28} \leq_{\text{META}} L_{29} \leq_{\text{META}} L_{30}$$

□

Repeated Proposition 5.3 (Proof C.4). If $I \leq_{\text{META}} I'$, then $S((I, \varphi)) \subseteq S((I', \varphi))$ for all $\varphi$. □

Repeated Theorem A.2 (Proof C.5). Let $\tau \in \llbracket P(R) \rrbracket_{\text{META}}$ be free from separation violations. Then, there is some $\sigma \in \llbracket MGC(R) \rrbracket_{\text{META}}$ such that (i) $\text{ctrl}^R(\tau) \subseteq \text{ctrl}^R(\sigma)$, (ii) $m_\tau^R = m_\sigma^R$, (iii) $m_\tau^{\downarrow \text{Var}} = m_\sigma^{\downarrow \text{Var}}$, (iv) $H(\tau) = H(\sigma)$, (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 $\tau \in \llbracket P(R) \rrbracket_{\text{META}}$ which is free from separation violations. If $R \not\models O$, then there is $\sigma \in \llbracket P \rrbracket_{\text{META}}$ with (i) $\text{ctrl}^P(\tau) \subseteq \text{ctrl}^P(\sigma)$, (ii) $m_\tau^P = m_\sigma^P$, (iii) $H(\tau) = H(\sigma)$, (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 $R \not\models O$ and good($O \llbracket P \rrbracket_{\text{META}}$), then good($\llbracket P(R) \rrbracket_{\text{META}}$). □

Repeated Theorem 5.11 (Proof C.9). If $R \not\models O$, then $\llbracket P(R) \rrbracket_{\text{META}}$ is DRF if $O \llbracket P \rrbracket_{\text{META}}$ 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 $\tau \in O\llbracket P \rrbracket_{Adr}^{Adr}$ with $m_\tau(p) \in owned_\tau(t)$. Then, we have: $p \in valid_\tau$ implies $p \in local_\tau$.

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 $O\llbracket P \rrbracket_{Adr}^{Adr}$ 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 $m_{\tau_1}(valid_{\tau_1}) \subseteq m_{\tau_2}(valid_{\tau_2})$ and $\tau_1 \ll A \tau_2 \ll B \tau_3$ holds, then we have $\tau_1 \ll_{A\cap B} \tau_3$.

Lemma B.34 (Proof C.14). If $adr(m_{\tau_1}|valid_{\tau_1}) \subseteq adr(m_{\tau_2}|valid_{\tau_2})$ and $\tau_1 \ll \tau_2 \ll \tau_3$ holds, then we have $\tau_1 \ll \tau_3$.

Lemma B.35 (Proof C.15). We have $valid_{\tau} \subseteq dom(m_{\tau})$.

Lemma B.36 (Proof C.16). We have $valid_{\tau} \subseteq dom(m_{\tau})$.

Lemma B.37 (Proof C.17). If $\tau \sim \sigma$, then $valid_{\tau} = valid_{\sigma}$.

Lemma B.38 (Proof C.18). If $\tau \sim \sigma$, then $adr(m_{\tau}|valid_{\tau}) = adr(m_{\sigma}|valid_{\sigma})$.

Lemma B.39 (Proof C.19). If $\tau \sim \sigma$, then $next-com(\tau) = next-com(\sigma)$.

Lemma B.40 (Proof C.20). If $\tau.(t, com, up) \in O\llbracket P \rrbracket_{Adr}^{Adr}$, $t \neq \bot$, then $com \in next-com(\tau)$.

Lemma B.41 (Proof C.21). If $frees_{h_2} \subset \{ a \}$, then we have:

$$h_3 \in F\llbracket O \rrbracket(h_1, h_2, a) \iff h_2, h_3 \in F\llbracket O \rrbracket(h_1, a)$$.

Lemma B.42 (Proof C.22). If $a \in fresh_{h_2}$, then $a \notin range(m_{\tau})$.

Lemma B.43 (Proof C.23). If $a \in fresh_{h_2}$, then $a \notin m_{\tau}(valid_{\tau})$ and $a.next \notin valid_{\tau}$.

Lemma B.44 (Proof C.24). We have $fresh_{\tau} \cap freed_{\tau} = \emptyset$ and $fresh_{\tau} \cap retired_{\tau} = \emptyset$.
The next set of lemmas provides insights about computations free from unsafe accesses.

**Lemma B.45** (Proof C.25). If $\varphi = \{ z_a \mapsto a \}$, then we have $(L_2, \varphi) \xrightarrow{H(t)} (L_3, \varphi)$ iff $a \in \text{retired}_r$, as well as $(L_2, \varphi) \xrightarrow{H(t)} (L_2, \varphi)$ iff $a \notin \text{retired}_r$. □

**Lemma B.46** (Proof C.26). If $t, \text{free}(a), \text{up} \in \mathcal{O}[\text{P}]^{Adr}_{Adr}$, then $a \in \text{retired}_r$. □

**Lemma B.47** (Proof C.27). Let $\tau, \text{act} \in \mathcal{O}[\text{P}]^{Adr}_{Adr}$ with $\text{act} = (t, \text{com}, \text{up})$. We have:

- $m_{\tau, \text{act}}(\text{valid}_{\tau, \text{act}}) \subseteq m_{\tau}(\text{valid}_{\tau}) \cup \{ m_{\tau, \text{act}}(p) \}$ if $\text{com} \equiv p := \text{malloc}$
- $m_{\tau, \text{act}}(\text{valid}_{\tau, \text{act}}) \subseteq m_{\tau}(\text{valid}_{\tau}) \setminus \{ a \}$ if $\text{com} \equiv \text{free}(a)$
- $m_{\tau, \text{act}}(\text{valid}_{\tau, \text{act}}) \subseteq m_{\tau}(\text{valid}_{\tau})$, if $\text{com} \equiv \text{env}(a)$
- $m_{\tau, \text{act}}(\text{valid}_{\tau, \text{act}}) \subseteq m_{\tau}(\text{valid}_{\tau})$ otherwise

and

- $\text{adr}(m_{\tau, \text{act}}|\text{valid}_{\tau, \text{act}}) \subseteq \text{adr}(m_{\tau}|\text{valid}_{\tau}) \cup \{ m_{\tau, \text{act}}(p) \}$ if $\text{com} \equiv p := \text{malloc}$
- $\text{adr}(m_{\tau, \text{act}}|\text{valid}_{\tau, \text{act}}) \subseteq \text{adr}(m_{\tau}|\text{valid}_{\tau}) \setminus \{ a \}$ if $\text{com} \equiv \text{free}(a)$
- $\text{adr}(m_{\tau, \text{act}}|\text{valid}_{\tau, \text{act}}) \subseteq \text{adr}(m_{\tau}|\text{valid}_{\tau})$, if $\text{com} \equiv \text{env}(a)$
- $\text{adr}(m_{\tau, \text{act}}|\text{valid}_{\tau, \text{act}}) = \text{adr}(m_{\tau}|\text{valid}_{\tau})$ otherwise. □

The next set of lemmas provides insights about computations free from unsafe accesses.

**Lemma B.48** (Proof C.28). If $\tau$ UAF and $a \in \text{freed}_r$, so $a \notin m_{\tau}(\text{valid}_{\tau})$ and $a.\text{next} \notin \text{valid}_{\tau}$. □

**Lemma B.49** (Proof C.29). If $\tau$ UAF, $\text{pexp} \in \text{VExp}(\tau)$, and $m_{\tau}(\text{pexp}) = \text{seg}$, so $\text{pexp} \in \text{valid}_{\tau}$. □

**Lemma B.50** (Proof C.30). If $\tau$ UAF, then $\text{VExp}(\tau) \subseteq \text{dom}(m_{\tau})$. □

**Lemma B.51** (Proof C.31). If $\tau \in \mathcal{O}[\text{P}]^{Adr}_{Adr}$ UAF, then $m_{\tau}(\text{VExp}(\tau) \setminus \text{valid}_{\tau}) \subseteq \text{freed}_r$. □

**Lemma B.52** (Proof C.32). If $\tau \in \mathcal{O}[\text{P}]^{Adr}_{Adr}$ UAF, then $m_{\tau}(\text{VExp}(\tau) \setminus \text{valid}_{\tau}) \subseteq \text{freed}_r$. □

**Lemma B.53** (Proof C.33). If $\tau \in \mathcal{O}[\text{P}]^{Adr}_{Adr}$ UAF, then we have:

$$\text{adr}(m_{\tau}|\text{valid}_{\tau}) \cap m_{\tau}(\text{VExp}(\tau) \setminus \text{valid}_{\tau}) \subseteq A.$$ □

**Lemma B.54** (Proof C.34). If $\tau \in \mathcal{O}[\text{P}]^{Adr}_{Adr}$ UAF, then $m_{\tau}(\text{CVar}) \cap m_{\tau}(\text{PVar} \setminus \text{valid}_{\tau}) = \emptyset$. □

Consider an SMR automaton $\mathcal{O}$. By Assumption 5.6, it is of the form $\mathcal{O} = \mathcal{O}_{\text{base}} \times \mathcal{O}_{\text{SMR}}$. Recall that elision support for $\mathcal{O}$, Definition 7.14, provides properties for $\mathcal{O}_{\text{SMR}}$ only. We generalize them to full $\mathcal{O}$.

**Lemma B.55** (Proof C.35). Let $\mathcal{O}$ support elision. For all $\tau \in \mathcal{O}[\text{P}]^{Adr}_{Adr}$ and $a, b, c \in \text{Adr}$ 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 $\tau \in \mathcal{O}[\text{P}]^{Adr}_{Adr}$ and $a, b \in \text{Adr}$ such that $a \neq b$ and $\mathcal{H}(\tau) = h$ and $h.\text{free}(a) \in \mathcal{S}(\mathcal{O})$, we have $\mathcal{F}_\mathcal{O}(h.\text{free}(a), b) = \mathcal{F}_\mathcal{O}(h, b)$. □
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.57 (Proof C.37). Let \( \mathcal{O} \) supports elision. For all \( \tau, \sigma \in \mathcal{O}[P]_{Adr}^{A} \) and \( a, b \in Adr \) such that \( \mathcal{F}_\mathcal{O}(\tau, a) \subseteq \mathcal{F}_\mathcal{O}(\sigma, a) \) and \( b \notin \text{retired}_\tau \) and \( b \in \text{fresh}_\sigma \), we 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 \( \text{swap}_{adr} \) is an address mapping, so is \( \text{swap}_{adr}^{-1} \). □

Lemma B.59 (Proof C.39). If \( \text{swap}_{\text{hist}}(h_1) = \text{swap}_{\text{hist}}(h_2) \), then \( h_1 = h_2 \). □

Lemma B.60 (Proof C.40). We have \( \text{swap}_{\text{hist}}(h) \in \text{swap}_{\text{hist}}(H) \iff h \in H \). □

Lemma B.61 (Proof C.41). For every \( \ominus \in \{ \setminus, \cup, \cap \} \) we have:

\[
\text{swap}_{adr}(A_1, A_2) = \text{swap}_{\text{exp}}(B_1, B_2)
\]

\[
\text{swap}_{\text{hist}}(C_1, C_2) = \text{swap}_{\text{hist}}(C_1, C_2)
\]

Lemma B.62 (Proof C.42). For all \( h \) we have \( \text{swap}_{\text{hist}}(\text{swap}_{\text{hist}}(h)) = h \). □

Lemma B.63 (Proof C.43). For all \( h \) we have \( h \in S(\mathcal{O}) \iff \text{swap}_{\text{hist}}(h) \in S(\mathcal{O}) \). □

Lemma B.64 (Proof C.44). We have \( \text{swap}_{\text{hist}}(\mathcal{F}_\mathcal{O}(h, a)) = \mathcal{F}_\mathcal{O}(\text{swap}_{\text{hist}}(h), \text{swap}_{adr}(a)) \) for all addresses \( a \in Adr \). □

Theorem B.65 (Proof C.45). For all \( \tau \in \mathcal{O}[P]_{Adr}^{A} \) and all address mappings \( \text{swap}_{adr} \), there is some \( \sigma \in \mathcal{O}[P]_{Adr}^{\text{swap}_{adr}(A)} \) such that:

\[
m_\sigma \circ \text{swap}_{\text{exp}} = \text{swap}_{adr} \circ m_\tau \quad \mathcal{H}(\sigma) = \text{swap}_{\text{hist}}(\mathcal{H}(\tau)) \quad \text{freed}_\sigma = \text{swap}_{adr}(\text{freed}_\tau)
\]

\[
\text{valid}_{\sigma} = \text{swap}_{\text{exp}}(\text{valid}_{\tau}) \quad \text{ctrl}(\sigma) = \text{ctrl}(\tau) \quad \text{fresh}_{\sigma} = \text{swap}_{adr}(\text{fresh}_{\tau})
\]

Lemma B.66 (Proof C.46). Let \( \mathcal{O} \) support elision. If \( \tau \in \mathcal{O}[P]_{Adr}^{A} \) and \( a \notin \text{adr}(m_\tau |_{\text{valid}_{\tau}}) \cup A \), then there is \( \sigma \in \mathcal{O}[P]_{Adr}^{\text{swap}_{adr}(A)} \) such that (i) \( \tau \prec \sigma \), (ii) \( \tau \in A \), (iii) \( \tau \prec \sigma \), (iv) \( \text{retired}_\tau \in \text{retired}_\sigma \cup \{ a \} \), (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 \( \text{pexp}, \text{xexp} \in V\text{Exp}(\tau) \), then \( m_\tau(\text{exp}) \neq m_\tau(\text{exp}') \) implies \( m_\sigma(\text{exp}) \neq m_\sigma(\text{exp}') \). □
Lemma B.67 (Ĥ Proof C.47). Consider \( r \cdot \mathrm{act} \in \mathcal{O} [ P ] \uparrow_{\mathrm{Adr}} \) and \( \sigma \in \mathcal{O} [ P ] \uparrow_{\mathrm{Adr}} \) with \( r \sim \sigma, r \preceq_A \sigma \), and \( r \preceq_A \sigma \). Let \( \text{act} = \langle t, \text{com}, \text{up} \rangle \). If one of the following cases applies:

(i) \( \text{com} \equiv x := y \) with \( x, y \in \text{Var} \),
(ii) \( \text{com} \equiv x := \text{q.sel} \) with \( x \in \text{Var} \) and \( q \in \text{valid}_r \), or
(iii) \( \text{com} \equiv p . \text{sel} := y \) with \( y \in \text{Var} \) and \( p \in \text{valid}_r \),

then, there is \( \text{act}' = \langle t, \text{com}, \text{up}' \rangle \) with \( \sigma . \text{act}' \in \mathcal{O} [ P ] \uparrow_{\mathrm{Adr}} \) as well as \( r . \text{act} \sim \sigma . \text{act}' \), \( r . \text{act} \preceq_A \sigma . \text{act}' \), and \( r . \text{act} \prec \sigma . \text{act}' \).

\qed

Lemma B.68 (Ĥ Proof C.48). Consider \( r . \text{act} \in \mathcal{O} [ P ] \uparrow_{\mathrm{Adr}} \) and \( \sigma \in \mathcal{O} [ P ] \uparrow_{\mathrm{Adr}} \) with \( r \sim \sigma, r \preceq_A \sigma \), and \( r \preceq_A \sigma \). Let \( \text{act} = \langle t, \text{com}, \text{up} \rangle \). If \( \sigma . \text{act} \in \mathcal{O} [ P ] \uparrow_{\mathrm{Adr}} \) and one of the following cases applies:

(i) \( \text{com} \equiv \text{in} : \text{func}(r_1, \ldots, r_n) \) with \( m_r(r_i) = m_o(r_i) \) for all \( 1 \leq i \leq n \),
(ii) \( \text{com} \equiv \text{re} : \text{func}, \)
(iii) \( \text{com} \equiv \text{assume} \bullet, \)
(iv) \( \text{com} \equiv \text{free}(a) \) with \( \mathcal{F}_O ( \text{h.free}(a), b ) = \mathcal{F}_O ( h, b ) \) for all \( b \neq a \) and \( h \in \{ \mathcal{H}(r), \mathcal{H}(\sigma) \} \),
(v) \( \text{com} \equiv p := \text{malloc} \) with \( m_{\text{r,act}}(p) = a \) and \( a \notin A \implies \mathcal{F}_O ( \tau, a ) \subseteq \mathcal{F}_O ( \sigma, a ) \), or
(vi) \( \text{com} \equiv \text{env}(a). \)

then \( r . \text{act} \sim \sigma . \text{act}, r . \text{act} \preceq_A \sigma . \text{act} \), and \( r . \text{act} \prec \sigma . \text{act} \).

\qed

Lemma B.69 (Ĥ Proof C.49). Let \( r = r_1 . \langle \bot, \text{env}(a), \text{up} \rangle . r_2 \in \mathcal{O} [ P ]_X \uparrow_{\text{UAF}} \) and \( \sigma = r_1 . r_2 \). Then:

(i) \( \sigma \in \mathcal{O} [ P ]_X \uparrow_{\text{UAF}} \), (ii) \( \text{ctrl}(r) = \text{ctrl}(\sigma) \), (iii) \( \mathcal{H}(r) = \mathcal{H}(\sigma) \), (iv) \( \text{fresh}_r = \text{fresh}_\sigma \), (v) \( \text{freed}_r = \text{freed}_\sigma \),
(vi) \( \text{retired}_r = \text{retired}_\sigma \), and (vii) \( \text{(exp} \cap \text{Adr}) \cap (\text{fresh}_r \cup \text{freed}_r) \neq \emptyset \) if \( m_r(\text{exp}) \neq m_o(\text{exp}) \).

\qed

Lemma B.70 (Ĥ Proof C.50). Let \( \mathcal{O} \) supports elision. If \( r \in \mathcal{O} [ P ]_X \uparrow_{\text{UAF}} \) and \( a \in \text{retired}_r \cap \text{freed}_r \), then \( \mathcal{O} [ P ]_X \) contains a double retire.

\qed

We turn to the reduction result from Chapter 7 which relies on pointer race freedom to verify \( \mathcal{O} [ P ]_{\overline{\text{Adr}}} \), under the much smaller semantics \( \mathcal{O} [ P ]_{\overline{\text{Adr}}} \).

\begin{repeatedtheorem}{20}{Ĥ Proof C.51}{\text{Repeated Theorem}}
\begin{enumerate}
\item Let \( \mathcal{O} \) support elision. Let \( \mathcal{O} [ P ]_{\overline{\text{Adr}}} \) be \text{PRF}, \text{DFR}, and free from harmful ABAs. Then, for all \( r \in \mathcal{O} [ P ]_{\overline{\text{Adr}}} \) and all \( a \in \text{Adr} \) there is \( \sigma \in \mathcal{O} [ P ]_{\overline{\text{Adr}}} \) such that \( r \sim \sigma, r \preceq \sigma, r \preceq_a \sigma \).
\end{enumerate}
\end{repeatedtheorem}

\begin{repeatedtheorem}{21}{Ĥ Proof C.52}{\text{Repeated Theorem}}
\begin{enumerate}
\item Let \( \mathcal{O} \) support elision. Let \( \mathcal{O} [ P ]_{\overline{\text{Adr}}} \) be \text{PRF}, \text{DFR}, and free from harmful ABAs. Then, \( \text{good}(\mathcal{O} [ P ]_{\overline{\text{Adr}}}) \iff \text{good}(\mathcal{O} [ P ]_{\overline{\text{Adr}}}) \).
\end{enumerate}
\end{repeatedtheorem}

\begin{repeatedtheorem}{22}{Ĥ Proof C.53}{\text{Repeated Theorem}}
\begin{enumerate}
\item Let \( \mathcal{O} \) support elision. Let \( \mathcal{O} [ P ]_{\overline{\text{Adr}}} \) be \text{PRF}, \text{DFR}, and free from harmful ABAs. Then, \( \mathcal{O} [ P ]_{\overline{\text{Adr}}} \) is \text{DFR}.
\end{enumerate}
\end{repeatedtheorem}

\begin{repeatedproposition}{15}{Ĥ Proof C.54}{\text{Repeated Proposition}}
The SMR automata \( \mathcal{O}_{\text{Base}} \times \mathcal{O}_{\text{EBR}} \) and \( \mathcal{O}_{\text{Base}} \times \mathcal{O}_{\text{HP}} \) and \( \mathcal{O}_{\text{Base}} \times \mathcal{O}_{\text{HP}} \times \mathcal{O}_{\text{HP}} \) support elision.
\end{repeatedproposition}

\begin{repeatedproposition}{23}{Ĥ Proof C.55}{\text{Repeated Proposition}}
If a call is racy wrt. \( \mathcal{O}_{\text{Base}} \times \mathcal{O}_{\text{EBR}} \) or \( \mathcal{O}_{\text{Base}} \times \mathcal{O}_{\text{HP}} \) or \( \mathcal{O}_{\text{Base}} \times \mathcal{O}_{\text{HP}} \times \mathcal{O}_{\text{HP}} \), then it is a call to retire with an invalid pointer as its argument.
\end{repeatedproposition}
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 \( [P]_O^{\infty} \). 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 \( \mathcal{O}[P]_{Adr}^{Adr} \) be MPRF and DRF. Then: for all \( \tau \in \mathcal{O}[P]_{Adr}^{Adr} \) there is \( \sigma \in \mathcal{O}[P]_O^{\infty} \) with: (i) \( \tau \sim \sigma \), (ii) \( \tau < \sigma \), (iii) \( \mathcal{F}_O(\tau, a) \subseteq \mathcal{F}_O(\sigma, a) \) for all \( a \in \text{fresh}_\sigma \), (iv) \( \text{retired}_\tau \subseteq \text{retired}_\sigma \), and (v) \( m_\tau(pexp) \neq m_\sigma(pexp) \) for all \( pexp, qexp \in \text{VExp}(\tau) \). Moreover, we have (vi) \( \tau \text{ UAF} \), (vii) \( \text{freed}_\tau \cap \text{adr}(m_\tau|\text{valid}_\tau) = \emptyset \), and (viii) \( \text{freed}_\tau \cap \text{retired}_\tau = \emptyset \). □

**Theorem B.72** (Proof C.57). For all \( \tau \in \mathcal{O}[P]_{Adr}^{Adr} \) UAF there is \( \sigma \in \mathcal{O}[P]_O^{\infty} \) 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 \( \text{inv}(\tau) \), and (vi) \( m_\tau(pexp) \neq m_\sigma(pexp) \) implies \( (\text{exp} \cap \text{Adr}) \cap (\text{fresh}_\tau \cup \text{freed}_\tau) \neq \emptyset \). □

**Repeated Theorem A.10** (Proof C.58). If \( \mathcal{O} \) supports elision and \( \mathcal{O}[P]_{Adr}^{Adr} \) is MPRF and DRF, then we have \( \text{good}(\mathcal{O}[P]_{Adr}^{Adr}) \iff \text{good}(\mathcal{O}[P]_O^{\infty}) \) and \( \mathcal{O}[P]_{Adr}^{Adr} \) is DRF. □

**Repeated Theorem 8.5** (Proof C.59). Let \( \mathcal{O} \) support elision and let \( \mathcal{O}[P]_{Adr}^{Adr} \) be SPRF and DRF. For all \( \tau \in \mathcal{O}[P]_{Adr}^{Adr} \) there is \( \sigma \in \mathcal{O}[P]_O^{\infty} \) with \( \tau \sim \sigma \), \( \tau < \sigma \), and \( \text{retired}_\tau \subseteq \text{retired}_\sigma \). □

**Repeated Theorem 8.6** (Proof C.60). If \( \tau \in \mathcal{O}[P]_{Adr}^{Adr} \) is SPRF, then there is some \( \sigma \in \mathcal{O}[P]_O^{\infty} \) such that \( \text{ctrl}(\tau) = \text{ctrl}(\sigma) \), \( m_\tau|\text{valid}_\tau = m_\sigma|\text{valid}_\sigma \), and \( \text{inv}(\sigma) \implies \text{inv}(\tau) \). □

**Repeated Theorem 8.7** (Proof C.61). If \( \mathcal{O} \) supports elision and \( \mathcal{O}[P]_{Adr}^{Adr} \) is SPRF and DRF, then we have \( \text{good}(\mathcal{O}[P]_{Adr}^{Adr}) \iff \text{good}(\mathcal{O}[P]_O^{\infty}) \) and \( \mathcal{O}[P]_{Adr}^{Adr} \) is DRF. □

### 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 \( z \), resp. \( z_a \) tracking a thread resp. an address. □

**Repeated Assumption 8.11.** Programs adhere to the following restricted syntax:

\[
\begin{align*}
\text{stmt} & ::= \text{stmt} \cdot \text{stmt} \mid \text{stmt} \oplus \text{stmt} \mid \text{stmt}^* \mid \text{beginAtomic}; \text{stmt}; \text{endAtomic} \\
& \mid \text{beginAtomic}; \text{com}; \text{endAtomic}.
\end{align*}
\]

**Repeated Assumption A.9.** There are no constant violations in \( \mathcal{O}[P]_{Adr}^{Adr} \). □

**Lemma B.73** (Proof C.62). If \( \Gamma_1 \leadsto \Gamma_2 \) and \( \Gamma_2 \leadsto \Gamma_3 \), then \( \Gamma_1 \leadsto \Gamma_3 \). □

**Lemma B.74** (Proof C.63). If \( \Gamma \vdash \{ \Gamma_1 \} \text{ stmt } \{ \Gamma_2 \} \) and \( \text{stmt} \xrightarrow{\text{com}} \text{stmt}' \), then there is an intermediate environment \( \Gamma \) such that \( \Gamma \vdash \{ \Gamma_1 \} \text{ com } \{ \Gamma \} \) and \( \Gamma \vdash \{ \Gamma \} \text{ stmt}' \{ \Gamma_2 \} \). □
Lemma B.75 (Φ Proof C.64). Let \( \vdash \{ \Gamma_{init} \} \ P \ \{ \Gamma \} \). Consider \( (pc_{init}, \epsilon) \rightarrow^* (pc, r) \) and some thread \( t \). Then there is \( \Gamma_1, \Gamma_2 \) with \( \vdash \{ \Gamma_{init} \} \ stmt(t, t) \ \{ \Gamma_1 \} \) and \( \vdash \{ \Gamma_1 \} \ pc(t) \ \{ \Gamma_2 \} \). □

Lemma B.76 (Φ Proof C.65). Consider some \( \tau \cdot act \in O[P]_{Adr}^\lor \) and \( t \neq thrd(act) \neq \bot \). Then, we have either \( stmt(t, t) = \text{skip} \) or \( stmt(t, t) = \text{stmt; endAtomic} \). □

Lemma B.77 (Φ Proof C.66). Let \( \tau \cdot act \in O[P]_{Adr}^\lor \) and \( t \neq thrd(act) \neq \bot \) and \( x \in PVar \cup AVar \). Assume \( \vdash \{ \Gamma_{init} \} \ stmt(t, t) \ \{ \Gamma \} \). Then, \( \Gamma \notin \Gamma(x) \) and \( x \notin \text{local}_t \implies \Gamma(x) \cap (\text{local}_s, \text{S}) = \emptyset \). □

Lemma B.78 (Φ Proof C.67). Let \( \tau \cdot act \in O[P]_{Adr}^\lor \) and \( t \neq thrd(act) \neq \bot \) and \( p \in PVar \cap \text{local}_t \). Then, \( p \in \text{valid}_t \), implies \( p \in \text{valid}_{\tau, act} \). Moreover, \( \text{noalias}_t(p) \) implies \( \text{noalias}_{\tau, act}(p) \). □

Lemma B.79 (Φ Proof C.68). Consider \( \tau \cdot act \in O[P]_{Adr}^\lor \) UAF such that \( act = \{ t, \oplus \text{inv} p = q, u \} \) and \( \text{inv}(\tau \cdot act) \). Then, \( \{ p, q \} \cap \text{valid}_t \neq \emptyset \) implies \( \{ p, q \} \subseteq \text{valid}_t \). □

Lemma B.80 (Φ Proof C.69). Let \( \tau \cdot act \in O[P]_{Adr}^\lor \) UAF such that \( act = \{ t, \oplus \text{inv active}(p, u) \} \) and \( \text{inv}(\tau \cdot act) \). Then, \( p \in \text{valid}_{\tau, act} \) and \( \text{reach}_{\text{act}, a}(\tau \cdot act) \subseteq \text{Loc}(\Gamma) \) for \( a = \text{m}_{\tau, act}(p) \). □

Lemma B.81 (Φ Proof C.70). Consider \( \tau \cdot act \in O[P]_{Adr}^\lor \) such that \( act = \{ t, \oplus \text{inv active}(r, u) \} \) and \( \text{inv}(\tau \cdot act) \). Then, we have for all \( a \in \text{repr}_{\tau, act}(r) \):

\[
\text{repr}_{\tau, act}(r) \cap \text{free}_{\tau, act} = \emptyset \quad \text{and} \quad \text{reach}_{\text{act}, a}(\tau \cdot act) \subseteq \text{Loc}(\Gamma) .
\]

□

Lemma B.82 (Φ Proof C.71). Consider \( r \in O[P]_{Adr}^\lor \). Let \( \Gamma, \Gamma' \) such that \( \Gamma \leadsto \Gamma' \). Let \( t \) be some thread. Let \( p \in PVar \) and \( a = \text{m}_{\tau, act}(p) \). Let \( r \in AVar \) and \( b \in \text{repr}_{\tau, act}(r) \). Then:

\[
\text{isValid}(\Gamma(p)) \implies p \in \text{valid}_r \quad \text{implies} \quad \text{isValid}(\Gamma'(p)) \implies p \in \text{valid}_r \\
\text{isValid}(\Gamma(r)) \implies b \notin \text{free}_r \quad \text{implies} \quad \text{isValid}(\Gamma'(r)) \implies b \notin \text{free}_r \\
L \in \Gamma(p) \implies \text{noalias}_t(p) \quad \text{implies} \quad L \in \Gamma'(p) \implies \text{noalias}_t(p) \\
\text{reach}_{\text{act}, a}(\tau) \subseteq \text{Loc}(\Gamma(p)) \quad \text{implies} \quad \text{reach}_{\text{act}, a}(\tau) \subseteq \text{Loc}(\Gamma'(p)) \\
\text{reach}_{\text{act}, a}(\tau) \subseteq \text{Loc}(\Gamma(r)) \quad \text{implies} \quad \text{reach}_{\text{act}, a}(\tau) \subseteq \text{Loc}(\Gamma'(r))
\]

□

Theorem B.83 (Φ Proof C.72). Consider thread \( t \), environment \( \Gamma \), and \( \tau \in O[P]_{Adr}^\lor \). UAF such that \( \text{inv}(\tau) \) and \( \vdash \{ \Gamma_{init} \} \ stmt(t, t) \ \{ \Gamma \} \). For all \( p \in PVar \), we have \( \text{reach}_{\text{act}, m_{\tau, act}(p)}(\tau) \subseteq \text{Loc}(\Gamma(p)) \) and \( \text{isValid}(\Gamma(p)) \implies p \in \text{valid}_r \). □

Corollary B.84 (Φ Proof C.73). Consider some thread \( t \), type environment \( \Gamma \), and \( \tau \in O[P]_{Adr}^\lor \). such that \( \text{inv}(\tau) \) and \( \vdash \{ \Gamma_{init} \} \ stmt(t, t) \ \{ \Gamma \} \). Then, \( \tau \) is PRF. □

Theorem B.85 (Φ Proof C.74). If \( \vdash P \) and \( \text{inv}(P) \subseteq O[P]_{Adr}^\lor \), then \( O[P]_{Adr}^\lor \) is MPRF. □

Theorem B.86 (Φ Proof C.75). If \( \vdash P \) and \( \text{inv}(P) \subseteq O[P]_{Adr}^\lor \), then \( O[P]_{Adr}^\lor \) is DRF. □

Theorem B.87 (Φ Proof C.76). If \( \vdash P \) and \( \text{inv}(P) \subseteq O[P]_{Adr}^\lor \), then \( \text{inv}(O[P]_{Adr}^\lor) \). □

Repeated Theorem 8.14 (Φ Proof C.77). For all threads \( t \) and all \( \tau \in O[P]_{Adr}^\lor \) with \( \text{inv}(\tau) \) we have the following: \( \vdash \{ \Gamma_{init} \} \ stmt(t, t) \ \{ \Gamma \} \) implies \( \vdash \{ \Gamma_{init} \} \ stmt(t, t) \ \{ \Gamma \} \). □

Repeated Theorem 8.15 (Φ Proof C.78). If \( \vdash P \) and \( \text{inv}(O[P]_{Adr}^\lor) \), then \( O[P]_{Adr}^\lor \) is SPRF. □
Repeated Theorem 8.16 (Φ Proof C.79). If \( \vdash P \) and \( \text{inv}(O\llbracket P \rrbracket_{\text{Addr}}) \), then \( O\llbracket P \rrbracket_{\text{Addr}} \) is DRF.  

Repeated Theorem 8.21 (Φ Proof C.80). If \( \vdash P \) and \( \text{inv}(\llbracket P \rrbracket_{\text{Addr}}) \), then \( O\llbracket P \rrbracket_{\text{Addr}} \) SPRF and we have \( \text{inv}(O\llbracket P \rrbracket_{\text{Addr}}) \).  

Repeated Theorem 8.22 (Φ Proof C.81). We have \( \text{inv}(\llbracket P \rrbracket_{\text{Addr}}) \) iff \( \text{safe}(\llbracket \text{inst}(P) \rrbracket_{\text{Addr}}) \). The instrumentation is linear in size.  

Repeated Theorem 8.24 (Φ Proof C.82). Type inference \( \vdash P \) runs in time \( \mathcal{O}(|P|^2) \).  

Repeated Theorem 8.26 (Φ Proof C.83). For \( \Phi(\Gamma_{\text{init}}, P, X) \) we have \( \text{lsol}(X) = \bigcap_{\vdash (\Gamma_{\text{init}}) \Gamma} \Gamma \). Hence \( \text{lsol}(X) \neq \top \) if and only if \( \vdash P \).  

Repeated Theorem A.11 (Φ Proof C.84). If \( \vdash P \) and \( \text{inv}(\llbracket P \rrbracket_{\text{Addr}}) \), then \( O\llbracket P \rrbracket_{\text{Addr}} \) is MPRF and DRF and we have \( \text{inv}(O\llbracket P \rrbracket_{\text{Addr}}) \).  

Repeated Proposition A.12 (Φ Proof C.85). If \( r \in O\llbracket P \rrbracket_{\text{Addr}} \), then \( m_r(\text{CVar}) \subseteq \text{active}(r) \).
We give the proofs for the meta theory from Appendix B.

C.1 Compositionality

Proof C.1 (Lemma B.28). By definition of $\preceq_*$ and $O_{EBR}$. 

Proof C.2 (Lemma B.29). By definition of $\preceq_*$ and $O^{k}_{HP}$. 

Proof C.3 (Lemma B.30). By definition of $\preceq_*$ and $O^{0.1}_{HP}$. 

Proof C.4 (Proposition 5.3). Consider some $h \in S((l_1, \varphi))$. We show that $h \in S((l_2, \varphi))$ 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, $h \in S((l_2, \varphi))$ follows as required. For the induction step, consider $f(\overline{v}).h \in S((l_1, \varphi))$. By Assumption 5.2, there are steps $(l_1, \varphi) \xrightarrow{f(\overline{v})} (l_1', \varphi)$ and $(l_2, \varphi) \xrightarrow{f(\overline{v})} (l_2', \varphi)$. The former step is due to a transition $l_1 \xrightarrow{f(\overline{v})} \overline{v}$ such that $\varphi(\overline{v} \rightarrow \overline{v})$ evaluates to true. Similarly, the latter step is due to a transition $l_2 \xrightarrow{f(\overline{v})} \overline{v}$ such that $\varphi(\overline{v} \rightarrow \overline{v})$ evaluates to true. This means $\varphi$ is a model for $g$ and $g'$. That is, $g \land g'$ is satisfiable. Then, $l_1 \preceq_{O} l_2$ yields $l_1' \preceq_{O} l_2'$. Note that we have $h \in S((l_1', \varphi))$ by definition. By induction, we get $h \in S((l_2', \varphi))$. Because SMR automata are deterministic by Assumption 5.2, we conclude $f(\overline{v}).h \in S((l_2, \varphi))$ as required. 

Proof C.5 (Theorem A.2). We proceed by induction over the structure of $\tau$. 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:

$$ (pc_1 \circ pc_3, \tau) \xrightarrow{z_{\text{ctrl}}(\tau)} (pc_1 \circ pc_3, \tau.\text{act}) \quad \text{with} \quad pc_1 \circ pc_3 \in \text{ctrl}(\tau). \quad (1) $$

By definition, $\tau.\text{act} \in \llbracket P(R) \rrbracket_{Adr}^{Adr}$. Assume we already constructed $\tau$ some $\sigma \in \llbracket MGC(R) \rrbracket_{Adr}^{Adr}$ with the desired properties. That is, there is $pc_3$ with $pc_2 \circ pc_3 \in \text{ctrl}(\sigma)$. Furthermore, we have the following: $m^R_\tau = m^R_\sigma$, $m^IVar_\tau = m^IVar_\varphi$, $\mathcal{H}(\tau) = \mathcal{H}(\varphi)$, $\text{fresh}_\tau \subseteq \text{fresh}_\varphi$, $\text{freed}_\tau \subseteq \text{freed}_\varphi$. 

Section C.1 Compositionality
and \( \text{used}(r) \subseteq \text{used}(\sigma) \). We construct a computation \( \sigma' \in \langle MGC(R) \rangle_{\text{Adr}}^{\text{Adv}} \) that mimics \( r.\text{act} \). To that end, we show that there is a program step of the form:

\[
(pc_2 \circ pc_3, \sigma) \not\vdash^* (pc_2 \circ pc_3, \sigma')
\]

with the following: \( m_{r.\text{act}}^R = m_{\sigma'}^R, m_{r.\text{act} \downarrow \text{IVar}} = m_{\sigma'} \downarrow \text{IVar}, \text{fresh}_{r.\text{act}} \subseteq \text{fresh}_{\sigma'}, \text{freed}_{r.\text{act}} \subseteq \text{freed}_{\sigma'} \), and \( \text{used}(r.\text{act}) \subseteq \text{used}(\sigma') \), as well as \( \mathcal{H}(r.\text{act}) = \mathcal{H}(\sigma') \). Let \( \text{act} = \langle t, \text{com}, \text{up} \rangle \).

\[\Box\ \text{Case 1: } Q = R\]

Step (1) is due to Rule (sos-std-par) followed by Rule (sos-std-smr). By definition, we have \( pc_1^l = pc_1 \). Let \( \text{stmt}_3 = pc_3(t) \). Then, \( pc_3^l = pc_3[t \mapsto \text{stmt}_3] \) with \( \text{stmt}_3 \stackrel{\text{com}}{\longrightarrow} \text{stmt}_3^l \). By definition of Rule (sos-std-par), we the step \( (\text{stmt}_2 \circ \text{stmt}_3, \sigma) \not\vdash^*_{R,t} (\text{stmt}_2 \circ \text{stmt}_3^l, \sigma.\text{act}) \) satisfies (2), provided we have \( \text{act} \in \text{Act}(\sigma, t, \text{com}) \). We show that \( \text{act} \in \text{Act}(\sigma, t, \text{com}) \) holds and that \( \sigma' = \sigma.\text{act} \) satisfies the required properties. To do this, we rely on the following:

\[
\forall \text{exp. } \text{com contains exp} \implies \text{exp} \in \text{IVar} \cup \text{Var}^R \cup \text{Sel}^R
\quad (3)
\]

\[
\forall \text{exp. } \text{com assigns to exp} \implies \text{exp} \in \text{Var}^R \cup \text{Sel}^R
\quad (4)
\]

\[
\forall \text{exp. } \text{exp} \in \text{IVar} \cup \text{Var}^R \cup \text{Sel}^R \implies m_r(\text{exp}) = m_\sigma(\text{exp})
\quad (5)
\]

Implications (3) and (4) follow from \( r.\text{act} \) being free from separation violations. The remaining implication (5) is due to \( \text{exp} \in \text{dom}(m_r \downarrow \text{IVar}) \cup \text{dom}(m_r^R) \) by definition together with both \( m_r^R = m_\sigma^R \) and \( m_r \downarrow \text{IVar} = m_\sigma \downarrow \text{IVar} \) by induction. Now, we do a case distinction over \( \text{com} \).

\[\Box\ \text{Case 1.1: } \text{com} \in \{ \text{in:func}(r), \text{re:func}, \text{env}(a) \}\]

By definition of Rule (sos-std-smr), the case does apply.

\[\Box\ \text{Case 1.2: } \text{com} \in \{ \text{beginAtomic}, \text{endAtomic}, \text{com} \equiv \text{skip} \}\]

By definition, \( \text{act} \in \text{Act}(\sigma, t, \text{com}) \) as required. We conclude by induction:

\[
m_r^{\text{r.\text{act}}} = m_r^R = m_\sigma = m_\sigma^{\text{r.\text{act}}}
\]

\[
m_r \downarrow \text{IVar} = m_\sigma \downarrow \text{IVar} = m_\sigma \downarrow \text{IVar}
\]

\[
\text{fresh}_{r.\text{act}} = \text{fresh}_r \subseteq \text{fresh}_\sigma = \text{fresh}_{\sigma.\text{act}}
\]

\[
\text{freed}_{r.\text{act}} = \text{freed}_r \subseteq \text{freed}_\sigma = \text{freed}_{\sigma.\text{act}}
\]

\[
\text{used}(r.\text{act}) = \text{used}(\tau) \subseteq \text{used}(\sigma) = \text{used}(\sigma.\text{act})
\]

\[
\mathcal{H}(r.\text{act}) = \mathcal{H}(\tau) = \mathcal{H}(\sigma) = \mathcal{H}(\sigma.\text{act})
\]

\[\Box\ \text{Case 1.3: } \text{com} \equiv \text{exp := exp}'\]

First, we show \( \text{act} \in \text{Act}(\sigma, t, \text{com}) \). To that end, we show that the update \( \text{up} \) is an appropriate update for \( \text{com} \) after \( \sigma \). If \( \text{exp} \in \text{Var} \), then the update is \( \text{up} = \left[ \text{exp} \mapsto m_r(\text{exp}') \right] \).

From (5), we obtain \( m_r(\text{exp}') = m_\sigma(\text{exp}') \). That is, \( \text{up} \) is appropriate. Otherwise, we have

\[172\]

Appendix C  Proof of Meta Theory
exp \notin Var. This means exp \in Sel^R by (3). So exp must be of the form exp \equiv p \cdot sel. Let 
a = m_r(p). The update is up = [a \cdot sel \mapsto m_r(exp)]. From (5), we get 
m_\sigma(p) = a and 
m_r(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_\sigma(exp') = m_\sigma(exp) = b for some 
address b, as argued above. So we get:

\[
m_{r,act} \downarrow \text{IVar} = m_r \downarrow \text{IVar} = m_\sigma \downarrow \text{IVar} = m_{\sigma,act} \downarrow \text{IVar} \\
m_{r,act}^R = m_{r}^R[\text{up}] = m_{\sigma}^R[\text{up}] = m_{\sigma,act}^R
\]

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_r(exp) = m_\sigma(exp) 
follows (3) and (5). So act \in Act(\sigma, t, com) follows because cond has the same truth value 
after \tau and \sigma. 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_r(\text{act}(p)) be the allocated address. We have a \in fresh_r \cup freed_r. By induction, 
we get a \in fresh_\sigma \cup freed_\sigma. This yields act \in Act(\sigma, t, com). Moreover, we get:

\[
\begin{align*}
fresh_{r,act} &= fresh_r \setminus \{a\} \subseteq fresh_\sigma \setminus \{a\} = fresh_{\sigma,act} \\
freed_{r,act} &= freed_r \setminus \{a\} \subseteq freed_\sigma \setminus \{a\} = freed_{\sigma,act} \\
\mathcal{H} = \mathcal{H}(\tau) &= \mathcal{H}(\sigma) = \mathcal{H}(\sigma, \text{act})
\end{align*}
\]

We turn to the remaining properties. We have up = [p \mapsto a, a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d] 
for some d. Observe that p \in Var^R holds by (4). Since Var^R \cap \text{IVar} = \emptyset, we get:

\[
\begin{align*}
m_{r,act} \downarrow \text{IVar} &= m_r \downarrow \text{IVar} = m_\sigma \downarrow \text{IVar} = m_{\sigma,act} \downarrow \text{IVar} \\
\text{used}(r, \text{act}) &= \text{used}(\tau) \subseteq \text{used}(\sigma) = \text{used}(\sigma, \text{act})
\end{align*}
\]

For p we have 
m_{r,act}(\text{act}(p)) = a = m_{\sigma,act}(p). For a.\text{next} we have:

\[
\begin{align*}
m_{r,act}(a.\text{next}) &= \text{seg} = m_{\sigma,act}(a.\text{next}) & \text{if} \text{next} \in Sel^R \\
m_{r,act}(a.\text{next}) &= \bot = m_{\sigma,act}a.\text{next} & \text{otherwise}.
\end{align*}
\]

Similarly, we obtain m_{r,act}(a.\text{data}) = m_{\sigma,act}(a.\text{data}). Altogether, this establishes the 
desired m_{r,act} = m_{\sigma,act} as up does not modify expressions besides p, a.\text{next}, and a.\text{data}.
\begin{itemize}
    \item **Case 1.6:** \( \text{com} \equiv \text{free}(p) \)
    
    We have \( \text{act} \in \text{Act}(\sigma, t, \text{com}) \) 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:
    \[
    \begin{align*}
      \text{fresh}_{\tau, \text{act}} &= \text{fresh}_\tau \setminus \{ a \} \subseteq \text{fresh}_\sigma \setminus \{ a \} = \text{fresh}_{\sigma, \text{act}} \\
      \text{freed}_{\tau, \text{act}} &= \text{freed}_\tau \cup \{ a \} \subseteq \text{freed}_\sigma \cup \{ a \} = \text{freed}_{\sigma, \text{act}} \\
      \mathcal{H}(\tau, \text{act}) &= \mathcal{H}(\tau).\text{free}(a) = \mathcal{H}(\sigma).\text{free}(a) = \mathcal{H}(\sigma.\text{act}) \\
      \text{used}(\tau, \text{act}) &= \text{used}(\tau) \subseteq \text{used}(\sigma) = \text{used}_{\sigma, \text{act}}
    \end{align*}
    \]
    where the last two equalities are due to \( \text{up} = \emptyset \).
    \end{itemize}

\begin{itemize}
    \item **Case 2:** \( Q = P \) and \( \text{com} \notin \text{env}(a) \)
    
    Step (1) is due to Rule \( \text{(sos-std-call)} \). Let \( \text{stmt}_1 = pc_1(t) \) and \( \text{stmt}_3 = pc_3(t) \). By definition, \( pc_1 = pc_1[t \mapsto \text{stmt}_1] \) and \( pc_3 = pc_3[t \mapsto \text{stmt}_3] \) with \( \text{stmt}_1 \circ \text{stmt}_3 \xrightarrow{\text{com}} \text{stmt}_1 \circ \text{stmt}_3 \).
    
    \begin{itemize}
        \item **Case 2.1:** \( \text{com} \equiv \text{in:func}(\tau) \)
            
            Step (1) involves Rule \( \text{(sos-std-call)} \): \( \text{stmt}_1 \equiv \text{skip} \) and \( \text{stmt}_3 \equiv R.\text{func; await func} \). Assume for a moment we have \( \text{stmt}_2 \xrightarrow{\text{com}} \text{stmt}_1 \) and \( \text{act} \in \text{Act}(\sigma, t, \text{com}) \). Then, we satisfy (2) by step \( (pc_2 \circ pc_3, \sigma) \xrightarrow{\text{R,t}} (pc_2 \circ pc_3, \sigma.\text{act}) \) due to Rule \( \text{(sos-std-call)} \) combined with Rule \( \text{(sos-std-par)} \). Now, we show that \( \text{act} \) is enabled after \( \sigma \). By assumption, \( r_i \in IVar \). By induction, \( \text{seg} \notin m_\tau(r_i) = m_\sigma(r_i) \). So, \( \sigma.\text{act} \in \mathcal{O}[P]_{\text{Adr}}^{\text{R,t}} \) by the assumptions on the MGC. This means we satisfy (2) because the step \( \text{stmt}_2 \xrightarrow{\text{com}} \text{stmt}_1 \) exists and \( \text{act} \in \text{Act}(\sigma, t, \text{com}) \) holds. We turn to the remaining properties and establish that \( \sigma' = \sigma.\text{act} \) is an adequate choice. Let \( \text{evt} \) be an event such that \( \mathcal{H}(\tau, \text{act}) = \mathcal{H}(\tau).\text{evt} \). Since we have established \( m_\tau(r_i) = m_\sigma(r_i) \) already, we get \( \mathcal{H}(\sigma.\text{act}) = \mathcal{H}(\sigma).\text{evt} \). By induction, \( \mathcal{H}(\tau, \text{act}) = \mathcal{H}(\sigma.\text{act}) \).

        \item **Case 2.2:** \( \text{com} \equiv \text{re:func} \)
            
            The step is due to Rule \( \text{(sos-std-return)} \): \( \text{stmt}_3 \equiv \text{await func} \) and \( \text{stmt}_3 \equiv \text{skip} \). We show that \( \sigma' = \sigma.\text{act} \) is an appropriate choice. We get \( \sigma.\text{act} \in \mathcal{O}[P]_{\text{Adr}}^{\text{R,t}} \) by the assumptions on the MGC. We find a step \( (pc_2 \circ pc_3, \sigma) \xrightarrow{\text{R,t}} (pc_2 \circ pc_3, \sigma.\text{act}) \) that satisfies (2), as in the previous case. Further, we get:
            \[
            \mathcal{H}(\tau, \text{act}) = \mathcal{H}(\tau).\text{re:func}(t) = \mathcal{H}(\sigma).\text{re:func}(t) = \mathcal{H}(\sigma.\text{act})
            \]
            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$_3$ = stmt$_3$. Observe that we satisfy (2) with $\sigma' = \sigma$ and $\sigma' = \sigma$.act for any act$' \in$ 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 := malloc and p $\notin$ IVar
Let $m_{r.act}(p) = a$. The update up is up $= [p \mapsto a, a$.next $\mapsto$ seg$a$.data $\mapsto$ d] with some d and a $\in$ fresh$_a$ $\cup$ freed$_a$. By induction, we have a $\in$ fresh$_a$ $\cup$ freed$_a$. Let q be some pointer variable of the MGC such that q $\notin$ IVar and q $\notin$ Var$^R$. We show that $\sigma' = \sigma$.act is an appropriate choice for act$' = \langle t, q := malloc, up$\rangle$ with update up $= [q \mapsto a, a$.next $\mapsto$ seg$a$.data $\mapsto$ d]. By the assumptions on the MGC, we have $\sigma$.act$' \in$ O[ P$^R$$_{Adr}$. That is, act$' \in$ Act($\sigma$, t, com) holds. As stated before, this satisfies (2). By induction and definition:

$$m_{r.act} \downarrow_{IVar} = m_{r} \downarrow_{IVar} = m_{\sigma} \downarrow_{IVar} = m_{\sigma}.act \downarrow_{IVar}$$

$$\mathcal{H}(r.act) = \mathcal{H}(r) = \mathcal{H}(\sigma) = \mathcal{H}(\sigma).act'$$

fresh$_{r.act} = fresh_{r} \subseteq \{ a \} \subseteq fresh_{\sigma} \subseteq \{ a \} = fresh_{\sigma}.act'$

freed$_{r.act} = freed_{r} \setminus \{ a \} \subseteq freed_{\sigma} \setminus \{ a \} = freed_{\sigma}.act'$

Next, observe that p $\in$ Var$^P$ since r.act is free from separation violations. Hence, we obtain:

$$used(r.act) = used(r) \cup \{ m_{r.act}(p) \} = used(r) \cup \{ m_{\sigma}.act(q) \}$$

$$\subseteq used(\sigma) \cup \{ m_{\sigma}.act(q) \} = used(\sigma).act'$$

It remains to show $m_{r.act}^R = m_{\sigma}.act$. It suffices to show $m_{r.act}^R(a$.sel$) = m_{\sigma}.act(a$.sel$)$ for all selectors sel $\in$ 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 $m_{r.act}^R(a$.sel$) = \nu = m_{\sigma}.act(a$.sel$)$ where we use $\nu = a$ if sel = next and $\nu = d$ if sel = data.

Case 2.3.2: com $\equiv$ p := malloc and p $\in$ IVar
Follows analogously to the previous case. Here, we can simply choose $\sigma' = \sigma$.act and observe that $m_{r.act}^R(p) = m_{\sigma}.act(p)$ holds in order to obtain $m_{r.act}^R = m_{\sigma}.act$.

Case 2.3.3: com $\equiv$ x := exp and x $\in$ IVar
Let up $= [x \mapsto \nu]$. We show that $\sigma' = \sigma$.act with act$' = \langle t, com', up$\rangle$ is appropriate. By the assumptions on the MGC, act$' \in$ Act($\sigma$, t, com) exists and is enabled, $\sigma$.act$' \in$ O[ P$^R$$_{Adr}$. We have act$' \in$ Act($\sigma$, t, com) and thus satisfy (2) as stated before. Moreover, we get:

$$m_{r.act} \downarrow_{IVar} = m_{r} \downarrow_{IVar}[up] = m_{\sigma} \downarrow_{IVar}[up] = m_{\sigma}.act \downarrow_{IVar}$$

Section C.1 Compositionality
Appendix C

and conclude the remaining properties by definition and induction as before.

- **Case 2.3.4:** \( \text{com} \equiv \text{exp} := \text{exp}' \) with \( \text{exp} \notin \text{IVar} \)

  We show that \( \sigma' = \sigma \) satisfies the desired properties. We get (2) for \( \text{pc}_2' = \text{pc}_2 \). Next, we show \( m^R_{r, \text{act}} = m^R_r \). To do so, it is sufficient to show \( m^R_{r, \text{act}} = m^R_r \) since \( m^R_r = m^R_\sigma \) holds by induction. To the contrary, assume \( m^R_{r, \text{act}} \neq m^R_r \). Note that, by definition, we have \( \text{dom}(m^R_r) = \text{dom}(m^R_{r, \text{act}}) \). Consider \( p \in \text{dom}(m^R_r) \). Because \( r \cdot \text{act} \) is free from separation violations, we have \( \text{exp} \neq \{ \) \( \) \( \} \). Hence, \( m^R_{r, \text{act}}(p) = m^R_r(p) \) holds. That is, there must be \( a \cdot \text{sel} \in \text{dom}(m^R_r) \) such that \( m^R_{r, \text{act}}(a \cdot \text{sel}) \neq m^R_r(a \cdot \text{sel}) \). Hence, we have \( \text{exp} \equiv q \cdot \text{sel} \) with \( m_r(q) = a \). Observe that \( a \cdot \text{sel} \in \text{dom}(m^R_r) \) means \( \text{sel} \in \text{Sel}^R \).

  So, \( \text{act} \) is a separation violation. This contradicts the assumptions and thus concludes the desired \( m^R_{r, \text{act}} = m^R_r \). The remaining properties follow by definition together with induction.

- **Case 2.3.5:** \( \text{com} \in \{ \) some \( \} \) as \( \sigma' = \sigma \) 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 \( (\text{sos - std - env}) \). We have \( \text{pc}_1' = \text{pc}_1 \) and \( \text{pc}_3' = \text{pc}_3 \) and \( t = \bot \). By definition, \( a \in \text{fresh}_r \cup \text{freen}_r \). By induction, \( a \in \text{fresh}_\sigma \cup \text{freen}_\sigma \). So, \( \text{act} \in \text{Act}(\sigma, \bot, \text{com}) \). That is, we obtain the step \( (\text{pc}_1 \circ \text{pc}_3, \sigma) \vdash_{\text{P}_\bot} (\text{pc}_1 \circ \text{pc}_3, \sigma \cdot \text{act}) \) which satisfies (2). Next, we establish \( m^R_{r, \text{act}} = m^R_{\sigma, \text{act}} \). To that end, consider \( \text{exp} \in \text{PExp} \cup \text{DExp} \). If \( \text{exp} \cap \text{Adr} \neq \{ \) \( \) \( \} \), then we get \( m^R_{r, \text{act}} \cdot \text{exp} = m^R_r \cdot \text{exp} = m^R_{\sigma, \text{act}} \cdot \text{exp} \) where the second equality holds by induction and the first/third equality holds by \( \text{up} \). It remains to show, for all \( \text{sel} \in \text{Sel}^R \), that \( m^R_{r, \text{act}}(a \cdot \text{sel}) = m^R_{\sigma, \text{act}}(a \cdot \text{sel}) \) holds. This follows from the fact that \( \text{up} \) and \( \text{up}' \) agree on the updates they perform on selectors. The remaining properties follow by induction since \( \text{act} \) 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 \( r \). In the base case, we have the empty computation \( r = \epsilon \). Then, the claim follows by definition for \( \sigma = \epsilon \). For the induction step, consider \( r \in \llbracket P(R) \rrbracket^{\text{Adr}} \) and the following program step in the standard semantics:

\[
(\text{pc}_1 \circ \text{pc}_2, r) \vdash_{Q_r} (\text{pc}_1 \circ \text{pc}_2, r \cdot \text{act}) \quad \text{with} \quad \text{pc}_1 \circ \text{pc}_2 \in \text{ctrl}(r) . 
\]  

(6)

By definition, \( r \cdot \text{act} \in \llbracket P(R) \rrbracket^{\text{Adr}} \). Assume we already constructed for \( r \) some \( \sigma \in \llbracket P(R) \rrbracket^{\text{Adr}} \) with the following: \( \text{stmt}_1 \in \text{ctrl}(\sigma), m_r = m^P_\sigma, H(r) = H(\sigma), \text{fresh}_r \subseteq \text{fresh}_\sigma, \text{freen}_r \subseteq \text{freen}_\sigma, \)
and retired \( \leq \) retired. We now construct a computation \( \sigma' \in \{P(R)\}_{Adr} \) that mimics \( r.\text{act} \). More precisely, we show that there is a program step in the SMR semantics of the form

\[
(pc_1, \sigma) \rightarrow^* (pc_1', \sigma')
\]

satisfying the following: \( m_{r.\text{act}}' = m_r' \), \( \mathcal{H}(r.\text{act}) = \mathcal{H}(\sigma') \), \( \text{fresh}_{r.\text{act}} \subseteq \text{fresh}_{\sigma'} \), \( \text{freed}_{r.\text{act}} \subseteq \text{freed}_{\sigma'} \), and \( \text{retired}_{r.\text{act}} \subseteq \text{retired}_{\sigma'} \). Let \( \text{act} = \{t, \text{com}, \text{up}\} \).

\[\Diamond \text{ Case 1: } Q = R\]

Step (6) is due to Rule (\text{sos-std-par}) followed by Rule (\text{sos-std-smr}). By definition, we have \( pc_1' = pc_1 \). Let \( stmt_2 = pc_2(t) \). Then, \( pc_2' = pc_2[t \mapsto stmt_2'] \) with \( stmt_2 \xrightarrow{\text{com}} stmt_2' \).

\[\Diamond \text{ Case 1.1: } \text{com} \in \{\text{in:func}(\mathcal{F}), \text{re:func}\}\]

According to Rule (\text{sos-std-smr}), the case does not apply.

\[\Diamond \text{ Case 1.2: } \text{com} \in \{\text{skip}, \text{beginAtomic}, \text{endAtomic}, \text{assume cond}\}\]

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:

\[
m_{r.\text{act}}' = m_r' = m_{\sigma} = m_{\sigma.\text{act}}
\]

\[
\mathcal{H}(r.\text{act}) = \mathcal{H}(\tau) = \mathcal{H}(\sigma) = \mathcal{H}(\sigma.\text{act})
\]

\[
\text{fresh}_{r.\text{act}} \subseteq \text{fresh}_{\tau} \subseteq \text{fresh}_{\sigma} \subseteq \text{fresh}_{\sigma.\text{act}}
\]

\[
\text{freed}_{r.\text{act}} \subseteq \text{freed}_{\tau} \subseteq \text{freed}_{\sigma} \subseteq \text{freed}_{\sigma.\text{act}}
\]

\[
\text{retired}_{r.\text{act}} \subseteq \text{retired}_{\tau} \subseteq \text{retired}_{\sigma} \subseteq \text{retired}_{\sigma.\text{act}}
\]

\[\Diamond \text{ Case 1.3: } \text{com} \equiv \text{exp := exp}'\]

We choose \( \sigma' = \sigma \). We immediately satisfy (7) by \( pc_1' = pc_1 \). Next, we show \( m_{r.\text{act}}' = m_{\sigma} \).

To that end, it suffices to establish \( m_{r.\text{act}}' = m_r' \). First, consider the case where \( \text{exp} \in \text{Var} \) holds. Then, \( \text{exp} \in \text{Var}^R \) because \( r.\text{act} \) is free from separation violations. By definition, we obtain \( m_{r.\text{act}} = m_r[\text{exp} \mapsto m_r(\text{exp}')] \). That is, \( m_{r.\text{act}}' = m_r' \) because \( \text{Var}^R \cap \text{Var}^p = \emptyset \).

Second, consider the remaining case where \( \text{exp} \notin \text{Var} \) holds. So, \( \text{exp} \equiv r.\text{p} \; . \; \text{sel} \) for some pointer \( r.\text{p} \) and selector \( \text{sel} \). Let \( a = m_r(p) \). We have \( m_{r.\text{act}} = m_r[\text{a.sel} \mapsto m_r(\text{exp}')] \).

Since \( r.\text{act} \) is free from separation violations, \( \text{sel} \in \text{Sel}^R \). So, we get \( m_{r.\text{act}}' = m_r' \) because of \( \text{Sel}^R \cap \text{Sel}^p = \emptyset \). The remaining properties follow by definition together with induction as before.

\[\Diamond \text{ Case 1.4: } \text{com} \equiv \text{p} := \text{malloc}\]

Let \( a = m_{r.\text{act}}(p) \). The update is \( \text{up} = [p \mapsto a, a.\text{next} \mapsto \text{seg}.a.\text{data} \mapsto d] \) for some \( d \).

We choose \( \sigma' = \sigma.\text{act}' \) with \( \text{act}' = \{t, \text{env}(a), \text{up}'\} \) and \( \text{up} = [a.\text{next} \mapsto \text{seg}.a.\text{data} \mapsto d] \).

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 \( \text{act}' \in \overline{\text{Act}}(\sigma, t, \text{env}(a)) \). Rule (\text{sos-env}) yields the step \( (pc_1, \sigma) \rightarrow (pc_1, \sigma.\text{act}') \).
which satisfies (7). Next, we show $m_{r,act}^p = m_{\sigma,act}^p$. To that end, we establish, for all expressions $\exp$, that $m_{r,act}^p(\exp) = m_{\sigma,act}^p(\exp)$ holds. By induction, we have:

$$m_{r,act}^p(\exp) = m_r^p(\exp) = m_{\sigma}^p(\exp) = m_{\sigma,act}^p(\exp) \quad \text{if} \quad \exp \not\in \{p, a.next, a.data\}.$$

We turn to $p, a.next$, and $a.data$. Regarding $p$, note that we have $p \not\in \Var^R$ because $r.act$ is free from separation violations. That is, $m_{r,act}^p(p) = \bot = m_{\sigma,act}^p(p)$. Consider $a.next$. If $next \in \Sel^p$, we get $m_{r,act}^p(a.next) = a = m_{\sigma,act}^p(a.next)$ due to the form of the updates $up$ and $up'$. Otherwise, we have $m_{r,act}^p(a.next) = \bot = m_{\sigma,act}^p(a.next)$ by definition of the memory separation. Similarly, we obtain $m_{r,act}^p(a.data) = m_{\sigma,act}^p(a.data)$. Altogether, we conclude the desired $m_{r,act}^p = m_{\sigma,act}^p$. The remaining properties follow by definition and induction as before.

\[\text{Case 1.5: } com \equiv \text{free}(p)\]

Let $m_r(p) = a$. We choose $\sigma' = \sigma.act'$ with $act' = \langle t, \text{free}(a), \emptyset \rangle$. By the standard semantics, we have $act \in \Act(r, t, com)$. Hence, $act' \in \Act(\sigma, \bot, \text{free}(a))$ holds according to the SMR semantics. Then, Rule $(\text{sos-free})$ yields $(pc_1, \sigma) \rightarrow (pc_1, \sigma.act')$ which satisfies (7). For the remaining properties, we conclude by definition and induction as before.

\[\text{Case 2: } Q = P \text{ and } com \not\equiv \text{env}(a)\]

Step (6) is due to Rule $(\text{sos-std-par})$. 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 $stmt_1 \circ stmt_2 \xrightarrow{\text{com}} stmt'_1 \circ stmt'_2$. We show that $\sigma' = \sigma.act$ is an appropriate choice. By the SMR semantics, $stmt_1 \circ stmt_2 \xrightarrow{\text{com}} stmt'_1$ holds. Hence, Rule $(\text{sos-par})$ yields $(pc_1, \sigma) \rightarrow (pc'_1, \sigma.act)$ satisfying (7), provided we have $act \in \Act(\sigma, t, com)$. We establish $act \in \Act(\sigma, t, com)$ and the remaining properties.

\[\text{Case 2.1: } com \in \{\text{skip, beginAtomic, endAtomic}\}\]

By definition, $act \in \Act(\sigma, t, com)$. We conclude by definition and induction:

$$m_{\tau,act}^p = m_r^p = m_{\sigma}^p = m_{\sigma,act}^p$$

$H(\tau.act) = H(\tau) = H(\sigma) = H(\sigma.act)$

$\text{fresh}_{\tau,act} = \text{fresh}_{\tau} \subseteq \text{fresh}_{\sigma} = \text{fresh}_{\sigma,act}$

$\text{freed}_{\tau,act} = \text{freed}_{\tau} \subseteq \text{freed}_{\sigma} = \text{freed}_{\sigma,act}$

$\text{retired}_{\tau,act} = \text{retired}_{\tau} \subseteq \text{retired}_{\sigma} = \text{retired}_{\sigma,act}$

\[\text{Case 2.2: } com \equiv \text{in: func}(\tau)\]

Step (6) involves Rule $(\text{sos-std-call})$. By assumption, we have $r_i \in \IVar$. By induction, we get $m_r(r_i) = m_{\sigma}(r_i)$. Hence, $m_{\sigma}(r_i) \neq \text{seg}$ because $m_r(r_i) \neq \text{seg}$ according to the semantics. This gives $act \in \Act(\sigma, t, com)$ by definition. Now, let $evt$ be the event emitted
by \( \text{act} \) after \( \tau \), that is, \( \mathcal{H}(\tau.\text{act}) = \mathcal{H}(\tau).\text{evt} \). Because we have already established \( m_\tau(r_\tau) = m_\sigma(r_\tau) \), \( \text{act} \) must emit the same event after \( \sigma \), i.e., \( \mathcal{H}(\sigma.\text{act}) = \mathcal{H}(\sigma).\text{evt} \). So, \( \mathcal{H}(\tau.\text{act}) = \mathcal{H}(\sigma.\text{act}) \) follows by induction. For the remaining property, let \( M \subseteq \text{Adr} \) such that \( M = \{ a \} \) if \( \text{evt} \equiv \text{in}:\text{retire}(t,a) \) and \( M = \emptyset \) otherwise. Then, we get:

\[
\text{retired}_\tau = \text{retired}_\sigma \cup M \subseteq \text{retired}_\sigma \cup M = \text{retired}_\sigma. 
\]

The remaining properties follow by definition and induction as before.

\( \textbf{Case 2.3: } \com \equiv \text{re} : \text{func} \)

Step (6) involves Rule \( \text{(sos-std-return)} \). By definition, \( \act \in \mathcal{Act}(\sigma, t, \text{re} : \text{func}) \). We get:

\[
\mathcal{H}(\tau.\text{act}) = \mathcal{H}(\tau).\text{re} : \text{func}(t) = \mathcal{H}(\sigma).\text{re} : \text{func}(t) = \mathcal{H}(\sigma.\text{act})
\]

and conclude the remaining properties by definition and induction as before.

\( \textbf{Case 2.4: } \com \equiv p := \text{malloc} \)

Let \( a = m_\tau(p) \). Then, the update is \( \text{up} = [p \mapsto a, \text{a.next} \mapsto \text{seg}, \text{a.data} \mapsto d] \) 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 \( \act \in \mathcal{Act}(\sigma, t, \com) \). Moreover, we get:

\[
\begin{align*}
\text{fresh}_\tau & = \text{fresh}_\tau \setminus \{ a \} \subseteq \text{fresh}_\sigma \setminus \{ a \} = \text{fresh}_\sigma, \\
\text{freed}_\tau & = \text{freed}_\tau \setminus \{ a \} \subseteq \text{freed}_\sigma \setminus \{ a \} = \text{freed}_\sigma, \\
\text{retired}_\tau & = \text{retired}_\sigma \subseteq \text{retired}_\sigma = \text{retired}_\sigma, \\
\mathcal{H}(\tau.\text{act}) = \mathcal{H}(\tau) = \mathcal{H}(\sigma) = \mathcal{H}(\sigma.\text{act})
\end{align*}
\]

It remains to establish \( m^\text{p}_\tau = m^\text{p}_\sigma = m^\text{p}_\sigma. \) By induction and the form of \( \text{up} \), we have:

\[
m^\text{p}_\tau(\exp) = m^\text{p}_\tau(\exp) = m^\text{p}_\sigma(\exp) = m^\text{p}_\sigma. \text{act} \exp = \text{if } \exp \notin \{ p, \text{a.next}, \text{a.data} \}.
\]

Hence, it suffices to show \( m^\text{p}_\tau(\exp') = m^\text{p}_\sigma(\exp') \) for \( \exp' \in \{ p, \text{a.next}, \text{a.data} \} \). By the definition of the memory separation, it suffices to show \( m^\text{p}_\tau(\exp') = m^\text{p}_\sigma(\exp') \). This follows immediately from the performed update \( \text{up} \).

\( \textbf{Case 2.5: } \com \equiv p.\text{sel} := \text{exp} \)

By definition of the syntx, we have \( \exp \in \text{Var} \). Let \( a = m_\tau(p) \) and \( v = m_\tau(\exp) \). Then, the update is \( \text{up} = [a.\text{sel} \mapsto v] \). Since \( \tau.\text{act} \) is free from separation violations, we get \( p, \exp \in \text{Var}_\tau \) and \( \text{sel} \in \text{Sel}_\tau \). Hence, \( \{ p, \exp, a.\text{sel} \} \subseteq \text{dom}(m^\text{p}_\tau) \). By induction, we have \( m^\text{p}_\tau(p) = a \) and \( m^\text{p}_\tau(\exp) = v \). This means \( \text{up} \) is a valid for \( \text{act} \) after \( \sigma \). That is, we obtain \( \act \in \mathcal{Act}(\sigma, t, \com) \). From \( \text{sel} \in \text{Sel}_\tau \) we get:

\[
m^\text{p}_\tau(\text{a.sel} \mapsto a) = m^\text{p}_\tau[\text{a.sel} \mapsto v] = m^\text{p}_\sigma = m^\text{p}_\sigma. 
\]
The remaining properties follow by definition and induction as before.

**Case 2.6:** $com \equiv p := \text{exp}'$

Analogous to the previous case.

**Case 2.7:** $com \equiv \text{assume cond}$

Let $\text{exp}$ be an expression in $\text{cond}$. 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 $\text{Var}^p$ and selectors from $\text{Sel}^p$ can occur in $\text{exp}$. Then, we arrive at $\text{act} \in \overline{\text{Acf}}(\sigma, t, \text{com})$ since $\text{cond}$ has the same truth value after $\tau$ and $\sigma$. The remaining properties follow by induction.

**Case 3:** $Q = P$ and $com \equiv \text{env}(a)$

Step (6) is due to Rule $(\text{sos-std-env})$. By definition of the rule, we have $pc_\tau = pc_\tau'$ as well as $up = [a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d]$ for some value $d$. By definition, $a \in \text{fresh}_\sigma \cup \text{freed}_\sigma$.

By induction, $a \in \text{fresh}_\sigma \cup \text{freed}_\sigma$. Hence, we obtain $\text{act} \in \overline{\text{Acf}}(\sigma, t, \text{com})$ such that the step $(pc_\tau, a) \rightarrow (pc_\tau', \sigma.\text{act})$ by Rule $(\text{sos-std-env})$ satisfies (7). Next, we show $m_{\tau.\text{act}} = m_{\sigma.\text{act}}$.

By induction together with the form of $up$, we have:

$$m_{\tau.\text{act}}(\text{exp}) = m_{\tau}(\text{exp}) = m_{\sigma}(\text{exp}) = m_{\sigma.\text{act}}(\text{exp}) \text{ if } \text{exp} \notin \{a.\text{next}, a.\text{data}\}.$$

Hence, it suffices to show $m_{\tau.\text{act}}(\text{exp}') = m_{\sigma.\text{act}}(\text{exp}')$ 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 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.

**Proof C.8** (Theorem 5.10). Note that Theorem A.4 implicitly assumes that $\lbrack P(R) \rbrack_{Adr}$ 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 $\tau \in \lbrack P(R) \rbrack_{Adr}$. From Theorem A.4 we get $\sigma \in O[\lbrack P \rbrack_{Adr}]$ with $\text{ctrl}^P(\tau) = \text{ctrl}(\sigma)$. By assumption, we have $\text{good}(\sigma)$. That is, $\text{ctrl}^P(\sigma) \cap \text{Fault} = \emptyset$. From this we get $\text{ctrl}^P(\tau) \cap \text{Fault} = \emptyset$. This gives $\text{good}(\tau)$ as required.

**Proof C.9** (Theorem 5.11). As noted in Proof C.8 already, Theorem 5.11 comes with the implicit assumption that $\lbrack P(R) \rbrack_{Adr}$ is free from separation violations. Towards a contradiction, assume that $\lbrack P(R) \rbrack_{Adr}$ is not free from double retires. That is, there is a computation $\tau.\text{act} \in \lbrack P(R) \rbrack_{Adr}$ with $\text{act} = (t, \text{in} \cdot \text{retire}(p), up)$ and $m_{\tau.\text{act}}(p) \notin \text{retired}_\tau$. Theorem A.4 yields $\sigma \in O[\lbrack P \rbrack_{Adr}]$ with $m_\sigma = m_\tau$ and $\text{retired}_\sigma \subseteq \text{retired}_\tau$. We obtain $\sigma.\text{act} \in O[\lbrack P \rbrack_{Adr}]$. To see that $\text{act}$ is enabled, note that $p \in \text{IVar} \subseteq \text{Var}^p$ by assumption and thus $m_\sigma(p) = m_\tau(p) \neq \text{seg}$. Moreover, this
means \( m_r(p) \in \text{retired}_r \). That is, \( \sigma . \text{act} \) is a double retire. This contradicts the assumption of the semantics \( O[\text{P}]^{\text{Adr}} \) being free from double retires.

\[ \]  

**C.2 Ownership**

**Proof C.10 (Theorem 6.7).** We show the contrapositive:

\[
\forall \tau, p, t. \quad p \notin \text{local}, \land p \in \text{valid}_r \quad \Longrightarrow \quad m_r(p) \notin \text{owned}_r(t).
\]

To that end, we proceed by induction over the structure of \( \tau \). In the base case, \( \tau = \epsilon \). Then, the claim follows by \( \text{owned}_r(t) = \emptyset \). For the induction step, consider \( r.\text{act} \in O[\text{P}]^{\text{Adr}} \) and assume that the claim holds for \( \tau \). Consider some thread \( t \) and some \( x \in \text{PVar} \setminus \text{local}_r \) such that \( p \in \text{valid}_r \). We show that \( m_r(p) \notin \text{owned}_r(t) \) holds. Let \( \text{act} = \langle t', \text{com}, \text{up} \rangle \).

\[ \diamond \text{Case 1: } t \neq t' \]

By definition, we have \( \text{owned}_r(t) \subseteq \text{owned}_r(t) \).

\[ \diamond \text{Case 1.1: } x \notin \text{shared} \]

If \( x \notin \text{shared} \), then \( x \) cannot occur in \( \text{com} \) by the semantics. Hence, \( x \in \text{valid}_r \) implies \( x \in \text{valid}_r \). Moreover, \( m_r.\text{act}(x) = m_r(x) \). By induction, \( m_r(x) \notin \text{owned}_r(t) \).

Hence, we obtain \( m_r.\text{act}(x) \notin \text{owned}_r.\text{act}(t) \) as required.

\[ \diamond \text{Case 1.2: } x \in \text{shared} \text{ and } [x \rightarrow \bullet] \notin \text{up} \]

That \( x \) does not receive an update means that it is not the target of an assignment nor an allocation. We get \( m_r(x) = m_r.\text{act}(x) \) by definition. Moreover, we obtain \( x \in \text{valid}_r \) by \( x \in \text{valid}_r.\text{act} \). By induction, \( m_r(x) \notin \text{owned}_r(t) \). Hence, \( m_r.\text{act}(x) \notin \text{owned}_r.\text{act}(t) \) as required.

\[ \diamond \text{Case 1.3: } x \in \text{shared} \text{ and } [x \rightarrow a] \notin \text{up} \]

By \( \text{owned}_r.\text{act}(t) \subseteq \text{owned}_r(t) \), we know that \( \text{com} \) cannot be an allocation targeting \( x \). So, \( \text{com} \equiv x := \text{pexp} \). First, consider \( \text{pexp} \in \text{PVar} \). To arrive at \( x \in \text{valid}_r.\text{act} \), we must have \( \text{pexp} \in \text{valid}_r \). As this gives a contradicting \( m_r.\text{act}(x) = a \notin \text{owned}_r.\text{act}(t) \), the case cannot apply. That is, \( \text{pexp} \equiv p.\text{next} \). Let \( b = m_r(p) \). To arrive at \( x \in \text{valid}_r.\text{act} \), we must have \( p, b.\text{next} \in \text{valid}_r \). By definition, this results in \( m_r.\text{act}(x) = a \notin \text{owned}_r.\text{act}(t) \). Hence, the case cannot apply.

\[ \diamond \text{Case 2: } t = t' \]

We distinguish three cases.

\[ \diamond \text{Case 2.1: } x \notin \text{shared} \]

By the semantics, \( x \) cannot occur in \( \text{com} \). We get \( x \in \text{valid}_r \) and \( m_r(x) = m_r.\text{act}(x) \).
Hence, \( m_{r,act}(x) \notin owned_{r}(t) \). If \( owned_{r,act}(t) \subseteq owned_{r}(t) \), then nothing remains to be shown. Consider now \( owned_{r,act}(t) \notin owned_{r}(t) \). By definition, this means we must have \( com \equiv p := \text{malloc} \) and thus \( owned_{r,act}(t) = owned_{r}(t) \cup \{a\} \) where \( a = m_{r,act}(p) \). If \( m_{r}(x) = a \), then \( x \notin valid_{r} \) by the definition of validity. Since this contradicts the previous \( x \in valid_{r} \), we must have \( m_{r}(x) \neq a \). Hence, \( m_{r,act}(x) \notin owned_{r,act}(t) \) follows as required.

\begin{itemize}
  \item Case 2.2: \( x \in shared \) and \( [x \mapsto \bullet] \notin up \)
  
  That \( x \) does not receive an update means it is not the target of an assignment nor an allocation. We get \( x \in valid_{r} \) and \( m_{r}(x) = m_{r,act}(x) \). We conclude as in the previous case.

  \item Case 2.3: \( x \in shared \) and \( [x \mapsto a] \subseteq up \)
      
  To the contrary, assume \( com \equiv x := \text{malloc} \). This means \( m_{r,act}(x) \in fresh_{r} \cup freed_{r} \). By definition, \( m_{r,act}(x) \notin owned_{r}(t) \). Because of \( x \in shared \), the allocated address is not owned, that is, \( m_{r,act}(x) \notin owned_{r,act}(t) \) by definition. Since this contradicts the choice of \( x \), we must have \( com \notin x := \text{malloc} \). Hence, we get \( m_{r,act}(x) \in owned_{r,act}(t) \subseteq trt \).

  Because \( com \) is no allocation but updates \( x \), it must be an assignment, \( com \equiv x := pexp \). By \( x \in shared \), we must have \( pexp \in PVar \) in order to get \( m_{r,act}(x) = a \in owned_{r,act}(t) \). To get \( x \in valid_{r,act} \), we must have \( pexp \in valid_{r} \). We get \( m_{r,act}(x) = a \notin owned_{r,act}(t) \). Since this contradicts the choice of \( x \), the case cannot apply.
\end{itemize}

The above case distinction is complete and thus concludes the induction.

\section{C.3 Reductions}

\begin{proof}[Proof C.11 (Lemma B.31)]
By definition.
\end{proof}

\begin{proof}[Proof C.12 (Lemma B.32)]
By definition.
\end{proof}

\begin{proof}[Proof C.13 (Lemma B.33)]
By definition.
\end{proof}

\begin{proof}[Proof C.14 (Lemma B.34)]
By definition.
\end{proof}

\begin{proof}[Proof C.15 (Lemma B.35)]
By definition we have:
\end{proof}
\[
\text{dom}(m_{r \mid \text{valid}_r}) \cap \text{Adr} = (\text{valid}_r \cup \text{DVar} \cup \{ a.\text{data} \mid a \in m_r(\text{valid}_r) \}) \cap \text{Adr} \\
= (\text{valid}_r \cap \text{Adr}) \cup (\{ a.\text{data} \mid a \in m_r(\text{valid}_r) \}) \cap \text{Adr} \\
= (\text{valid}_r \cap \text{Adr}) \cup (\{ a \mid a \in m_r(\text{valid}_r) \}) \\
= (\text{valid}_r \cap \text{Adr}) \cup m_r(\text{valid}_r)
\]

and
\[
\text{range}(m_{r \mid \text{valid}_r}) \cap \text{Adr} = m_r(\text{dom}(m_{r \mid \text{valid}_r})) \cap \text{Adr} = m_r(\text{dom}(m_{r \mid \text{valid}_r}) \cap \text{PExp}) \\
= m_r(\text{valid}_r)
\]

so that we conclude as follows

\[
\text{adr}(m_{r \mid \text{valid}_r}) \cap \text{Adr} = (\text{dom}(m_{r \mid \text{valid}_r}) \cap \text{range}(m_{r \mid \text{valid}_r})) \cap \text{Adr} \\
= (\text{valid}_r \cap \text{Adr}) \cup m_r(\text{valid}_r) \cup m_r(\text{valid}_r) \\
= (\text{valid}_r \cap \text{Adr}) \cup m_r(\text{valid}_r).
\]

\[\text{Proof C.16} \text{ (Lemma B.36).} \text{ The claim holds for } \epsilon \text{ since } \text{valid}_r = \text{PVar and } \text{PVar} \subseteq \text{dom}(m_r) \text{ by definition. Towards a contradiction, assume the claim does not hold. Then, there is a shortest computation } r.\text{act} \in \mathcal{C}[\text{P}]_{\text{Adr}} \text{ with } m_{r.\text{act}}(\text{valid}_r) \notin \text{dom}(m_{r.\text{act}}). \text{ That is, there is } \text{pexp} \in \text{valid}_{r.\text{act}} \text{ with } m_{r.\text{act}}(\text{pexp}) = \bot. \text{ First, consider the case } \text{pexp} \notin \text{valid}_r. \text{ In order to validate } \text{pexp}, \text{ act must be (i) an allocation } p := \text{malloc} \text{ with } m_{r.\text{act}}(p) = a \text{ as well as } \text{pexp} \in \{ p, a.\text{next} \}, \text{ (ii) an assumption } \text{assume} \text{pexp} = q \text{ with } q \in \text{valid}_r, \text{ or (iii) an assignment of the form } \text{pexp} := \text{qexp} \text{ with } \text{qexp} \in \text{valid}_r. \text{ Case (i) cannot apply as it results in } \text{pexp} \in \text{dom}(m_{r.\text{act}}). \text{ Case (ii) cannot apply because } m_r(\text{pexp}) = m_r(q) \text{ together with } q \in \text{dom}(m_r) = \text{dom}(m_{r.\text{act}}) \text{ by minimality gives } \text{pexp} \in \text{dom}(m_{r.\text{act}}). \text{ So case (iii) must apply. By minimality, however, } \text{qexp} \in \text{valid}_r \text{ yields } m_r(\text{qexp}) \neq \bot. \text{ We get } m_{r.\text{act}}(\text{pexp}) \neq \bot, \text{ contradicting the choice of } \text{pexp}. \text{ So the case does not apply and } \text{pexp} \in \text{valid}_r \text{ must hold.}
\]

Consider now the case that \text{pexp} \in \text{valid}_r. \text{ By minimality, } m_r(\text{pexp}) \neq \bot. \text{ So act must update } \text{pexp} \text{ to } \bot. \text{ To do that, act must be an assignment } \text{pexp} := \text{qexp} \text{ with } m_r(\text{qexp}) = \bot. \text{ We have } \text{qexp} \notin \text{valid}_r. \text{ Hence, we get } \text{pexp} \notin \text{valid}_{r.\text{act}}. \text{ This contradicts the choice of } \text{pexp}. \]

\[\text{Proof C.17} \text{ (Lemma B.37).} \text{ We conclude as follows using the definition of } \tau \sim \sigma, \text{ set theory, and the definition of restrictions:}
\]
\[
\tau \sim \sigma \implies m_{r \mid \text{valid}_r} = m_{\sigma \mid \text{valid}_\sigma} \implies \text{dom}(m_{r \mid \text{valid}_r}) = \text{dom}(m_{\sigma \mid \text{valid}_\sigma}) \\
\implies \text{dom}(m_{r \mid \text{valid}_r}) \cap \text{PExp} = \text{dom}(m_{\sigma \mid \text{valid}_\sigma}) \cap \text{PExp} \implies \text{valid}_r = \text{valid}_\sigma
\]

where the last implication is due to Lemma B.36.

\[\text{Proof C.18} \text{ (Lemma B.38).} \text{ Follows immediately from } \tau \sim \sigma, \text{ Lemma B.35, and Lemma B.37.}\]
Proof C.19 (Lemma B.39). Let $r \sim \sigma$ and let $com \in \text{next-com}(r)$. By definition, $pc \in \text{ctrl}(r)$ exists such that for some thread $t$ we have $pc(t) \xrightarrow{\text{com}} \tau$. By $r \sim \sigma$, we have $\text{ctrl}(\tau) = \text{ctrl}(\sigma)$. Hence, $com \in \text{next-com}(\sigma)$.

Proof C.20 (Lemma B.40). Let $r.\text{act} \in \mathcal{O}[P]_{\text{Adr}}$ with $\text{act} = \langle t, \text{com}, \text{up} \rangle$ and $t \neq \bot$. By the semantics, there is $(pc, r) \rightarrow (pc[t \mapsto \text{stmt}], r.\text{act})$ for some $pc \in \text{ctrl}(\tau)$ with $pc(t) \xrightarrow{\text{com}} \text{stmt}$. Hence, the required $com \in \text{next-com}(r)$ 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 $h_3 \in \mathcal{F}_\mathcal{O}(h_1, h_2, a)$. Towards a contradiction, assume that $h_2, h_3 \notin \mathcal{F}_\mathcal{O}(h_1, a)$ holds. We have $\text{frees}_{h_1} \subseteq \{ a \}$ by assumption and $\text{frees}_{h_3} \subseteq \{ a \}$ by definition. So, $h_1, h_2, h_3 \notin \mathcal{S}(\mathcal{O}_{\text{SMR}})$. By definition, however, this gives $h_3 \notin \mathcal{F}_\mathcal{O}(h_1, h_2, a)$, contradicting the assumption.

Proof C.22 (Lemma B.42). To the contrary, let $r.\text{act} \in \mathcal{O}[P]_{\text{Adr}}$ be the shortest computation such that there is a $a \in \text{fresh}_{r.\text{act}}$ with $a \in \text{range}(m_{r.\text{act}})$. By definition, there is $\text{pexp} \in \text{PExp}$ with $m_{r.\text{act}}(\text{pexp}) = a$. Moreover, $a \in \text{fresh}_r$. By minimality, we get $m_r(\text{pexp}) \neq a$. Hence, $\text{act}$ must update $\text{pexp}$ to $a$. That is, $\text{act}$ performs an allocation or a pointer assignment. In the former case, we have $\text{act} = \langle t, \text{pexp} := \text{malloc}, [p \mapsto a, \ldots] \rangle$. Then, $a \notin \text{fresh}_{r.\text{act}}$ follows by definition. Since this contradicts the assumption, the case cannot apply. That is, $\text{act}$ is of the form $\text{act} = \langle t, \text{pexp} := \text{qexp}, \text{up} \rangle$ with $m_r(\text{qexp}) = a$. This means $a \in \text{range}(m_r)$, contradicting the minimality of $r.\text{act}$.

Proof C.23 (Lemma B.43). Towards a contradiction, assume there is a shortest $r.\text{act} \in \mathcal{O}[P]_{\text{Adr}}$ such that there is some $a \in \text{fresh}_{r.\text{act}}$ with $a \in m_{r.\text{act}}(\text{valid}_{r.\text{act}}) \lor a.\text{next} \in \text{valid}_{r.\text{act}}$. Note that $r.\text{act}$ is indeed the shortest such computation since the claim holds for $\varepsilon$. By monotonicity, we have $a \in \text{fresh}_r$. By minimality of $r.\text{act}$ we have $a \notin m_r(\text{valid}_r)$ and $a.\text{next} \notin \text{valid}_r$. If $a.\text{next} \in \text{valid}_{r.\text{act}}$, then $\text{act}$ is an assignment of the form $p.\text{next} := q$ with $m_r(p) = a$; note that an allocation of $a$ could validate $a.\text{next}$ as well but would give $a \notin \text{fresh}_{r.\text{act}}$ contradicting the assumption. This means $a \in \text{range}(m_r)$. So we get $a \notin \text{fresh}_r$ from Lemma B.42. This contradicts $a \in \text{fresh}_r$ from above. Hence, the case does not apply and have $a.\text{next} \notin \text{valid}_{r.\text{act}}$. By assumption then, $a \in m_{r.\text{act}}(\text{valid}_{r.\text{act}})$. So there is some $\text{pexp} \in \text{valid}_{r.\text{act}}$ with $m_{r.\text{act}}(\text{pexp}) = a$. Since we already established $a \notin m_r(\text{valid}_r)$, we must have $\text{pexp} \notin \text{valid}_r$ or $m_r(\text{pexp}) \neq a$. Consider $\text{pexp} \notin \text{valid}_r$. That is, $\text{act}$ validates $\text{pexp}$. To do so, $\text{act}$ must be an assignment, an allocation, or an assumption. We do a case distinction.

\textbf{Case 1:} $\text{act} = \langle t, \text{pexp} := \text{qexp}, \text{up} \rangle$

Then $\text{qexp} \in \text{valid}_r$ and $m_r(\text{qexp}) = a$. We obtain $a \in m_r(\text{valid}_r)$. This constitutes a contradiction, the case does not apply.
Case 2: $act = \langle t, p := \mathsf{malloc}, [p \mapsto a, a\text{.next} \mapsto \mathsf{seg}, a\text{.data} \mapsto d] \rangle$

We obtain $a \notin \mathsf{fresh}_{r\text{-}act}$ what contradicts the assumption. The case does not apply.

Case 3: $act = \langle t, \text{assume } p = q, \text{ up} \rangle$

Wlog. $\mathsf{pexp} \equiv p$ and $q \in \mathsf{valid}_r$ and $a = m_{r\text{-}act}(\mathsf{pexp}) = \mathcal{m}_r(\mathsf{pexp}) = m_r(q)$ must hold. We then obtain $a \in m_r(\mathsf{valid}_r)$. This constitutes a contradiction, the case does not apply.

So $\mathsf{pexp} \in \mathsf{valid}_r$ must hold and thus $m_r(\mathsf{pexp}) \neq a$. That is, $act$ updates $\mathsf{pexp}$ to $a$ (with $a \neq \mathsf{seg}$). To do so, $act$ must be an assignment or an allocation. We conclude a contradiction as before. $\blacksquare$

Proof C.24 (Lemma B.44). Towards a contradiction, assume $\mathsf{fresh}_r \cap \mathsf{freed}_r \neq \emptyset$. Let $a \in \mathsf{fresh}_r$ and $a \in \mathsf{freed}_r$. The latter means that $\tau$ is of the form $\tau = \tau_1.\mathsf{act}_2$ with $act = \langle t, \mathsf{free}(a), \text{up} \rangle$. Then, by definition, $a \notin \mathsf{fresh}_{\tau_1\text{-}act}$. By monotonicity, this yields $a \notin \mathsf{fresh}_r$. Since this contradicts the assumption, we must have $\mathsf{fresh}_r \cap \mathsf{freed}_r = \emptyset$ as required.

Towards a contradiction, assume $\mathsf{fresh}_r \cap \mathsf{retired}_r \neq \emptyset$. Let $a \in \mathsf{fresh}_r$ and $a \in \mathsf{retired}_r$. The latter means that $\tau$ is of the form $\tau = \tau_1.\mathsf{act}_2$ with $act = \langle t, \mathsf{in}\text{-}\mathsf{retire}(p), \text{up} \rangle$ and $m_{\tau_1}(p) = a$. The contrapositive of Lemma B.42 gives $a \notin \mathsf{fresh}_{\tau_1\text{-}act}$. By monotonicity, we get $a \notin \mathsf{fresh}_r$. Since this contradicts the assumption, we must have $\mathsf{fresh}_r \cap \mathsf{retired}_r = \emptyset$ as required. $\blacksquare$

Proof C.25 (Lemma B.45). Let $a \in \mathsf{Adr}$ and $\varphi = \{ z_a \mapsto a \}$. The claim holds for $e$. Towards a contradiction, assume there is a shortest $\tau.\mathsf{act} \in \mathcal{O}[\mathcal{P}][\mathcal{Adr}]$ such that $(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau,\mathsf{act})} (L_3, \varphi)$ and $a \notin \mathsf{retired}_{\tau,\mathsf{act}}$. If $\mathcal{H}(\tau,\mathsf{act}) = \mathcal{H}(\tau)$, then we have $\mathsf{retired}_{\tau,\mathsf{act}} = \mathsf{retired}_r$. This contradicts the minimality of $\tau.\mathsf{act}$. So $\mathcal{H}(\tau,\mathsf{act})$ is of the form $\mathcal{H}(\tau,\mathsf{act}) = \mathcal{H}(\tau)$. That contradicts the assumption.

Case 1: $(L_2, \varphi) \xrightarrow{h} (L_3, \varphi)$

By definition of $\mathcal{O}_{\text{base}}$, there is not step $(L_1, \varphi) \xrightarrow{\mathsf{evt}} (L_3, \varphi)$. Hence, this case cannot apply.

Case 2: $(L_2, \varphi) \xrightarrow{h} (L_3, \varphi)$

We must have $(L_2, \varphi) \xrightarrow{\mathsf{evt}} (L_3, \varphi)$. This means $\mathsf{evt}$ is of the form $\mathsf{evt} = \mathsf{retire}(t, a)$ for some $t$. That is, $act = \langle t, \mathsf{retire}(p), \text{up} \rangle$ with $m_\tau(p) = a$. Thus, $a \in \mathsf{retired}_{\tau,\mathsf{act}}$, contradicting the assumption.

Case 3: $(L_2, \varphi) \xrightarrow{h} (L_3, \varphi)$

By minimality, we have $a \notin \mathsf{retired}_r$. To arrive at $a \notin \mathsf{retired}_r$, we must have $\mathsf{evt} = \mathsf{free}(a)$. This yields $(L_3, \varphi) \xrightarrow{\mathsf{evt}} (L_2, \varphi)$. So, $(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau,\mathsf{act})} (L_2, \varphi)$. This contradicts the assumption.

The above case distinction is complete and proves that $(L_2, \varphi) \xrightarrow{\mathcal{H}(\tau)} (L_3, \varphi)$ implies $a \in \mathsf{retired}_r$. Consider now the reverse direction. To that end, consider $\tau \in \mathcal{O}[\mathcal{P}][\mathcal{Adr}]$ and $a \notin \mathsf{retired}_r$. 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 $\tau \notin \mathcal{O}[\mathcal{P}][\mathcal{Adr}]$. Hence, $l = L_2$. The remaining follows analogously. $\blacksquare$
Proof C.26 (Lemma B.46). Let \( r.\text{act} \in \mathcal{O}[P]^{\text{adr}} \) with \( \text{act} = \{ t, \text{free}(a), \uparrow p \} \). Let \( \varphi = \{ z_a \mapsto a \} \).

We have \( \mathcal{H}(r.\text{act}) = h.\text{free}(a) \) with \( h = \mathcal{H}(\tau) \). By definition, we have \( h.\text{free}(a) \in S(O) \).

Since \( O = O_{\text{Base}} \times O_{\text{SMR}} \) by convention, we have \( h.\text{free}(a) \in S(O_{\text{Base}}) \). So \( (L_2, \varphi) \xrightarrow{\tau} (L_3, \varphi) \) as for otherwise \( \text{free}(a) \) would take \( O_{\text{Base}} \) to \( L_1 \) and thus give \( r.\text{act} \notin \mathcal{O}[P]^{\text{adr}} \). Now, Lemma B.45 yields the desired \( a \in \text{retired}_r \).

\[ \square \]

Proof C.27 (Lemma B.47). Let \( r.\text{act} \in \mathcal{O}[P]^{\text{adr}} \) with \( \text{act} = \{ t, \text{com}, \uparrow p \} \).

\( \diamond \) **Case 1**: \( \text{com} \equiv \text{exp} := \text{exp}' \)

We show \( m_r(\text{act}) \subseteq m_r(\text{valid}_r) \) and \( \text{adr}(m_r(\text{act})|\text{valid}_r) \subseteq \text{adr}(m_r|\text{valid}_r) \).

\( \diamond \) **Case 1.1**: \( \text{com} \equiv p := q.\text{next} \)

Let \( a = m_r(q) \). By definition, \( a \neq \text{seg} \). Let \( b = m_r(a.\text{next}) \). Hence, \( \uparrow p = [p \mapsto b] \).

If \( a.\text{next} \in \text{valid}_r \) and \( q \in \text{valid}_r \) we have:

\[
m_r(\text{valid}_r) = m_r(\text{valid}_r \setminus \{ p \}) \cup \{ m_r(\text{act}) \} = m_r(\text{valid}_r \setminus \{ p \}) \cup \{ b \} \subseteq m_r(\text{valid}_r) \cup \{ b \} = m_r(\text{valid}_r)
\]

where the last equality holds by \( b = m_r(a.\text{next}) \in m_r(\text{valid}_r) \). Otherwise, we get:

\[
m_r(\text{valid}_r) = m_r(\text{valid}_r \setminus \{ p \}) = m_r(\text{valid}_r \setminus \{ p \}) \subseteq m_r(\text{valid}_r).
\]

Altogether, we obtain \( m_r(\text{act}) \subseteq m_r(\text{valid}_r) \) as required. Combining this with

\[
\text{valid}_r(\text{act}) \cap \text{Adr} \subseteq (\text{valid}_r \cap \{ p \}) \cap \text{Adr} = \text{valid}_r \cap \text{Adr}
\]

yields \( \text{adr}(m_r(\text{act})|\text{valid}_r) \subseteq \text{adr}(m_r|\text{valid}_r) \) by Lemma B.35.

\( \diamond \) **Case 1.2**: \( \text{com} \equiv p := q \) or \( \text{com} \equiv p.\text{next} := q \)

Analogous to the previous case.

\( \diamond \) **Case 1.3**: \( \text{com} \in \{ u := \text{op}(\overline{u}), u := q.\text{data}, p.\text{data} := u \} \)

By definition, \( \text{valid}_r(\text{act}) = \text{valid}_r \) and \( m_r(p.\text{exp}) = m_r(\text{pexp}) \) for all \( \text{pexp} \in \text{PExp} \). So, we conclude \( m_r(\text{valid}_r(\text{act}) = m_r(\text{valid}_r) \) and \( \text{adr}(m_r(\text{act})|\text{valid}_r(\text{act}) = \text{adr}(m_r|\text{valid}_r) \).

\( \diamond \) **Case 2**: \( \text{com} \equiv p := \text{malloc} \)

Let \( a = m_r(p) \). The update is \( \uparrow p = [p \mapsto a, a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d] \) for some \( d \). We have \( \text{valid}_r(\text{act}) = \text{valid}_r \cup \{ p, a.\text{next} \} \). So, \( \text{valid}_r(\text{act}) \cap \text{Adr} = (\text{valid}_r \cap \text{Adr}) \cup \{ a \} \). We get:

\[
m_r(\text{valid}_r(\text{act}) = m_r(\text{valid}_r \setminus \{ p, a.\text{next} \}) \cup m_r(\{ p, a.\text{next} \}) = m_r(\text{valid}_r \setminus \{ p, a.\text{next} \}) \cup \{ a \} \subseteq m_r(\text{valid}_r) \cup \{ a \}.
\]

Combining the above yields \( \text{adr}(m_r(\text{act})|\text{valid}_r(\text{act}) \subseteq \text{adr}(m_r|\text{valid}_r) \) by Lemma B.35.
\[ \text{Case 3: } \text{com} \equiv \text{free}(a) \]

We have \( m_{\text{r,act}} = m_r \) and \( \text{valid}_{\text{r,act}} \subseteq \text{valid}_r \). Hence, \( m_{\text{r,act}}(\text{valid}_{\text{r,act}}) \subseteq m_r(\text{valid}_r) \). Moreover, \( \text{valid}_{\text{r,act}} \cap \text{Adr} \subseteq \text{valid}_r \cap \text{Adr} \). So \( \text{adr}(m_{\text{r,act}}|_{\text{valid}_{\text{r,act}}}) \leq \text{adr}(m_r|_{\text{valid}_r}) \) by Lemma B.35.

\[ \text{Case 4: } \text{com} \equiv \text{env}(a) \]

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 \( \text{valid}_{\text{r,act}} = \text{valid}_r \) and thus \( \text{valid}_{\text{r,act}} \cap \text{Adr} = \text{valid}_r \cap \text{Adr} \). Moreover, we get:

\[
\text{valid}_{\text{r,act}}(\text{valid}_{\text{r,act}}) \leq m_{\text{r,act}}(\text{valid}_r \setminus \{ a.\text{next} \}) \cup m_{\text{r,act}}(\{ a.\text{next} \})
\]

\[
= m_r(\text{valid}_r \setminus \{ a.\text{next} \}) \cup \emptyset \leq m_r(\text{valid}_r).
\]

Combining the above yields the desired \( \text{adr}(m_{\text{r,act}}|_{\text{valid}_{\text{r,act}}}) \leq \text{adr}(m_r|_{\text{valid}_r}) \) by Lemma B.35.

\[ \text{Case 5: otherwise} \]

We show \( m_{\text{r,act}}(\text{valid}_{\text{r,act}}) = m_r(\text{valid}_r) \) and \( \text{adr}(m_{\text{r,act}}|_{\text{valid}_{\text{r,act}}}) = \text{adr}(m_r|_{\text{valid}_r}) \).

\[ \text{Case 5.1: } \text{com} \in \{ \text{in:func}(\overline{\text{v}}), \text{re:func}, \text{skip}, \text{beginAtomic}, \text{endAtomic}, \text{@inv} \} \]

By definition, we have \( \text{valid}_{\text{r,act}} = \text{valid}_r \) and \( m_{\text{r,act}} = m_r \). Hence, we conclude the desired \( m_{\text{r,act}}(\text{valid}_{\text{r,act}}) = m_r(\text{valid}_r) \) and \( \text{adr}(m_{\text{r,act}}|_{\text{valid}_{\text{r,act}}}) = \text{adr}(m_r|_{\text{valid}_r}) \).

\[ \text{Case 5.2: } \text{com} \equiv \text{assume cond} \]

We have \( m_r(\text{valid}_r) = m_{\text{r,act}}(\text{valid}_{\text{r,act}}) \). This follows from \( \text{act} \) validating only pointers that are equal to already valid pointers. Further, \( \text{valid}_{\text{r,act}} \cap \text{Adr} = \text{valid}_r \cap \text{Adr} \) since \( \text{act} \) may only validate pointers. Hence, \( \text{adr}(m_{\text{r,act}}|_{\text{valid}_{\text{r,act}}}) = \text{adr}(m_r|_{\text{valid}_r}) \) by Lemma B.35.

The above case distinction is complete and concludes the claim. \[ \blacksquare \]

**Proof C.28 (Lemma B.48).** To the contrary, assume there is a shortest \( \text{r,act} \in \text{C}[\text{P}^{\text{Adr}}] \) such that there is an \( a \in \text{freed}_{\text{r,act}} \) with \( a \in m_{\text{r,act}}(\text{valid}_{\text{r,act}}) \) and \( a.\text{next} \notin \text{valid}_{\text{r,act}} \). Note that \( \text{r,act} \) is indeed the shortest such computation since the claim is vacuously true for \( \varepsilon \). First, consider the case where \( a \notin \text{freed}_r \). By minimality, \( a \notin m_r(\text{valid}_r) \) and \( a.\text{next} \notin \text{valid}_r \). If \( a.\text{next} \in \text{valid}_{\text{r,act}} \), then \( \text{act} \) must be an assignment of the form \( p.\text{next} := q \) with \( m_r(p) = a \). (Note that an allocation of \( a \) could validate \( a.\text{next} \) as well but would give \( a \notin \text{freed}_{\text{r,act}} \) and thus contradict the assumption.) Now, \( p \notin \text{valid}_r \) must hold because \( a \notin m_r(\text{valid}_r) \). Hence, \( \text{act} \) raises an unsafe access. This contradicts the assumption. So we must have \( a \in m_{\text{r,act}}(\text{valid}_{\text{r,act}}) \). Recall that we have \( a \notin m_r(\text{valid}_r) \). Along the lines of Proof C.23 of Lemma B.43, this means that \( \text{act} \) must be an allocation of \( a \). So, \( a \notin \text{freed}_{\text{r,act}} \) follows by definition. As this contradicts the assumption, the case cannot apply. Altogether, we must have \( a \notin \text{freed}_r \). Then, \( \text{act} \) must execute \( \text{free}(a) \).

We get \( a.\text{next} \notin \text{valid}_{\text{r,act}} \) and \( \text{pexp} \notin \text{valid}_{\text{r,act}} \). For all \( \text{pexp} \) that satisfy \( m_r(\text{pexp}) = a \). That is, we obtain \( a \notin m_{\text{r,act}}(\text{valid}_{\text{r,act}}) \) and \( a.\text{next} \notin \text{valid}_{\text{r,act}} \). This contradicts the assumption. \[ \blacksquare \]
Proof C.29 (Lemma B.49). We show the following proposition:

\[ \forall \tau, \text{pexp} \cdot \tau \text{ UAF} \land m_\tau(\text{pexp}) = \text{seg} \land \text{pexp} \notin \text{valid}_\tau \]

\[ \implies (\text{pexp} \cap \text{Adr}) \cap (\text{fresh}_\tau \cup \text{freed}_\tau) \neq \emptyset \]

which implies the lemma. To see this, let \( \tau \text{ UAF} \) and \( \text{pexp} \in V\text{Exp}(\tau) \) with \( m_\tau(\text{pexp}) = \text{seg} \). To the contrary, assume \( \text{pexp} \notin \text{valid}_\tau \). Then the above yields \( (\text{pexp} \cap \text{Adr}) \cap (\text{fresh}_\tau \cup \text{freed}_\tau) \neq \emptyset \).

Hence, \( \text{pexp} \) must be of the form \( \text{pexp} \equiv a_.\text{next} \) with \( a \in \text{fresh}_\tau \cup \text{freed}_\tau \). Lemmas B.42 and B.48 give \( a \notin m_\tau(\text{valid}_\tau) \). This means \( a_.\text{next} \notin V\text{Exp}(\tau) \). Because this contradicts \( \text{pexp} \in V\text{Exp}(\tau) \), we must have \( \text{pexp} \in \text{valid}_\tau \) as required.

We now show the above proposition. To the contrary, assume there is a shortest \( \tau.\text{act} \in \mathcal{O}[P]_{\text{Adr}} \) UAF with \( m_{\tau.\text{act}}(\text{pexp}) = \text{seg} \) and \( \text{pexp} \notin \text{valid}_{\tau.\text{act}} \) and \( (\text{pexp} \cap \text{Adr}) \cap (\text{fresh}_{\tau.\text{act}} \cup \text{freed}_{\tau.\text{act}}) = \emptyset \).

First, consider \( \text{pexp} \in \text{PVar} \). Then, we obtain \( (\text{pexp} \cap \text{Adr}) \cap (\text{fresh}_\tau \cup \text{freed}_\tau) = \emptyset \) because of \( \text{pexp} \cap \text{Adr} = \emptyset \). By minimality of \( \tau.\text{act} \), we must have \( m_{\tau}(\text{pexp}) = \text{seg} \) or \( \text{pexp} \in \text{valid}_\tau \). In the former case, \( \text{act} \) executes an assignment of the form \( \text{pexp} := q\text{expr} \) with \( m_{\tau}(\text{qexp}) = \text{seg} \). To arrive at \( \text{pexp} \notin \text{valid}_{\tau.\text{act}} \), we must have \( \text{qexp} \in \text{PVar} \setminus \text{valid}_\tau \) or \( \text{qexp} \equiv q_.\text{next} \land q \notin \text{valid}_\tau \) or \( \text{qexp} \equiv q_.\text{next} \land m_{\tau}(q)_.\text{next} \notin \text{valid}_\tau \). By minimality and \( \text{PVar} \cap \text{Adr} = \emptyset \), the first case cannot apply. The second case cannot apply because it raises an unsafe access, contradicting the assumption that \( \tau.\text{act} \) is UAF. So, \( \text{qexp} \equiv q_.\text{next} \) with \( m_{\tau}(q) = a \) and \( a_.\text{next} \notin \text{valid}_\tau \). This means \( \text{qexp} \cap \text{Adr} = \{ a \} \). We get \( a \in \text{fresh}_\tau \cup \text{freed}_\tau \) by minimality. Then, Lemmas B.43 and B.48 yield \( a \notin m_{\tau}(\text{valid}_\tau) \). That is, \( q \notin \text{valid}_\tau \). Consequently, \( \text{act} \) raises an unsafe access, contradicting the UAF assumption. Hence, the case cannot apply; we must have \( m_{\tau}(\text{pexp}) = \text{seg} \) and \( \text{pexp} \in \text{valid}_\tau \). In order for \( \text{act} \) to invalidate \( \text{pexp} \), it must assign \( \text{pexp} \) from an invalid seg-holding expression (note that value \text{seg} cannot be the target of a \text{free} command).

We conclude a contradiction to \( \tau.\text{act} \) UAF as before. Altogether, \( \text{pexp} \in \text{PVar} \) cannot hold. We must have \( \text{pexp} \notin \text{PVar} \).

Consider now \( \text{pexp} \notin \text{PVar} \). That is, \( \text{pexp} \equiv a_.\text{next} \) for some address \( a \in \text{Adr} \). By definition, we get \( \text{pexp} \cap \text{Adr} = \{ a \} \). By minimality, \( m_{\tau}(\text{pexp}) \neq \text{seg} \) or \( \text{pexp} \in \text{valid}_\tau \), or \( a \in \text{fresh}_\tau \cup \text{freed}_\tau \).

In the latter case, \( \text{act} \) must allocate \( a \) to arrive at \( (\text{pexp} \cap \text{Adr}) \cap (\text{fresh}_{\tau.\text{act}} \cup \text{freed}_{\tau.\text{act}}) = \emptyset \). This, however, validates \( a_.\text{next} \), contradicting \( \text{pexp} \notin \text{valid}_{\tau.\text{act}} \). Hence, we have \( a \notin \text{fresh}_\tau \cup \text{freed}_\tau \).

If we have \( \text{pexp} \in \text{valid}_\tau \), then \( \text{act} \) must invalidate \( \text{pexp} \). To do this, \( \text{act} \) must execute \text{free}(a) or \( p_.\text{next} := q \) with \( m_{\tau}(p) = a \). In the former case, we get \( a \in \text{freed}_{\tau.\text{act}} \). Hence, we arrive at \( (\text{pexp} \cap \text{Adr}) \cap (\text{fresh}_{\tau.\text{act}} \cup \text{freed}_{\tau.\text{act}}) = \{ a \} \), contradicting the assumption. That is, the case cannot apply so that \( \text{act} \) must be an assignment. To get \( m_{\tau.\text{act}}(\text{pexp}) = \text{seg} \) and \( \text{pexp} \notin \text{valid}_{\tau.\text{act}} \), we must have \( m_{\tau}(q) = \text{seg} \) and \( q \notin \text{valid}_\tau \). Since we have \( q \in \text{PVar} \), \( q \cap \text{Adr} = \emptyset \) follows. That is, \( (q \cap \text{Adr}) \cap (\text{fresh}_\tau \cup \text{freed}_\tau) = \emptyset \). Altogether, this contradicts the minimality of \( \tau.\text{act} \).  


Proof C.30 (Lemma B.50). We show the following proposition:

$$\forall \tau, pexp. \ \tau \text{ UAF } \land m_\tau(pexp) = \bot \quad \implies \{ pexp \} \cap \text{Adr } \notin \{ \text{valid}_\tau \}$$

which implies the lemma. To see this, consider $\tau$ UAF and $pexp \in \text{VExp}(\tau)$. To the contrary, assume $pexp \notin \text{dom}(m_\tau)$. That is, $m_\tau(pexp) = \bot$. The above then yields $pexp \cap \text{Adr } \notin \text{valid}_\tau$. Because $\emptyset \subseteq m_\tau(\text{valid}_\tau)$ trivially holds, we must have $pexp \cap \text{Adr } = \{ a \}$ for some address $a$ with $a \notin m_\tau(\text{valid}_\tau)$. This means $pexp \equiv a.\text{next}$. Hence, $pexp \notin \text{VExp}(\tau)$ by definition. Because this contradicts the choice of $pexp$, we must have $pexp \in \text{dom}(m_\tau)$ as required.

We now establish the above proposition. To the contrary, assume a shortest $\tau.\text{act} \in O[P]^{\text{Adr}}$ UAF such that there is some $pexp$ with $m_{\tau.\text{act}}(pexp) = \bot$ and $\{ pexp \} \cap \text{Adr} \subseteq m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}})$. By minimality, $m_{\tau}(pexp) \neq \bot$ or $\{ pexp \} \cap \text{Adr} \not\subseteq m_{\tau}(\text{valid}_{\tau})$. First, consider $m_{\tau}(pexp) \neq \bot$. In order to arrive at $m_{\tau.\text{act}}(pexp) = \bot$, action $\text{act}$ must execute an assignment $pexp := qexp$ with $m_{\tau}(qexp) = \bot$. If $qexp \in \text{PVar}$, then we have $\{ qexp \} \cap \text{Adr} = \emptyset \subseteq m_{\tau}(\text{valid}_{\tau})$. 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 $\text{act}$ raises an unsafe access. This contradicts the assumption of $\tau.\text{act}$ being UAF. Altogether, we get $m_{\tau}(pexp) = \bot$ and thus we must have $\{ pexp \} \cap \text{Adr} \notin m_{\tau}(\text{valid}_{\tau})$.

Consider the case $\{ pexp \} \cap \text{Adr} \notin m_{\tau}(\text{valid}_{\tau})$ now. Observe that $\text{PVVar} \cap \text{Adr} = \emptyset \subseteq m_{\tau}(\text{valid}_{\tau})$. 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 $\{ pexp \} \cap \text{Adr} \subseteq m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}})$ we must have $a \in m_{\tau.\text{act}}(\text{valid}_{\tau.\text{act}})$. That is, $\text{act}$ 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, $\text{act}$ must perform an allocation of $a$ to do that:

$$\text{act} = \langle t, p := \text{malloc}([p \mapsto a, a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d]) \rangle$$

for some $d$. However, this results in $m_{\tau.\text{act}}(pexp) = \bot$, contradicting the assumption. ■

Proof C.31 (Lemma B.51). We proceed by induction over the structure of $\tau$. In the base case, $\tau = \epsilon$. Then, the claim follows from $\text{VExp}(\epsilon) = \text{PVVar} \subseteq \text{valid}_\tau$ by definition. For the induction step, consider some $\tau.\text{act} \in O[P]^{\text{Adr}}$ UAF and assume we have already establish that $pexp \notin \text{valid}_{\tau}$ implies $m_\tau(pexp) \in \text{frees}_\tau$. Let $pexp \in \text{VExp}(\tau.\text{act})$ with $pexp \notin \text{valid}_{\tau.\text{act}}$. We now establish that $m_{\tau.\text{act}}(pexp) \in \text{frees}_{\tau.\text{act}}$ holds. Let $\text{act} = \langle t, \text{com}, \text{up} \rangle$.

• Case 1: $\text{com} \equiv p := q.\text{next}$
  
  Let $a = m_\tau(q)$. By definition, we have $a \neq \text{seg}$. Let $b = m_\tau(a.\text{next})$. So, $\text{up} = [p \mapsto b]$. First, consider the case $pexp \equiv p$. By definition, $pexp \in \text{VExp}(\tau)$. Further, $q \in \text{valid}_{\tau}$ since $\tau.\text{act}$ 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 $m_\tau(a.\text{next}) \in \text{frees}_\tau = \text{frees}_{\tau.\text{act}}$. Hence, $m_{\tau.\text{act}}(p) \in \text{frees}_{\tau.\text{act}}$ as required.
Consider now $pexp \not\equiv p$. Then, $pexp \not\in \text{valid}_r$ by definition and. Moreover, Lemma B.47 yields $m_{r, \text{act}}(\text{valid}_{r, \text{act}}) \subseteq m_r(\text{valid}_r)$. Hence, we obtain $pexp \in VExp(\tau)$. By induction, we get $m_r(pexp) \in \text{frees}_r$. By definition, this means $m_{r, \text{act}}(pexp) \in \text{frees}_{r, \text{act}}$ as required.

**Case 2:** $\text{com} \equiv p := q$ or $\text{com} \equiv p\text{.next} := q$

Analogous to the previous case.

**Case 3:** $\text{com} \equiv u := \text{op}(\bar{u})$

By definition, $\text{frees}_r = \text{frees}_{r, \text{act}}$ and $\text{valid}_r = \text{valid}_{r, \text{act}}$. Moreover, $m_r(pexp) = m_{r, \text{act}}(pexp)$ for all $pexp \in \text{PExp}$. We obtain $VExp(\tau) = VExp(\tau.\text{act})$. Hence, we conclude by induction.

**Case 4:** $\text{com} \in \{ u := q.\text{data}, p.\text{data} := u \}$

Analogous to the previous case.

**Case 5:** $\text{com} \in \{ \text{in}:\text{func}(\bar{v}), \text{re}:\text{func}, \text{skip}, \text{beginAtomic}, \text{endAtomic}, @\text{inv} \bullet \}$

Analogous to the previous case.

**Case 6:** $\text{com} \equiv p := \text{malloc}$

Let $a = m_{r, \text{act}}(p)$. The update is $u = [p \mapsto a, a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d]$ for some $d$. We have $p, a.\text{next} \in \text{valid}_{r, \text{act}}$ by definition. So, $pexp \not\in \{ p, a.\text{next} \}$ and $pexp \not\in \text{valid}_r$. Then, we obtain $m_{r, \text{act}}(pexp) = m_r(pexp)$. Moreover, we get $m_{r, \text{act}}(\text{valid}_{r, \text{act}}) \subseteq m_r(\text{valid}_r) \cup \{ a \}$ by Lemma B.47. Hence, $pexp \in VExp(\tau)$ because $pexp \not\equiv a.\text{next}$. So by induction, we obtain the desired $m_{r, \text{act}}(pexp) = m_r(pexp) \in \text{frees}_r = \text{frees}_{r, \text{act}}$.

**Case 7:** $\text{com} \equiv \text{assume cond}$

By definition, we have $\text{frees}_r = \text{frees}_{r, \text{act}}$ and $m_r(\text{valid}_r) = m_{r, \text{act}}(\text{valid}_{r, \text{act}})$. The latter follows from the fact that $\text{act}$ validates only pointers that are equal to already valid pointers. Hence, we obtain $pexp \in VExp(\tau)$. Then, we conclude $m_{r, \text{act}}(pexp) = m_r(pexp) \in \text{frees}_r = \text{frees}_{r, \text{act}}$ by induction.

**Case 8:** $\text{com} \equiv \text{free}(a)$

We have $m_{r, \text{act}} = m_r$ and $\text{valid}_{r, \text{act}} \subseteq \text{valid}_r$ by definition. Further, we get that $m_r(qexp) = \tau$ implies $qexp \not\in \text{valid}_{r, \text{act}}$ for all $qexp$. That is, we have $a \not\in m_{r, \text{act}}(\text{valid}_{r, \text{act}}) \subseteq m_r(\text{valid}_r)$. Hence, $pexp \not\in VExp(\tau) \setminus \{ a.\text{next} \}$. Moreover, $\text{frees}_{r, \text{act}} = \text{frees}_r \cup \{ a \}$. If $pexp \not\in \text{valid}_r$, then we conclude by induction. Otherwise, we have $pexp \in \text{valid}_r$. That is, $pexp$ is invalidated by $\text{act}$. This means $pexp \equiv a.\text{next}$ or $m_r(pexp) = a$. Since we already showed that the former case cannot apply, we have $m_r(pexp) = a$. Then, $m_{r, \text{act}}(pexp) = a \in \text{frees}_{r, \text{act}}$. 

---

**Appendix C** Proof of Meta Theory
\textbf{Case 9:} \( \text{com} \equiv \text{env}(a) \)

The update takes the form \( \text{up} = [a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d] \) for some data value \( d \). By definition, we have \( \text{valid}_{r,\text{act}} = \text{valid}_r \) and so we get:

\[
m_{r,\text{act}}(\text{valid}_{r,\text{act}}) = m_{r,\text{act}}(\text{valid}_r) \subseteq m_{r,\text{act}}(\text{valid}_r \setminus \{a.\text{next}\}) \cup m_{r,\text{act}}(\{a.\text{next}\})
\]

\[= m_r(\text{valid}_r \setminus \{a.\text{next}\}) \cup \emptyset \subseteq m_r(\text{valid}_r).\]

This means \( \text{pexp} \in \text{VExp}(r) \) and \( \text{pexp} \notin \text{valid}_r \). By induction, \( m_r(\text{pexp}) \in \text{frees}_r = \text{frees}_{r,\text{act}} \).

Because \( a \in \text{fresh}_r \cup \text{freed}_r \) according to the semantics, we get \( a \notin m_r(\text{valid}_r) \) by Lemmas B.43 and B.48. That is, \( a\text{.next} \notin \text{VExp}(r) \) by definition. Thus, \( \text{pexp} \notin a\text{.next} \).

By Assumption A.9, we have \( \text{Proof C.34} \). By the update, we get \( m_r(\text{pexp}) = m_{r,\text{act}}(\text{pexp}) \). We arrive at the desired \( m_{r,\text{act}}(\text{pexp}) \in \text{frees}_{r,\text{act}} \).

The above case distinction is complete and concludes the induction. \hfill \( \blacksquare \)

\textbf{Proof C.32} (Lemma B.52). Let \( \text{pexp} \in \text{VExp}(r) \setminus \text{valid}_r \). By Lemma B.51, \( m_r(\text{pexp}) = a \in \text{frees}_r \) for some address \( a \). Since \( r \in \mathcal{O}[P]^{\text{adr}}_{\text{adr}} \), address \( a \) remains free once it is freed, it cannot be reallocated. Hence, \( m_r(\text{pexp}) \in \text{frees}_r \) follows as required. \hfill \( \blacksquare \)

\textbf{Proof C.33} (Lemma B.53). Let \( r \in \mathcal{O}[P]^{\text{adr}}_{\text{adr}} \UAF \). Consider some \( a \in \text{Adr} \) with \( a \in \text{adr}(m_r|\text{valid}_r) \) and \( a \notin m_r(\text{VExp}(r) \setminus \text{valid}_r) \). We show \( a \in A \). The former and Lemma B.35 yields \( a \in \text{valid}_r \cap \text{Adr} \) or \( a \in m_r(\text{valid}_r) \).

Note that \( a \in \text{valid}_r \cap \text{Adr} \) implies \( a\text{.next} \in \text{valid}_r \). Hence, the contrapositive of Lemma B.48 gives \( a \notin \text{frees}_r \).

Altogether, this means that \( a \) has been freed and reallocated in \( r \). That is, \( a \in A \) must hold. \hfill \( \blacksquare \)

\textbf{Proof C.34} (Lemma B.54). Let \( r \in \mathcal{O}[P]^{\text{adr}}_{\text{adr}} \UAF \). Consider some \( C \in \text{CVar} \) and \( p \in \text{PVar} \setminus \text{valid}_r \).

By Assumption A.9, we have \( m_r(C) \notin \text{frees}_r \cup \text{retired}_r \). By definition, \( p \in \text{VExp}(r) \setminus \text{valid}_r \).

Then, Lemma B.51 yields \( m_r(p) \in \text{frees}_r \). Hence, \( m_r(C) \neq m_r(p) \) must hold as required. \hfill \( \blacksquare \)

\textbf{Proof C.35} (Lemma B.55). Let \( \mathcal{O} \) support elision. Let \( r \in \mathcal{O}[P]^{\text{adr}}_{\text{adr}} \) with \( \mathcal{H}(r) = h \).

By the semantics, we have \( h \in \mathcal{S}(\mathcal{O}_{\text{Base}}) \) and can thus invoke Definition 7.14 with \( h \). Let \( a, b, c \in \text{Adr} \) be addresses with \( a \neq b \neq c \). 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, \( \mathcal{O} = \mathcal{O}_{\text{Base}} \times \mathcal{O}_{\text{SMR}} \). Hence, \( \mathcal{F}_{\mathcal{O}_{\text{SMR}}}(h, c) = \mathcal{F}_{\mathcal{O}_{\text{SMR}}}(h[a/b], c) \) by Property (i) of Definition 7.14. We get \( \mathcal{F}_{\mathcal{O}_{\text{Base}}}(h, c) = \mathcal{F}_{\mathcal{O}_{\text{Base}}}(h[a/b], c) \) by definition of \( \mathcal{O}_{\text{Base}} \).

To see this, consider some \( h' \in \mathcal{F}_{\mathcal{O}_{\text{Base}}}(h, c) \). By definition, \( \text{frees}_{h'} \subseteq \{c\} \) and \( h.h' \in \mathcal{S}(\mathcal{O}_{\text{Base}}) \).

Now, consider \( h' \in \mathcal{F}_{\mathcal{O}}(h, c) \). By definition, we have \( \text{frees}_{h'} \subseteq \{c\} \) and \( h.h' \in \mathcal{S}(\mathcal{O}) \).

The former gives \( \text{frees}_{h'[a/b]} \subseteq \{c\} \). Because of \( \mathcal{O} = \mathcal{O}_{\text{Base}} \times \mathcal{O}_{\text{SMR}} \), we have both \( h.h' \in \mathcal{S}(\mathcal{O}_{\text{Base}}) \).
and $h.h' \in S(O_{SMR})$. Then, $h' \in F_{O_{bas}}(h, c)$ and $h' \in F_{O_{bas}}(h, c)$. So $h' \in F_{O_{bas}}(h[a/b], c)$ and $h' \in F_{O_{bas}}(h[a/b], c)$. That is, $h.h'[a/b] \in S(O_{bas})$ as well as $h.h'[a/b] \in S(O_{SMR})$.

Finally, we arrive at $h.h'[a/b] \in S(O)$. That is, we have $h'[a/b] \in F_{O}(h, c)$. Altogether, we have established $F_{O}(h, c) \in F_{O}(h[a/b], c)$. The reverse direction follows analogously.

**Proof C.36** (Lemma B.56). Let $O$ support elision. Let $r \in O[\mathcal{P}]^{Adr}$ with $\mathcal{H}(r) = h$. By the semantics, we have $h \in S(O_{bas})$ and can thus invoke Definition 7.14 with $h$. Let $a \neq b$. Assume we have $h._{free}(a) \in S(O)$. By Assumption 5.6, $O = O_{bas} \times O_{SMR}$. Elision support, Property (iii) of Definition 7.14, gives $F_{O_{bas}}(h._{free}(a), b) = F_{O_{bas}}(h, b)$. To conclude, it remains to show $F_{O_{bas}}(h._{free}(a), b) = F_{O_{bas}}(h, b)$. The inclusion $F_{O_{bas}}(h._{free}(a), b) \subseteq F_{O_{bas}}(h, b)$ follows from the definition of $O_{bas}$. The interesting case is the reverse inclusion. Consider some $h' \in F_{O_{bas}}(h, b)$. By definition, $frees_{h'} \subseteq \{b\}$ and $h' \in S(O_{bas})$. The latter means that for all addresses $c$ and all steps $(L_{2}, \varphi) \triangleright (l_{1}, \varphi) \overset{b}{\triangleright} (l_{2}, \varphi)$ of $O$ with $\varphi = \{z_{a} \mapsto c\}$ we have that $l_{2}$ is not accepting. Observe that $h._{free}(a) \in S(O)$ means $l_{1}$ is not accepting as well. If we have $c \neq a$, then $(l_{1}, \varphi) \overset{\text{free}(a)}{\triangleright} (l_{1}, \varphi)$. So we get $(L_{2}, \varphi) \overset{\text{free}(a)}{\triangleright} (l_{2}, \varphi)$ by Assumption 5.2. Otherwise, there is a program step $(l_{1}, \varphi) \overset{\text{free}(a)}{\triangleright} (l_{1}, \varphi)$ for some $l_{1}$. Note that location $l_{1}$ is not accepting because we have $h._{free}(a) \in S(O)$. Now, let $l_{2}$ be any location with $(l_{1}, \varphi) \overset{b}{\triangleright} (l_{2}, \varphi)$. By definition, we know that only a free event can take $O_{bas}$ to an accepting location for $\varphi$. That is, $l_{1}$ is not accepting because of $a \notin frees_{h'}$. Altogether, we obtain $h._{free}(a).h' \in S(O_{bas})$ in any case. That is, we arrive at $h' \in F_{O_{bas}}(h._{free}(a), b)$. We conclude the desired $F_{O_{bas}}(h._{free}(a), b) = F_{O_{bas}}(h, b)$.

**Proof C.37** (Lemma B.57). Let $r, \sigma \in O[\mathcal{P}]^{Adr}$. Let $h = \mathcal{H}(r)$ and let $h' = \mathcal{H}(\sigma)$. Let $a, b \in Adr$. By assumption, we have $F_{O}(h, a) \subseteq F_{O}(h', a)$ as well as $b \notin fresh_{h}$ and $b \in fresh_{h'}$. We now establish that $F_{O}(h, b) \subseteq F_{O}(h', b)$ holds. By Assumption 5.6, $O = O_{bas} \times O_{SMR}$. By elision support, Property (ii) of Definition 7.14, we have $F_{O_{bas}}(h, b) \subseteq F_{O_{bas}}(h', b)$. In order to conclude, it remains to show $F_{O_{bas}}(h, b) \subseteq F_{O_{bas}}(h', b)$. Consider $h \in F_{O_{bas}}(h, b)$. By definition, we have:

$$frees_{h} \subseteq \{b\} \quad \text{and} \quad h.h \in S(O_{bas})$$

Let $c$ be some address and let $\varphi = \{z_{a} \mapsto c\}$. Consider the following steps:

$$(L_{2}, \varphi) \overset{a}{\triangleright} (l_{1}, \varphi) \overset{b}{\triangleright} (l_{2}, \varphi) \quad \text{and} \quad (L_{2}, \varphi) \overset{\text{fresh}(c)}{\triangleright} (l_{1}', \varphi) \overset{b}{\triangleright} (l_{2}', \varphi)$$.

From $h.h \in S(O_{bas})$ 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 $c \notin frees_{h}$ and the fact that only free$(c)$ can take $O_{bas}$ to an accepting location for $\varphi$. That is, we
have \( l_1 \in \{ L_2, L_3 \} \). Recall that \( b \notin \text{retire}_b \). This means that, every \( \text{in: retire}(b, b) \) event is followed by an \( \text{free}(b) \) event. Hence, \( l_1 \neq L_3 \) according to the transitions in \( O_{\text{Base}} \). 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.

\[ \triangle \text{Case 2: } b \neq c \]

Assume for a moment that \( l'_1 \) is not accepting. Then \( l_2 \) is not accepting because \( c \notin \text{frees}_i \) and only \( \text{free}(c) \) can take \( O_{\text{Base}} \) to an accepting location for \( \varphi \). So it remains to show that location \( l'_1 \) is not accepting indeed. To that end, recall \( h' = \mathcal{H}(\sigma) \) and \( \sigma \in \mathcal{O}[P]_{\text{Adr}}^{\hat{\omega}} \). As argued before, \( O_{\text{Base}} \) reaches its final location \( L_1 \) only upon events \( \text{evt} \) of the form \( \text{evt} = \text{free}(b) \) and cannot leave \( L_1 \) afterwards. So, if \( h' \notin \mathcal{S}(O_{\text{Base}}) \) was true, then there is decomposition of \( \sigma \) that takes the form \( \sigma = \sigma_1 . \text{act} . \sigma_2 \) with \( \text{act} = \langle t, \text{free}(b), \text{up} \rangle \) and \( \mathcal{H}(\sigma_1 . \text{act}) \notin \mathcal{S}(O_{\text{Base}}) \). The latter means \( \mathcal{H}(\sigma_1 . \text{act}) \notin \mathcal{S}(O) \). This contradicts the enabledness of \( \sigma_1 . \text{act} \) and thus contradicts \( \sigma \in \mathcal{O}[P]_{\text{Adr}}^{\hat{\omega}} \). Hence, \( h' \in \mathcal{S}(O_{\text{Base}}) \). That is, \( h'_1 \) is not accepting, as required.

That \( h'_2 \) is not accepting for any \( \varphi \) means \( h'_1 . h \in \mathcal{S}(O_{\text{Base}}) \). Hence, \( h \in \mathcal{F}_{O_{\text{Base}}}(h'_1, b) \) as required.

\[ \blacksquare \]

Proof C.38 (Lemma B.58). Follows from the fact that \( \text{swap}_{\text{adr}} \) is a bijection.

Proof C.39 (Lemma B.59). To the contrary, assume there is a shortest history \( h . \text{evt} \) such that there are \( h_1 . \text{evt}_1 \neq h_2 . \text{evt}_2 \) with \( \text{swap}_{\text{adr}}(h_1 . \text{evt}_1) = h . \text{evt} = \text{swap}_{\text{adr}}(h_2 . \text{evt}_2) \). Note that \( h_1 . \text{evt}_1 \) and \( h_2 . \text{evt}_2 \) and \( h . \text{evt} \) all have the same length. Also note that \( h . \text{evt} \) is indeed a shortest such history because the claim holds for \( \varepsilon \). Let \( \text{evt} \) be of the form \( \text{evt} \equiv \text{in: func}(t, \overline{\sigma}) \). By choice, we have \( \text{swap}_{\text{hist}}(\text{evt}_i) = \text{evt} \). That is, \( \text{evt}_i \) must be of the form \( \text{evt}_i \equiv \text{in: func}(t, \overline{\sigma}_i) \) with \( \overline{\sigma} = \text{swap}_{\text{adr}}^{-1}(\overline{\sigma}_1) \). So \( \overline{\sigma}_1 = \text{swap}_{\text{adr}}(\overline{\sigma}) = \overline{\sigma}_2 \). Hence, \( \text{evt}_1 = \text{evt}_2 \). Moreover, \( h_1 = h_2 \) by minimality of \( h . \text{evt} \). Altogether, we obtain \( h_1 . \text{evt}_1 = h_2 . \text{evt}_2 \) which contradicts the assumption. The remaining cases follow analogously.

Proof C.40 (Lemma B.60). If \( h \in H \), then \( h' \in \text{swap}_{\text{hist}}(H) \) for \( \text{swap}_{\text{hist}}(h) = h' \) by definition. For the reverse direction, we know that there is some \( h \in H \) with \( \text{swap}_{\text{hist}}(h) = h' \). Lemma B.59 yields \( h = h' \). This yields \( h \in H \) as desired.

Proof C.41 (Lemma B.61). Consider \( a' \in \text{Adr} \). Since \( \text{swap}_{\text{adr}} \) is a bijection, there is \( a \in \text{Adr} \) such that \( a' = \text{swap}_{\text{adr}}(a) \) and \( a' \notin \text{swap}_{\text{adr}}(\text{Adr} \setminus \{ a \}) \). Hence, \( a \notin A_1 \Rightarrow a' \notin \text{swap}_{\text{adr}}(A_1) \). Moreover, \( a \in A_1 \Rightarrow a' \in \text{swap}_{\text{adr}}(A_1) \) by choice of \( a \). So \( a' \in \text{swap}_{\text{adr}}(A_1) \Leftrightarrow a \in A_1 \). Similarly, \( a' \in \text{swap}_{\text{adr}}(A_2) \Leftrightarrow a \in A_2 \). Thus, \( a' \in \text{swap}_{\text{adr}}(A_1) \cap \text{swap}_{\text{adr}}(A_2) \Leftrightarrow a \in A_1 \cap A_2 \). With the same arguments we get \( a \in A_1 \cap A_2 \Leftrightarrow a' \in \text{swap}_{\text{adr}}(A_1 \cap A_2) \), concluding the first equivalence.

As before, we have

\[ a'.\text{next} \in \text{swap}_{\text{exp}}(B_1) \Leftrightarrow a.\text{next} \in B_1 \quad \text{and} \quad a'.\text{next} \in \text{swap}_{\text{exp}}(B_2) \Leftrightarrow a.\text{next} \in B_2 \]
and derive \( a'.next \in \text{swap}_{exp}(B_1) \otimes \text{swap}_{exp}(B_2) \iff a.next \in B_1 \otimes B_2 \). And with the same arguments we get \( a.next \in B_1 \otimes B_2 \iff a'.next \in \text{swap}_{exp}(B_1 \otimes B_2) \). This concludes the second equivalence.

Consider now some history \( h' \). Since \( \text{swap}_{adr} \) is a bijection, there is \( h \) with \( \text{swap}_{hist}(h) = h' \). Then, Lemma B.60 yields \( h' \in \text{swap}_{hist}(C_1) \iff h \in C_1 \). Similarly, \( h' \in \text{swap}_{hist}(C_2) \iff h \in C_2 \). We arrive at \( h' \in \text{swap}_{hist}(C_1) \otimes \text{swap}_{hist}(C_2) \iff h \in C_1 \otimes C_2 \). Finally, Lemma B.60 yields that \( h \in C_1 \otimes C_2 \iff h' \in \text{swap}_{hist}(C_1 \otimes C_2) \) holds. This concludes the third equivalence. ■

**Proof C.42** (Lemma B.62). Consider some event \( evt \equiv \text{in}: \text{func}(t, \overline{v}) \). Then we have:

\[
\text{swap}_{hist}^{-1}(\text{swap}_{hist}(evt)) = \text{swap}_{hist}^{-1}(\text{swap}_{hist}(\text{in}: \text{func}(t, \overline{v}))) = \text{swap}_{hist}^{-1}(\text{in}: \text{func}(t, \text{swap}_{adr}(\overline{v})))
\]

\[
= \text{in}: \text{func}(t, \text{swap}_{hist}^{-1}(\text{swap}_{adr}(\overline{v}))) = \text{in}: \text{func}(t, \overline{v})
\]

Analogously, for \( \text{free}(a) \) and \( \text{re}: \text{func}(t) \). The overall claim follows then from inductively applying the above to \( h \). In the base case, we have \( h = \epsilon \) and \( \text{swap}_{hist}^{-1}(\text{swap}_{hist}(\epsilon)) = \epsilon \) by definition. For \( h.evt \) one has

\[
\text{swap}_{hist}^{-1}(\text{swap}_{hist}(h.evt)) = \text{swap}_{hist}^{-1}(\text{swap}_{hist}(h).\text{swap}_{hist}(evt))
\]

\[
= \text{swap}_{hist}(h).\text{swap}_{hist}^{-1}(\text{swap}_{hist}(evt)) = h.evt
\]

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:

\[
(l, \varphi) \xrightarrow{\delta} (l', \varphi) \iff (l, \text{swap}_{adr} \circ \varphi) \xrightarrow{\text{swap}_{hist}(h)} (l', \text{swap}_{adr} \circ \varphi).
\]  

(8)

To that end, let \( (l, \varphi) \xrightarrow{\text{evt}} (l', \varphi) \) be some SMR automaton step and let \( \overline{z} = \text{dom}(\varphi) \) be the SMR automaton variables. We show that also

\[
(l, \varphi) \xrightarrow{\text{swap}_{hist}(\text{evt})} (l', \varphi)
\]

with \( \varphi' = \text{swap}_{adr} \circ \varphi \)

is an SMR automaton step. We focus on events of the form \( \text{evt} \equiv \text{in}: \text{func}(t, \overline{v}) \); the remaining cases follow analogously. By the definition, there is a transition

\[
l \xrightarrow{\text{func}(\varphi), \varrho} l'
\]

such that \( \varrho[\overline{\tau} \mapsto \overline{v}, \overline{z} \mapsto \varphi'(\overline{z})] \models \text{true} \).

Now, turn to \( \text{swap}_{hist}(\text{evt}) \). It takes the form \( \text{swap}_{hist}(\text{evt}) \equiv \text{in}: \text{func}(\overline{w}) \) with \( \overline{w} = \text{swap}_{adr}(\overline{v}) \).

Hence, the above transition matches. We show that it is also enabled. To that end, we need show that the guard evaluates to \text{true}, i.e., \( \varrho[\overline{\tau} \mapsto \overline{w}, \overline{z} \mapsto \varphi'(\overline{z})] \models \text{true} \). Intuitively, this holds because
guards are composed of (in)equalities which are stable under the bijection $\text{swap}_{\text{adr}}$. Formally, $g$ is equivalent to:

$$g \models \bigwedge_i \bigvee_j \text{var}_{i,1} \triangleq \text{var}_{i,2} \quad \text{with} \quad \text{var}_{i,k} \in \{r, z\} \quad \text{and} \quad \triangleq \in \{=, \neq\}.$$ 

Hence, we have to show

$$(\text{var} \triangleq \text{var}')[r \mapsto \overline{r}, z \mapsto \varphi(z)] \models \text{true} \iff (\text{var} \triangleq \text{var}')[\overline{r} \mapsto \overline{z}, z \mapsto \varphi'(z)] \models \text{true}$$

for every $i, j$ and $\text{var} = \text{var}_{i,1}$ and $\text{var}' = \text{var}_{i,2}$. We conclude this as follows:

$$(\text{var} \triangleq \text{var}')[r \mapsto \overline{r}, z \mapsto \varphi(z)] \models \text{true}$$

$$\iff \text{var}[r \mapsto \overline{r}, z \mapsto \varphi(z)] \triangleq \text{var}'[\overline{r} \mapsto \overline{z}, z \mapsto \varphi(z)]$$

$$\iff \text{swap}_{\text{adr}}(\text{var}[r \mapsto \overline{r}, z \mapsto \varphi(z)]) \triangleq \text{swap}_{\text{adr}}(\text{var}'[\overline{r} \mapsto \overline{z}, z \mapsto \varphi(z)])$$

$$\iff \text{var}[r \mapsto \text{swap}_{\text{adr}}(\overline{r}), z \mapsto \text{swap}_{\text{adr}}(\varphi(z))] \triangleq \text{var}'[\overline{r} \mapsto \text{swap}_{\text{adr}}(\overline{z}), z \mapsto \text{swap}_{\text{adr}}(\varphi(z))]$$

$$\iff \text{var}[\overline{r} \mapsto \overline{z}, z \mapsto \varphi'(z)] \triangleq \text{var}'[\overline{r} \mapsto \overline{z}, z \mapsto \varphi'(z)]$$

$$\iff (\text{var} \triangleq \text{var}')[r \mapsto \overline{r}, z \mapsto \varphi'(z)] \models \text{true}$$

where the second equivalence holds because $\text{swap}_{\text{adr}}$ is a bijections, and the third equivalence holds because $\text{var}$ and $\text{var}'$ are either contained in $r$ or $z$ by the definition of SMR automata. We conclude (8).

Altogether, the overall implication

$$(l, \varphi) \xrightarrow{h} (l', \varphi) \implies (l, \text{swap}_{\text{adr}} \circ \varphi) \xrightarrow{\text{swap}_{\text{hist}}(h)} (l', \text{swap}_{\text{adr}} \circ \varphi)$$

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 $(l, \text{swap}_{\text{adr}} \circ \varphi) \xrightarrow{\text{swap}_{\text{hist}}(h)} (l', \text{swap}_{\text{adr}} \circ \varphi)$ using the address mapping $\text{swap}_{\text{adr}}^{-1}$. This yields:

$$(l, \text{swap}_{\text{adr}} \circ \varphi) \xrightarrow{\text{swap}_{\text{hist}}(h)} (l', \text{swap}_{\text{adr}} \circ \varphi)$$

$$\implies (l, \text{swap}_{\text{adr}}^{-1} \circ \text{swap}_{\text{adr}} \circ \varphi) \xrightarrow{\text{swap}_{\text{hist}}^{-1}(\text{swap}_{\text{hist}}(h))} (l', \text{swap}_{\text{adr}}^{-1} \circ \text{swap}_{\text{adr}} \circ \varphi).$$

Then, Lemma B.62 together with $\text{swap}_{\text{adr}}^{-1} \circ \text{swap}_{\text{adr}} = \text{id}$ gives:

$$(l, \text{swap}_{\text{adr}}^{-1} \circ \text{swap}_{\text{adr}} \circ \varphi) \xrightarrow{\text{swap}_{\text{hist}}(h)} (l', \text{swap}_{\text{adr}}^{-1} \circ \text{swap}_{\text{adr}} \circ \varphi)$$

$$\implies (l, \text{swap}_{\text{adr}} \circ \varphi) \xrightarrow{\text{swap}_{\text{hist}}(h)} (l', \text{swap}_{\text{adr}} \circ \varphi).$$

This concludes the claim.
Proof C.44 (Lemma B.64). Let $a \in \text{Addr}$. We conclude as follows:

\[
\begin{align*}
    h' &\in F_C(h, a) \\
\iff h.h' &\in S(O) \land \text{frees}(h') \subseteq \{ a \} \\
\iff \text{swap}_{\text{hist}}(h).\text{swap}_{\text{hist}}(h') &\in S(O) \land \text{frees}(\text{swap}_{\text{hist}}(h')) \subseteq \{ \text{swap}_{\text{adr}}(a) \} \\
\iff \text{swap}_{\text{hist}}(h') &\in F_C(\text{swap}_{\text{hist}}(h), \text{swap}_{\text{adr}}(a))
\end{align*}
\]

where the second equivalence holds because of Lemma B.63.

\[
\begin{proof}
\end{proof}
\]

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 $\tau.\text{act} \in O[[P]]^{\text{swap}_{\text{adr}}(A)}_\text{Addr}$ and assume that we have already constructed $\sigma$ with:

- (P1) $\sigma \in O[[P]]^{\text{swap}_{\text{adr}}(A)}_\text{Addr}$
- (P2) $\forall \text{pexp} \in \text{PExp}. m_\sigma(\text{swap}_{\text{exp}}(\text{pexp})) = \text{swap}_{\text{adr}}(m_\tau(\text{pexp}))$
- (P3) $\forall \text{dexp} \in \text{DExp}. m_\sigma(\text{swap}_{\text{exp}}(\text{dexp})) = m_\tau(\text{dexp})$
- (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) $\sigma.\text{act}' \in O[[P]]^{\text{swap}_{\text{adr}}(A)}_\text{Addr}$
- (G2) $\forall \text{pexp} \in \text{PExp}. m_{\sigma.\text{act}}(\text{swap}_{\text{exp}}(\text{pexp})) = \text{swap}_{\text{adr}}(m_{\tau.\text{act}}(\text{pexp}))$
- (G3) $\forall \text{dexp} \in \text{DExp}. m_{\sigma.\text{act}}(\text{swap}_{\text{exp}}(\text{dexp})) = m_{\tau.\text{act}}(\text{dexp})$
- (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 $\text{com} \neq \text{frees}(a)$ and $\text{com} \neq \text{env}(a)$. Otherwise, we replace the address that is used: $\text{com}' = \text{frees}(\text{swap}_{\text{adr}}(a))$ 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 $\text{act}'$ is enabled after $\sigma$. This, together with (P1) yields the desired property. We do a case distinction on the executed command $\text{com}$.

- **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 $\text{swap}_{\exp}(q) = q$.

♦ *Ad (G2).* For $p$ the claim follows by choice of $\text{up}'$. So consider $p_{\exp} \in P\exp \setminus \{ p \}$. Then, we conclude as follows:

$$m_{\sigma, \text{act}}(\text{swap}_{\exp}(p_{\exp})) = m_\sigma(\text{swap}_{\exp}(p_{\exp})) = \text{swap}_{\text{adr}}(m_\tau(p_{\exp}))$$

where the first equality holds because only $p$ is updated by $\text{up}'$ and $\text{swap}_{\exp}(p_{\exp}) \neq p$, the second equality holds by (P2), and the third equality holds because only $p$ is updated by $\text{up}$.

♦ *Ad (G3).* Neither $\text{up}$ nor $\text{up}'$ update data expressions. We get $m_{\tau, \text{act}}(d_{\exp}) = m_\tau(d_{\exp})$ and $m_{\sigma, \tau, \text{act}}(d_{\exp}) = m_\sigma(d_{\exp})$ for all $d_{\exp} \in D\exp$. Hence, the claim follows from (P3).

♦ *Ad (G4).* By $\text{swap}_{\exp}(q) = q$ 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 $\text{swap}_{\exp}$ as follows:

$$\text{valid}_{\sigma, \text{act}} = \text{valid}_\sigma \otimes \{ p \} = \text{swap}_{\exp}(\text{valid}_\tau) \otimes \text{swap}_{\exp}(\{ p \})$$

$$= \text{swap}_{\exp}(\text{valid}_\tau \otimes \{ p \}) = \text{swap}_{\exp}(\text{valid}_{\tau, \text{act}})$$

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 $\text{up} = [p \mapsto b]$. We choose $\text{up}' = [p \mapsto \text{swap}_{\text{adr}}(b)]$. 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:

$$a' = m_\sigma(q) = m_\sigma(\text{swap}_{\exp}(q)) = \text{swap}_{\text{adr}}(m_\tau(q)) = \text{swap}_{\text{adr}}(a) .$$

Observe that this means $a' \neq \text{seg}$. Then, we conclude as follows:

$$b' = m_\sigma(a'_{\text{next}}) = m_\sigma(\text{swap}_{\text{adr}}(a)_{\text{next}}) = m_\sigma(\text{swap}_{\exp}(a_{\text{next}}))$$

$$= \text{swap}_{\text{adr}}(m_\tau(a_{\text{next}})) = \text{swap}_{\text{adr}}(b) .$$
Proof of Meta Theory

\(\diamond\) **Ad (G2).** For \(p\) the claim follows by choice of \(\text{up}'\). For \(\text{pexp} \in P\text{Exp} \setminus \{ p\}\) the claim follows from (P2) together with \(\text{up}\) and \(\text{up}'\) not modifying the valuation of \(\text{pexp}\), as in the previous case.

\(\diamond\) **Ad (G3).** Neither \(\text{up}\) nor \(\text{up}'\) update data expressions. Hence, the claim follows by (P3).

\(\diamond\) **Ad (G4).** By definition, \(\text{swap}_{\text{exp}}(q) = q\). So (P4) yields \(q \in \text{valid}_r \iff q \in \text{valid}_d\). Since (G1) from above gives \(a' = \text{swap}_{\text{addr}}(a)\), we have \(\text{swap}_{\text{exp}}(a.\text{next}) = a'.\text{next}\). As a consequence, (P4) yields \(a.\text{next} \in \text{valid}_r \iff a'.\text{next} \in \text{valid}_d\). This means we have \(p \in \text{valid}_r.\text{act} \iff p \in \text{valid}_d.\text{act}'\) because \(m_r(q) \in \text{Adr}\). Together with (P4) this concludes the claim because \(\text{act}/\text{act}'\) affects only the validity of \(p\).

\(\diamond\) **Case 3:** \(\text{com} \equiv p.\text{next} = q\)

Analogous to the previous case.

\(\diamond\) **Case 4:** \(\text{com} \equiv u = \text{op}(u_1, \ldots, u_n)\)

The update is \(\text{up} = [u \mapsto d]\) with \(d = \text{op}(m_r(u_1), \ldots, m_r(u_i))\). Choose \(\text{up}' = \text{up}\). Since the pointer expression valuations, the validity, and the fresh/freed address are not altered as well as no event is emitted by \(\text{act}/\text{act}'\), (G2) and (G4) to (G7) follow immediately from (P2) and (P4) to (P7).

\(\diamond\) **Ad (G1).** By (P3) we have \(m_r(u_i) = m_r(u_i)\). Hence, \(\text{act}'\) is enabled after \(\sigma\).

\(\diamond\) **Ad (G3).** We have \(m_{r.\text{act}}(\text{swap}_{\text{exp}}(u)) = m_{r.\text{act}}(u) = d = m_{r.\text{act}}(u)\). And because no other data expressions are updated by \(\text{up}/\text{up}'\), the claim follows from (P3).

\(\diamond\) **Case 5:** \(\text{com} \equiv u = q.\text{data}\)

Let \(a = m_r(q)\). By definition, \(a \neq \text{seg}\). Let \(d = m_r(a.\text{data})\). So \(\text{up} = [u \mapsto d]\). Choose \(\text{up}' = \text{up}\). Since the pointer expression valuations, the validity, and the fresh/freed address are not altered as well as no event is emitted by \(\text{act}/\text{act}'\), (G2) and (G4) to (G7) follow from (P2) and (P4) to (P7).

\(\diamond\) **Ad (G1).** Let \(a' = m_\sigma(q)\) and \(d' = m_\sigma(a'.\text{data})\). For enabledness of \(\text{act}'\), we have to show \(d' = d\). To see this, first note that by (P2) we have:

\[
a' = m_\sigma(q) = m_\sigma(\text{swap}_{\text{exp}}(q)) = \text{swap}_{\text{addr}}(m_r(q)) = \text{swap}_{\text{addr}}(a) .
\]

This means \(a' \neq \text{seg}\). Together with (P3) we conclude as follows:

\[
d' = m_\sigma(a'.\text{data}) = m_\sigma(\text{swap}_{\text{exp}}(a.\text{data})) = m_r(a.\text{data}) = d .
\]

\(\diamond\) **Ad (G3).** We have \(m_{r.\text{act}}'(u) = d = m_{r.\text{act}}(u) = m_{r.\text{act}}(\text{swap}_{\text{exp}}(u))\). And because no other data expressions are updated by \(\text{up}/\text{up}'\), the claim follows from (P3).
Case 6: \( \text{com} \equiv p.data = u \)
 Analogous to the previous case.

Case 7: \( \text{com} \equiv p := \text{malloc} \)
 Let \( a = m_{r.\text{act}}(p) \). The update is \( up = [p \mapsto a, a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d] \) for some \( d \). By definition, we have \( a \in \text{fresh}_r \cup (\text{freed}_r \cap A) \). Choose \( up' \) as follows

\[
up' = [p \mapsto \text{swap}_{adr}(a), \text{swap}_{adr}(a).\text{next} \mapsto \text{seg}, \text{swap}_{adr}(a).\text{data} \mapsto d].
\]

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:

\[
\text{fresh}_\sigma \cup (\text{freed}_\sigma \cap \text{swap}_{adr}(A)) = \text{swap}_{adr}(\text{fresh}_r) \cup (\text{swap}_{adr}(\text{freed}_r) \cap \text{swap}_{adr}(A)) = \text{swap}_{adr}(\text{fresh}_r \cup (\text{freed}_r \cap A)).
\]

Then, \( a \in \text{fresh}_r \cup (\text{freed}_r \cap A) \) yields \( \text{swap}_{adr}(a) \in \text{fresh}_\sigma \cup (\text{freed}_\sigma \cap \text{swap}_{adr}(A)) \).

Ad (G2). We have

\[
m_{\sigma.\text{act}}(\text{swap}_{exp}(p)) = m_{\sigma.\text{act}}(p) = \text{swap}_{adr}(a) = \text{swap}_{adr}(m_{r.\text{act}}(p)) \quad \text{and} \quad m_{\sigma.\text{act}}(\text{swap}_{exp}(a.\text{next})) = m_{\sigma.\text{act}}(\text{swap}_{adr}(a).\text{next}) = \text{seg} = \text{swap}_{adr}(\text{seg}) = \text{swap}_{adr}(m_{r.\text{act}}(a.\text{next})).
\]

Since no other pointer expressions are updated, the claim follows from (P2).

Ad (G3). We have:

\[
m_{\sigma.\text{act}}(\text{swap}_{exp}(a.\text{data})) = m_{\sigma.\text{act}}(\text{swap}_{adr}(a).\text{data}) = d = m_{r.\text{act}}(a.\text{data}).
\]

Since no other data expression is updated, the follows from (P3).

Ad (G4). We conclude using (P4) and Lemma B.61 as follows:

\[
\text{valid}_{\sigma.\text{act}} = \text{valid}_{\sigma} \cup \{p, \text{swap}_{adr}(a).\text{next}\} = \text{swap}_{exp}(\text{valid}_r) \cup \text{swap}_{exp}(\{p, a.\text{next}\}) = \text{swap}_{exp}(\text{valid}_r \cup \{p, a.\text{next}\}) = \text{swap}_{exp}(\text{valid}_{r.\text{act}}).
\]

Ad (G5). We conclude using (P5) and Lemma B.61 as follows:

\[
\text{freed}_{\sigma.\text{act}} = \text{freed}_\sigma \setminus \text{swap}_{adr}(a) = \text{swap}_{adr}(\text{freed}_r) \setminus \text{swap}_{adr}(a) = \text{swap}_{adr}(\text{freed}_r \setminus a) = \text{swap}_{adr}(\text{freed}_{r.\text{act}}).
\]
\* Ad (G6). We conclude using (P6) and Lemma B.61 as follows:

\[
\text{fresh}_{\sigma, \text{act}} = \text{fresh}_{\sigma} \setminus \text{swap}_{\text{adr}}(a) = \text{swap}_{\text{adr}}(\text{fresh}_{r}) \setminus \text{swap}_{\text{adr}}(a) = \text{swap}_{\text{adr}}(\text{fresh}_{\bar{r}}) \setminus \text{swap}_{\text{adr}}(\text{fresh}_{\bar{r}, \text{act}}).
\]

\* Case 8: \( \text{com} \equiv \text{free}(a) \)

The update is \( \text{up} = \emptyset \). Choose \( \text{com}' = \text{free}(\text{swap}_{\text{adr}}(a)) \) and \( \text{up}' = \emptyset \). Then, (G2) and (G3) follow from (P2) and (P3) because \( m_r = m_{r, \text{act}} \) and \( m_\sigma = m_{\sigma, \text{act}} \).

\* Ad (G1). By the semantics, \( \text{free}(a) \in F_C(r, a) \). By (P7) together with Lemma B.64 we get \( \text{free}(\text{swap}_{\text{adr}}(a)) \in F_C(\sigma, \text{swap}_{\text{adr}}(a)) \). Hence, \( \text{act}' \) is enabled after \( \sigma \).

\* Ad (G4). Consider \( \text{pexp}' \in PExp \). Since \( \text{swap}_{\text{adr}} \) is a bijection, there is some \( \text{pexp} \in PExp \) such that \( \text{swap}_{\text{exp}}(\text{pexp}) = \text{pexp}' \). First, we get:

\[
\text{pexp}' \in \text{valid}_r \iff \text{swap}_{\text{exp}}(\text{pexp}) \in \text{valid}_r
\]

\[
\iff \text{swap}_{\text{exp}}(\text{pexp}) \in \text{swap}_{\text{exp}}(\text{valid}_r) \iff \text{pexp} \in \text{valid}_r
\]

where the second equivalence is due to (P4) and the third equivalence holds since \( \text{swap}_{\text{adr}} \) is a bijection. Second, we have:

\[
m_\sigma(\text{pexp}') \neq \text{swap}_{\text{adr}}(a) \iff m_\sigma(\text{swap}_{\text{exp}}(\text{pexp})) \neq \text{swap}_{\text{adr}}(a)
\]

\[
\iff \text{swap}_{\text{adr}}(m_\sigma(\text{pexp})) \neq \text{swap}_{\text{adr}}(a) \iff m_r(\text{pexp}) \neq a
\]

where the second equivalence is due to (P2) and the third equivalence holds since \( \text{swap}_{\text{adr}} \) is a bijection. Last, we have:

\[
\text{pexp}' \cap \text{Adr} \neq \{ \text{swap}_{\text{adr}}(a) \} \iff \text{swap}_{\text{exp}}(\text{pexp}) \cap \text{Adr} \neq \{ \text{swap}_{\text{adr}}(a) \}
\]

\[
\iff \text{pexp} \cap \text{Adr} \neq \{ a \}
\]

Altogether, this gives:

\[
\text{pexp}' \in \text{valid}_{\sigma, \text{act}}' \iff \text{pexp}' \in \text{valid}_r \land m_\sigma(\text{pexp}') \neq \text{swap}_{\text{adr}}(a) \land \text{pexp}' \cap \text{Adr} \neq \{ \text{swap}_{\text{adr}}(a) \}
\]

\[
\iff \text{pexp} \in \text{valid}_r \land m_r(\text{pexp}) \neq a \land \text{pexp} \cap \text{Adr} \neq \{ a \}
\]

\[
\iff \text{pexp} \in \text{valid}_{r, \text{act}}
\]

\[
\iff \text{swap}_{\text{exp}}(\text{pexp}) \in \text{swap}_{\text{exp}}(\text{valid}_{r, \text{act}})
\]

\[
\iff \text{pexp}' \in \text{swap}_{\text{exp}}(\text{valid}_{r, \text{act}})
\]

where the last but first equivalence holds because \( \text{swap}_{\text{adr}} \) is a bijection.
\( \diamond \) Ad (G5). We conclude using (P5) and Lemma B.61:

\[
\text{freed}_{\sigma, \text{act}} = \text{freed}_\sigma \cup \{ \text{swap}_{\text{adr}}(a) \} = \text{swap}_{\text{adr}}(\text{freed}_\tau) \cup \{ \text{swap}_{\text{adr}}(a) \} \\
= \text{swap}_{\text{adr}}(\text{freed}_\tau \cup \{ a \}) = \text{swap}_{\text{adr}}(\text{freed}_{\tau, \text{act}}).
\]

\( \diamond \) Ad (G6). We conclude using (P6) and Lemma B.61:

\[
\text{fresh}_{\sigma, \text{act}} = \text{fresh}_\sigma \setminus \{ \text{swap}_{\text{adr}}(a) \} = \text{swap}_{\text{adr}}(\text{fresh}_\tau) \setminus \{ \text{swap}_{\text{adr}}(a) \} \\
= \text{swap}_{\text{adr}}(\text{fresh}_\tau \setminus \{ a \}) = \text{swap}_{\text{adr}}(\text{fresh}_{\tau, \text{act}}).
\]

\( \diamond \) Ad (G7). We conclude using (P7)

\[
H(\sigma, \text{act}^I) = H(\sigma).\text{free}(\text{swap}_{\text{adr}}(a)) = \text{swap}_{\text{hist}}(H(\tau)).\text{swap}_{\text{hist}}(\text{free}(a)) \\
= \text{swap}_{\text{hist}}(H(\tau).\text{free}(a)) = \text{swap}_{\text{hist}}(H(\tau.\text{act})).
\]

\( \diamond \) Case 9: \( \text{com} \equiv \text{assume} \) 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 \( \text{act}/\text{act}' \), (G2), (G3) and (G5) to (G7) follow immediately from (P2), (P3) and (P5) to (P7).

\( \diamond \) Ad (G1). First, consider the case where \( \text{cond} \) 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, \( \text{act}' \) is enabled after \( \sigma \).

Now, consider the case where \( \text{cond} \) 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, \( \text{act}' \) is enabled after \( \sigma \).

\( \diamond \) Ad (G4). Note that we have:

\[
\text{valid}_{\sigma, \text{act}^I} \neq \text{valid}_\sigma \\
\iff \text{cond} \equiv p = q \land \{ p, q \} \cap \text{valid}_\sigma \neq \emptyset \land \{ p, q \} \not\in \text{valid}_\tau \\
\iff \text{cond} \equiv p = q \land \{ p, q \} \cap \text{swap}_{\exp}(\text{valid}_\tau) \neq \emptyset \land \{ p, q \} \not\in \text{swap}_{\exp}(\text{valid}_\tau) \\
\iff \text{cond} \equiv p = q \land \{ p, q \} \cap \text{valid}_\tau \neq \emptyset \land \{ p, q \} \not\in \text{valid}_\tau \\
\iff \text{valid}_{\tau, \text{act}} \neq \text{valid}_\tau
\]

where the second equivalence is by (P4). Consider the case \( \text{valid}_{\sigma, \text{act}^I} \neq \text{valid}_\sigma \). So we conclude using (P4) and Lemma B.61:

\[
\text{valid}_{\sigma, \text{act}^I} = \text{valid}_\sigma \cup \{ p, q \} = \text{swap}_{\exp}(\text{valid}_\tau) \cup \text{swap}_{\exp}(\{ p, q \}) \\
= \text{swap}_{\exp}(\text{valid}_\tau \cup \{ p, q \}) = \text{swap}_{\exp}(\text{valid}_{\tau, \text{act}}).
\]
In all other cases, we have $valid_{\tau, act} = valid_{\tau}$ and $valid_{\tau, act'} = valid_{\tau}$. Then, the claim follows immediately from (P4).

Case 10: $com \equiv \text{in}_i : \text{func}(r_1, \ldots, r_n)$

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 $i \in \{1, \ldots, n\}$. By definition, we have $m_{\tau}(p_i) \in \text{Adr} \cup \text{Dom}$. Hence, we get $\text{swap}_{\text{adr}}(m_{\tau}(p_i)) \in \text{Adr} \cup \text{Dom}$. Then (P2) yields $m_{\tau}(p_i) \in \text{Adr} \cup \text{Dom}$. 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 $\text{in}_i : \text{func}(t, v_1, \ldots, v_n)$ with $v_i = m_{\tau}(r_i)$. By (P2) and (P3), $m_{\sigma}(r_i) = \text{swap}_{\text{adr}}(v_i)$. So, act emits the event $\text{in}_i : \text{func}(t, \text{swap}_{\text{adr}}(v_1), \ldots, \text{swap}_{\text{adr}}(v_n))$. We conclude by (P7):

$$H(\sigma, act') = H(\sigma).\text{in}_i : \text{func}(t, \text{swap}_{\text{adr}}(v_1), \ldots, \text{swap}_{\text{adr}}(v_n))$$
$$= \text{swap}_{\text{hist}}(H(\tau)).\text{swap}_{\text{hist}}(\text{in}_i : \text{func}(t, v_1, \ldots, v_n))$$
$$= \text{swap}_{\text{hist}}(H(\tau).\text{in}_i : \text{func}(t, v_1, \ldots, v_n)) = \text{swap}_{\text{hist}}(H(\tau, act)).$$

Case 11: $com \equiv \text{re}$

Follows analogously to the previous case.

Case 12: $com \equiv \text{env}(a)$

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 $com' = \text{env}(\text{swap}_{\text{adr}}(a))$ and $[\text{swap}_{\text{adr}}(a).\text{next} \mapsto \text{seg}, \text{swap}_{\text{adr}}(a).\text{data} \mapsto d]$. Then, (G4) to (G7) follow immediately from (P4) to (P7).

Ad (G1). We have to show that $\text{swap}_{\text{adr}}(a) \in \text{fresh}_{\tau} \cup \text{freed}_{\tau}$. This follows immediately from (P5) and (P6) together with $a \in \text{fresh}_{\tau} \cup \text{freed}_{\tau}$.

Ad (G2). We have:

$$m_{\tau, act'}(\text{swap}_{\text{exp}}(a.\text{next})) = m_{\tau, act'}(\text{swap}_{\text{adr}}(a).\text{next}) = \bot$$
$$= m_{\tau, act}(a.\text{next}) = \text{swap}_{\text{adr}}(m_{\tau, act}(a.\text{next})).$$

For all other $pexp \in PExp \setminus a.\text{next}$ we conclude with (P2) as follows:

$$m_{\tau, act}(\text{swap}_{\text{exp}}(pexp)) = m_{\tau}(\text{swap}_{\text{exp}}(pexp)) = \text{swap}_{\text{adr}}(m_{\tau}(pexp))$$
$$= \text{swap}_{\text{adr}}(m_{\tau, act}(pexp)).$$

Appendix C Proof of Meta Theory
Ad (G3). We have:

\[ m_{\sigma,act}(\text{swap}_\text{exp}(a.\text{data})) = m_{\sigma,act}(\text{swap}_\text{adr}(a).\text{data}) = d = m_{\tau,act}(a.\text{data}). \]

For all other \( \text{dexp} \in D\text{Exp} \setminus \{ a.\text{data} \} \) we conclude with (P3) as follows:

\[ m_{\sigma,act}(\text{swap}_\text{exp}(\text{dexp})) = m_{\sigma}(\text{swap}_\text{exp}(\text{dexp})) = m_{\tau}(\text{dexp}) = m_{\tau,act}(\text{dexp}). \]

Case 13: \( \text{com} \in \{ \text{skip}, \text{beginAtomic}, \text{endAtomic}, \text{@inv} \} \)

Since \( \text{act} \) 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 \( O \) support elision. Let \( b \in \text{fresh}_r \setminus A \). Let \( \text{swap}_\text{adr} : \text{Adr} \rightarrow \text{Adr} \) be the address mapping defined by \( \text{swap}_\text{adr}(a) = b \), \( \text{swap}_\text{adr}(b) = a \), and \( \text{swap}_\text{adr}(c) = c \) such that \( a \neq c \neq b \). Theorem B.65 yields \( \sigma \in O[\text{P}^{\text{swap}_\text{adr}(A)}_{\text{Adr}}] \) with

- \( m_{\sigma} \circ \text{swap}_\text{exp} = \text{swap}_\text{adr} \circ m_{\tau} \),
- \( H(\sigma) = \text{swap}_\text{hist}(H(\tau)) \),
- \( \text{valid}_\sigma = \text{swap}_\text{exp}(\text{valid}_\tau) \),
- \( \text{fresh}_\sigma = \text{swap}_\text{adr}(\text{fresh}_\tau) \),
- \( \text{freed}_\sigma = \text{swap}_\text{adr}(\text{freed}_\tau) \), and
- \( \text{ctrl}(\sigma) = \text{ctrl}(\tau) \).

We show that \( \sigma \) satisfies the claim. First, note that \( \text{swap}_\text{adr}(A) = A \) by \( a, b \notin A \). Consequently, we have \( \sigma \in O[\text{P}^A_{\text{Adr}}] \). We first show the following auxiliaries:

\[ a.\text{next}, b.\text{next} \notin \text{valid}_\tau \]  
\[ f = f^{-1} \text{ for } f \in \{ \text{swap}_\text{adr}, \text{swap}_\text{exp}, \text{swap}_\text{hist} \} \]  
\[ \forall \text{pexp} \in \text{valid}_\tau. \text{pexp} \equiv \text{swap}_\text{exp}(\text{pexp}) \]  
\[ \forall \text{pexp} \in \text{valid}_\tau. m_{\tau}(\text{pexp}) = \text{swap}_\text{adr}(m_{\tau}(\text{pexp})) \]  
\[ \forall c \in m_{\tau}(\text{valid}_\tau). c.\text{data} \equiv \text{swap}_\text{exp}(c.\text{data}) \]  
\[ \forall c \in \text{Adr} \setminus \{ a, b \}. F_O(\tau, c) \subseteq F_O(\sigma, c) \]  
\[ \forall c \in A. c \in \text{retired}_\tau \iff c \in \text{retired}_\sigma \]

Ad (9). Holds by \( a \notin \text{adr}(m_{\tau}|_{\text{valid}_\tau}) \), and by \( b \in \text{fresh}_r \) together with Lemma B.48.

Ad (10). Holds by choice of \( \text{swap}_\text{adr} \).

Ad (11). Holds by (9) and the definition of \( \text{swap}_\text{adr} \) and its induced \( \text{swap}_\text{exp} \).

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 \( b \in \text{fresh}_r \) giving \( b \notin \text{range}(m_{\tau}) \) by Lemma B.42 and thus \( b \notin m_{\tau}(\text{valid}_\tau) \).
○ Ad (13). From $c \in m_\sigma(\text{valid}_z)$ we get some $\text{pexp} \in \text{valid}_z$ with $m_\sigma(\text{pexp}) = c$. Thus, (12) yields the following: $c = m_\sigma(\text{pexp}) = \text{swap}_\text{adv} (m_\sigma(\text{pexp})) = \text{swap}_\text{adv} (c)$.

○ Ad (14). Let $c \in \text{Adr} \setminus \{a, c\}$. Then,

$$F_O(\tau, c) = F_O(\mathcal{H}(\tau), c) = F_O(\text{swap}_\text{hist}(\mathcal{H}(\tau)), c) = F_O(\mathcal{H}(\sigma), c) = F_O(\sigma, c)$$

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 $\text{swap}_\text{adv} (c) = c$. To the contrary, assume that we have $c \in \text{retired}_\tau$ but $c \notin \text{retired}_\sigma$. The former means that $\tau$ is of the form $\tau = \tau_1.\text{act}_{\tau_2}$ with $\text{act} = \langle t, \text{com}, \text{up} \rangle$ and $\text{com} \equiv \text{in:retired}_p$ and $m_{\tau_2}(p) = c$. Let $\mathcal{H}(\tau) = h_1.\text{evt}h_2$ 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 $\sigma$ in Proof C.45 of Theorem B.65, we know that $\sigma$ is of the form $\sigma = \sigma_1.\text{act}_{\sigma_2}$. Moreover, we have $\mathcal{H}(\sigma) = \text{swap}_\text{hist}(\mathcal{H}(\tau))$. Hence, we get $\text{swap}_\text{hist}(\mathcal{H}(\sigma)) = \mathcal{H}(\tau)$. This, in turn, means that $c \notin \text{swap}_\text{hist}^{-1}(\text{frees}_{h_2})$. By $\text{swap}_\text{adv} (c) = c$, we obtain that $c \notin \text{frees}_{h_2}$ holds. Further, $\text{swap}_\text{hist}^{-1}(\text{in:retire}(t, c)) = \text{in:retire}(t, c)$ by $\text{swap}_\text{adv} (c) = c$. Hence, we arrive at $c \in \text{retired}_\sigma$. The reverse direction follows analogously.

○ Ad $\tau \sim \sigma$. We already have $\text{ctrl}(\sigma) = \text{ctrl}(\tau)$. We get:

$$\text{dom}(m_\sigma|_{\text{valid}_z})$$

$$= \text{valid}_z \cup \text{DVar} \cup \{c.\text{data} \mid c \in m_\sigma(\text{valid}_z)\}$$

$$= \text{swap}_\text{exp}(\text{valid}_z) \cup \text{DVar} \cup \{c.\text{data} \mid c \in \text{swap}_\text{adv}(m_\tau(\text{swap}_\text{exp}^{-1}(\text{valid}_z)))\}$$

$$= \text{swap}_\text{exp}(\text{valid}_z) \cup \text{DVar} \cup \{c.\text{data} \mid c \in \text{swap}_\text{adv}(m_\tau(\text{valid}_z))\}$$

$$= \text{valid}_z \cup \text{DVar} \cup \{c.\text{data} \mid c \in \text{swap}_\text{adv}(m_\tau(\text{valid}_z))\}$$

$$= \text{valid}_z \cup \text{DVar} \cup \{c.\text{data} \mid c \in m_\tau(\text{valid}_z)\} = \text{dom}(m_\tau|_{\text{valid}_z})$$

where the first equality is by definition, the second by the properties of $\sigma$, the third by (10), the fourth by (11), the fifth by (12), and the last by definition. Then, we get for all pointer expression $\text{pexp} \in \text{dom}(m_\sigma|_{\text{valid}_z}) \cap \text{PExp}$:

$$m_\sigma(\text{pexp}) = \text{swap}_\text{adv}(m_\tau(\text{swap}_\text{exp}^{-1}(\text{pexp}))) = \text{swap}_\text{adv}(m_\tau(\text{pexp})) = m_\tau(\text{pexp})$$

using the properties of $\sigma$, the fact that $\text{pexp} \in \text{valid}_z$ must hold, and (10) to (12). Moreover, we get for $\text{dexp} \in \text{dom}(m_\sigma|_{\text{valid}_z}) \cap \text{DExp}$:

$$m_\sigma(\text{dexp}) = m_\tau(\text{swap}_\text{exp}^{-1}(\text{dexp})) = m_\tau(\text{dexp})$$

by (13) because $c.\text{data} \in \text{dom}(m_\sigma|_{\text{valid}_z}) = \text{dom}(m_\tau|_{\text{valid}_z})$ implies $c \in m_\tau(\text{valid}_z)$. Altogether, this yields the desired $m_\tau|_{\text{valid}_z} = m_\sigma|_{\text{valid}_z}$.
\( Ad \, \tau \preceq A \, \sigma \). Let \( c \in A \). We show \( \tau \preceq c \, \sigma \). We have \( a \neq c \neq b \) and thus \( \text{swap}_{\text{adr}}(c) = c \). So using (10) we get:

\[
c \in \text{fresh}_{\sigma} \cup \text{freed}_{\sigma} \iff c \in \text{swap}_{\text{adr}}(\text{fresh}_{\tau}) \cup \text{swap}_{\text{adr}}(\text{freed}_{\tau}) \]
\[
\iff c \in \text{swap}_{\text{adr}}(\text{fresh}_{\tau} \cup \text{freed}_{\tau}) \]
\[
\iff \text{swap}_{\text{adr}}(c) \in \text{swap}_{\text{adr}}(\text{fresh}_{\tau} \cup \text{freed}_{\tau}) \]
\[
\iff c \in \text{fresh}_{\tau} \cup \text{freed}_{\tau}.
\]

Moreover, we have by (15):

\[
c \in \text{retired}_{\tau} \iff c \in \text{retired}_{\sigma}.
\]

From (14) we get \( F_{\sigma}(\tau, c) \subseteq F_{\sigma}(\sigma, c) \). Then, we have for \( p \in \text{PVar} \):

\[
c = m_{\sigma}(p) \iff c = \text{swap}_{\text{adr}}(m_{\tau}(\text{swap}_{\text{exp}}(p))) \]
\[
\iff \text{swap}_{\text{adr}}(c) = \text{swap}_{\text{adr}}(m_{\tau}(\text{swap}_{\text{exp}}(p))) \]
\[
\iff c = m_{\tau}(p).
\]

Now, consider \( c' \in m_{\tau}(\text{valid}_{\tau}) \). We have:

\[
c = m_{\sigma}(c'.\text{next}) \iff c = \text{swap}_{\text{adr}}(m_{\tau}(\text{swap}_{\text{exp}}(c'.\text{next}))) \]
\[
\iff \text{swap}_{\text{adr}}(c) = \text{swap}_{\text{adr}}(m_{\tau}(\text{swap}_{\text{exp}}(c'.\text{next}))) \]
\[
\iff c = m_{\tau}(\text{swap}_{\text{exp}}(c'.\text{next})).
\]

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 form \( b \in \text{fresh}_{\tau} \) together with Lemma B.43.

\( Ad \, \tau \prec \sigma \). 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 \( c \notin \text{retired}_{\sigma} \). 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 \( c \in \text{retired}_{\tau} \).

\( Ad \, \text{Memory Implication} \). 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
This concludes the claim.

Proof C.47 (Lemma B.67). Let \( r.\text{act} \in \mathcal{O}[P]_\text{Adr}^{A} \) and \( \sigma \in \mathcal{O}[P]_\text{Adr}^{A} \) with \( r \sim \sigma \), \( r \leq_A \sigma \), and \( r \leq_A \sigma \).

Unrolling the premise gives:

\[
\begin{align*}
(P1) \quad & \text{ctrl}(r) = \text{ctrl}(\sigma) \\
(P2) \quad & m_{\sigma}(\text{valid}_{\sigma}) = m_{\sigma}(\text{valid}_{\sigma}) \\
(P3) \quad & \forall b \in \text{adr}(m_{\sigma}(\text{valid}_{\sigma})) \cup A. \mathcal{F}_\sigma(r, b) \subseteq \mathcal{F}_\sigma(\sigma, b) \\
(P4) \quad & \forall a \in A \forall p \in PVar. \ m_{r}(p) = a \iff m_{\sigma}(p) = a \\
(P5) \quad & \forall a \in A \forall b \in m_{r}(\text{valid}_{\sigma}) . \ m_{r}(b.\text{next}) = a \iff m_{\sigma}(b.\text{next}) = a \\
(P6) \quad & \forall a \in A . a \in \text{fresh}_{r} \cup \text{freed}_{r} \iff a \in \text{fresh}_{\sigma} \cup \text{freed}_{\sigma} \\
(P7) \quad & \forall a \in A . a \in \text{retired}_{r} \iff a \in \text{retired}_{\sigma}
\end{align*}
\]

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:

\[
\begin{align*}
(G0) \quad & \sigma.\text{act}' \in \mathcal{O}[P]_\text{Adr}^{A} \\
(G1) \quad & \text{ctrl}(r.\text{act}) = \text{ctrl}(\sigma.\text{act}') \\
(G2) \quad & m_{r.\text{act}}(\text{valid}_{r.\text{act}}) = m_{\sigma.\text{act}'}(\text{valid}_{\sigma.\text{act}'}) \\
(G3) \quad & \forall b \in \text{adr}(m_{r.\text{act}}(\text{valid}_{r.\text{act}})) \cup A. \mathcal{F}_\sigma(r.\text{act}, b) \subseteq \mathcal{F}_\sigma(\sigma.\text{act}', b) \\
(G4) \quad & \forall a \in A \forall p \in PVar. \ m_{r.\text{act}}(p) = a \iff m_{\sigma.\text{act}'}(p) = a \\
(G5) \quad & \forall a \in A \forall b \in m_{r.\text{act}}(\text{valid}_{r.\text{act}}) . \ m_{r.\text{act}}(b.\text{next}) = a \iff m_{\sigma.\text{act}'}(b.\text{next}) = a \\
(G6) \quad & \forall a \in A . a \in \text{fresh}_{r.\text{act}} \cup \text{freed}_{r.\text{act}} \iff a \in \text{fresh}_{\sigma.\text{act}'} \cup \text{freed}_{\sigma.\text{act}'} \\
(G7) \quad & \forall a \in A . a \in \text{retired}_{r.\text{act}} \iff a \in \text{retired}_{\sigma.\text{act}'}
\end{align*}
\]

Because \( \text{act}' \) executes the same command \( \text{com} \) as \( \text{act} \), we get (G1) from (P1) provided (G0) holds.

We do not comment on (G1) in the following.

\( \diamond \) **Case 1:** \( \text{com} \equiv x := y \)

Let \( b = m_{\sigma}(q) \) and \( b = m_{\sigma}(q) \). The update is \( \text{up} = [p \mapsto b] \). We choose \( \text{up}' = [p \mapsto b'] \).

Then, (G0) holds by the choice of \( \text{up}' \) together with (P1).
Ad (G2) for \( q \in \text{valid}_r \). By (P2) and Lemma B.37 we have \( q \in \text{valid}_\sigma \). Furthermore, (P2) yields \( b = b' \). We get

\[
m_r |_{\text{valid}_{r,act}} = m_r[p \mapsto b]|_{\text{valid}_r \cup \{p\}} = (m_r|_{\text{valid}_r})(p \mapsto b) .
\]

The second equality holds by \( m_r(q) = b \). That is, we preserve \( b \) data in \( m_r|_{\text{valid}_r} \) and only need to update the mapping of \( p \). Similarly, we get \( m_\sigma |_{\text{valid}_{\sigma,act}} = (m_\sigma|_{\text{valid}_\sigma})(p \mapsto b') \). Then, we conclude by \( m_r|_{\text{valid}_r} = m_\sigma |_{\text{valid}_\sigma} \) from (P2) together with \( b = b' \).

Ad (G2) for \( q \notin \text{valid}_r \). By (P2) together with Lemma B.37 we have \( q \notin \text{valid}_\sigma \). We get

\[
m_r |_{\text{valid}_{r,act}} = m_r[p \mapsto b]|_{\text{valid}_r \setminus \{p\}} = m_r|\text{valid}_r \setminus \{p\} .
\]

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 \( m_\sigma |_{\text{valid}_{\sigma,act}} = m_\sigma |\text{valid}_\sigma \setminus \{p\} \). We conclude by:

\[
m_r|\text{valid}_r\setminus\{p\} = (m_r|\text{valid}_r)|\text{valid}_r\setminus\{p\} = (m_\sigma|\text{valid}_\sigma)|\text{valid}_r\setminus\{p\}
\]

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 \( \text{act} \) and \( \text{act}' \) do not emit an event, it is sufficient to show \( \text{adr}(m_r |_{\text{valid}_{r,act}}) \subseteq \text{adr}(m_r |_{\text{valid}_r}) \). This follows from Lemma B.47.

Ad (G4). Let \( a \in A \). Only the valuation of \( p \) is changed by both \( \text{act} \) and \( \text{act}' \). So by (P4) it suffices to show that \( m_r(p) = a \iff m_\sigma |_{\text{valid}_\sigma}(p) = a \) holds. We conclude by:

\[
m_r(p) = a \iff m_r(q) = a \iff m_\sigma(q) = a \iff m_\sigma |_{\text{valid}_\sigma}(p) = a
\]

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 \( c \in m_r |_{\text{valid}_{r,act}} \). By Lemma B.47, \( c \in m_r |_{\text{valid}_r} \). We conclude:

\[
m_r |_{\text{valid}_{r,act}}(c.\text{next}) = a \iff m_r(c.\text{next}) = a \iff m_\sigma(c.\text{next}) = a
\]

where the first/last equivalence hold because \( \text{act}/\text{act}' \) does not update any pointer selector and the second equivalence holds by \( c \in m_r |_{\text{valid}_r} \) together with (P5).

Ad (G6) and (G7). By definition, we have \( \text{fresh}_r = \text{fresh}_{r,act} \) and \( \text{freed}_r = \text{freed}_{r,act} \) as well as \( \text{retired}_r = \text{retired}_{r,act} \). Similarly, for \( \sigma \). Then, we conclude by (P6) and (P7).
\textbf{Case 2: }\textit{com} \equiv x \coloneqq q, \text{sel}

By assumption, we have \( q \in \text{valid}_r \). Let \( b = m_r(q) \). By the semantics, we get \( b \neq \text{seg} \). Hence, we obtain \( m_\sigma(q) = b \) from (P2). Let \( c = m_r(b, \text{next}) \) and let \( c' = m_\sigma(b, \text{next}) \). 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).

\textbf{Ad (G2)} for \( b, \text{next} \in \text{valid}_r \). By (P2) and Lemma B.37, \( b, \text{next} \in \text{valid}_r \). So, \( c = c' \). We get
\[
m_{r, \text{act}}|_{\text{valid}_r, \text{act}} = m_r[p \mapsto c]|_{\text{valid}_r, \cup\{p\}} = (m_r|_{\text{valid}_r})[p \mapsto c].
\]

The second equality holds by \( m_r(b, \text{next}) = c \). That is, we preserve \( c \).data in \( m_r|_{\text{valid}_r} \) and only need to update \( p \). Similarly, \( m_{\sigma, \text{act}}'|_{\text{valid}_r, \text{act}'} = (m_\sigma|_{\text{valid}_r})[p \mapsto c'] \). Then, we conclude by \( m_r|_{\text{valid}_r} = m_\sigma|_{\text{valid}_r} \) from (P2) together with \( c = c' \).

\textbf{Ad (G2)}. By (P2) together with Lemma B.37, we have \( b, \text{next} \notin \text{valid}_r \). We get
\[
m_{r, \text{act}}|_{\text{valid}_r, \text{act}} = m_r[p \mapsto c]|_{\text{valid}_r, \setminus\{p\}} = m_r|_{\text{valid}_r, \setminus\{p\}}.
\]

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 \( m_{\sigma, \text{act}}'|_{\text{valid}_r, \text{act}'} = m_\sigma|_{\text{valid}_r, \setminus\{p\}} \). We conclude by:
\[
m_r|_{\text{valid}_r, \setminus\{p\}} = (m_r|_{\text{valid}_r})|_{\text{valid}_r, \setminus\{p\}} = (m_\sigma|_{\text{valid}_r})|_{\text{valid}_r, \setminus\{p\}} = m_\sigma|_{\text{valid}_r, \setminus\{p\}}.
\]

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.

\textbf{Ad (G3)}. By (P3) together with the fact that \text{act} and \text{act}' do not emit an event, it is sufficient to show \( \text{adr}(m_{r, \text{act}}|_{\text{valid}_r, \text{act}}) \subseteq \text{adr}(m_r|_{\text{valid}_r}) \). This follows from Lemma B.47.

\textbf{Ad (G4)}. Let \( a \in A \). Only the valuation of \( p \) is changed by both \text{act} and \text{act}'. So by (P4) it suffices to show that \( m_{r, \text{act}}(p) = a \Longleftrightarrow m_{\sigma, \text{act}}(p) = a \) holds. We conclude by:
\[
m_{r, \text{act}}(p) = a \iff m_r(b, \text{next}) = a \iff m_\sigma(b, \text{next}) = a \iff m_{\sigma, \text{act}}(p) = a
\]

where the first/last equivalence holds due to the update \( up/up' \). The second equivalence is due to (P5) because \( q \in \text{valid}_r \) from above yields \( b = m_r(q) \in m_r(\text{valid}_r) \).
\[\begin{align*}
\text{Case 3: } com &\equiv p.s\text{el := } y \\
&
\text{By assumption, we have } p \in valid_r \text{ and thus } p \in valid_s \text{ by (P2) together with Lemma B.37. Then, the claim follows analogously to the previous case.}
\end{align*}\]

\[\begin{align*}
\text{Case 4: } com &\equiv u := op(u_1, \ldots, u_n) \\
&
\text{Let } d_i = m_r(u_i). \text{ Then, } up = [u \mapsto d] \text{ with } d = op(d_1, \ldots, d_n). \text{ We have } u_i \in dom(m_r \mid valid_r) \text{ by definition. So (P2) yields } m_{\sigma}(u_i) = d_i. \text{ We choose } up' = [u \mapsto d]. \text{ Then, (G0) holds by the choice of } up' \text{ together with (P1). The remaining (G1) to (G7) follow immediately from (P1) to (P7).}
\end{align*}\]

\[\begin{align*}
\text{Case 5: } com &\equiv u := q.d\text{ata} \\
&
\text{By assumption, } q \in valid_r. \text{ Let } b = m_r(q). \text{ By definition, } b \neq seg. \text{ Let } d = m_r(b.d\text{ata}). \text{ The update is } up = [u \mapsto d]. \text{ By definition, } b.d\text{ata} \in dom(m_r \mid valid_r). \text{ So (P2) and Lemma B.37 together with (P2) yields } m_{\sigma}(q) = b \text{ and } m_{\sigma}(b.d\text{ata}) = d. \text{ We choose } up' = up. \text{ Then, (G0) holds by the choice of } up' \text{ together with (P1). The remaining (G1) to (G7) follow immediately from (P1) to (P7).}
\end{align*}\]

\[\begin{align*}
\text{Case 6: } com &\equiv p.d\text{ata} := u' \\
&
\text{By assumption, we have } p \in valid_r \text{ and thus } p \in valid_s \text{ by (P2) together with Lemma B.37. Then, the claim follows analogously to the previous case.}
\end{align*}\]

The above case distinction concludes the claim. \(\Box\)

**Proof C.48** (Lemma B.68). Let \(\tau.\text{act} \in O\[P\]_{\text{Adr}}^A\) and \(\sigma.\text{act} \in O\[P\]_{\text{Adr}}^A\) with \(\tau \sim \sigma, \tau \preceq_A \sigma,\) and \(\tau \prec_A \sigma.\) Let act = \(\{t, com, up\}.\) Unrolling the premise gives:

\[\begin{align*}
\text{(P0)} & \quad \sigma.\text{act} \in O\[P\]_{\text{Adr}}^A \\
\text{(P1)} & \quad \text{ctrl}(\tau) = \text{ctrl}(\sigma) \\
\text{(P2)} & \quad m_r \mid valid_r = m_r \mid valid_s \\
\text{(P3)} & \quad \forall b \in \text{adr}(m_r \mid valid_r) \cup A. \ F_\text{O}(\tau, b) \subseteq F_\text{O}(\sigma, b) \\
\text{(P4)} & \quad \forall a \in A \forall p \in P\text{Var}. \ m_r(p) = a \iff m_\sigma(p) = a \\
\text{(P5)} & \quad \forall a \in A \forall b \in m_r(\text{valid}_r). \ m_r(b.\text{next}) = a \iff m_\sigma(b.\text{next}) = a
\end{align*}\]
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) \( \forall b \in \text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) \cup A. \ F_{\sigma}(\tau.\text{act}, b) \subseteq F_{\sigma}(\sigma.\text{act}, b) \)

(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 \( \text{act} \) executes the same command \( \text{com} \) as \( \text{act} \), we get (G1) from (P0) and (P1) holds. We do not comment on (G1) in the following.

\[ \triangleright \text{Case 1: } \text{com} \equiv \text{in}:\text{func}(r_{1}, \ldots, r_{n}) \]

The update is \( \text{up} = \emptyset \). We choose \( \text{up}' = \text{up} \). 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, \( \text{act} \) emits \( \text{evt} \in \text{in}:\text{func}(t, v_{1}, \ldots, v_{n}) \) after both \( \tau \) and \( \sigma \).

This means that we have \( H(\tau.\text{act}) = H(\tau).\text{evt} \) as well as \( H(\sigma.\text{act}) = H(\sigma).\text{evt} \). Thus, (G3) follows from (P3) together with Lemma B.41.

\[ \triangleright \text{Case 2: } \text{com} \equiv \text{re}:\text{func} \]

Analogous to the previous case.

\[ \triangleright \text{Case 3: } \text{com} \equiv \text{assume } p = q \]

The update is \( \text{up} = \emptyset \). 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).

\[ \triangleright \text{Ad (G2).} \] 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 \( \text{valid}_{\tau.\text{act}} = \text{valid}_{\tau} \cup \{ p, q \} \) by definition. So we get \( \text{valid}_{\sigma.\text{act}} = \text{valid}_{\sigma} \cup \{ p, q \} \) by (P2) together with Lemma B.37. From
Lemma B.47 we get $m_{r,\text{act}}(\text{valid}_{r,\text{act}}) = m_r(\text{valid}_r)$ and $m_{r,\text{act}}(\text{valid}_{r,\text{act}}) = m_r(\text{valid}_r)$. Combined with (P2), we obtain:

$$\text{dom}(m_{r,\text{act}}|_{\text{valid}_{r,\text{act}}}) = \text{valid}_{r,\text{act}} \cup \text{DVar} \cup \{ c.\text{data} \mid c \in m_{r,\text{act}}(\text{valid}_{r,\text{act}}) \}$$

Let $\exp \in \text{dom}(m_{r,\text{act}}|_{\text{valid}_{r,\text{act}}})$. It remains to show $m_{r,\text{act}}(\exp) = m_{\sigma,\text{act}}(\exp)$. By the above, we have $\exp \in \text{dom}(m_r|_{\text{valid}_r}) \cup \{ p, q \}$. So by (P2) together with $m_{r,\text{act}} = m_r$ and $m_{\sigma,\text{act}} = m_\sigma$, it remains to establish $m_{r,\text{act}}(p) = m_{\sigma,\text{act}}(p)$ and $m_{r,\text{act}}(q) = m_{\sigma,\text{act}}(q)$.

Wlog, $p \in \text{valid}_r$. Then, we have $m_r(p) = m_\sigma(p)$. Hence, $m_{r,\text{act}}(p) = m_{\sigma,\text{act}}(p)$. And the assumption in $\text{com}$ requires $p$ and $q$ to have the same valuation, $m_{r,\text{act}}(q) = m_{\sigma,\text{act}}(q)$.

This concludes the claim.

\begin{itemize}
  \item Ad (G3). We have $\mathcal{F}_\sigma(\tau, c) = \mathcal{F}_\sigma(\sigma, c)$ as well as $\mathcal{F}_\sigma(\sigma, c) = \mathcal{F}_\sigma(\sigma, c)$ for all $c \in \text{Adr}$ since $\text{act}$ does not emit an event. Thus, $\text{adr}(m_{r,\text{act}}|_{\text{valid}_{r,\text{act}}}) \subseteq \text{adr}(m_r|_{\text{valid}_r})$, by Lemma B.47. Then, we conclude by (P3).
  \item Ad (G5). Follows from $m_{r,\text{act}} = m_r$ and $m_{\sigma,\text{act}} = m_\sigma$, together with (P5) and Lemma B.47.
\end{itemize}

\begin{itemize}
  \item Case 4: $\text{com} \equiv \text{assume } p \neq q$
  Since $\text{act}$ has no effect on $\tau$ and $\sigma$, (G1) to (G7) follow immediately from (P1) to (P7).
  \item Case 5: $\text{com} \equiv \text{assume } \text{pred}($\text{false}$)$
  Since $\text{act}$ has no effect on $\tau$ and $\sigma$, (G1) to (G7) follow immediately from (P1) to (P7).
  \item Case 6: $\text{com} \equiv \text{assume } \text{malloc}$
  Let $b = m_{r,\text{act}}(p)$. The update is $\text{up} = [p \mapsto b, b.\text{next} \mapsto \text{seg}, b.\text{data} \mapsto d]$ for some $d$. By definition, we have $b \in \text{fresh}_r \cup \text{fresh}_\sigma$. By (P0) we have $b \in \text{fresh}_r \cup \text{fresh}_\sigma$. Note that the following holds by assumption:

$$b \notin A \implies \mathcal{F}_\sigma(\tau, b) \subseteq \mathcal{F}_\sigma(\sigma, b) \quad (16)$$

Before we establish the remaining properties, we show two auxiliary statements:

$$m_{r,\text{act}}(\text{valid}_{r,\text{act}}) = m_r(\text{valid}_r \setminus \{ p, b.\text{next} \}) \cup \{ b \} \quad (17)$$

$$m_{\sigma,\text{act}}(\text{valid}_{\sigma,\text{act}}) = m_\sigma(\text{valid}_\sigma \setminus \{ p, b.\text{next} \}) \cup \{ b \} \quad (18)$$

\textbf{Section C.3 Reductions}
\[ Ad (17). \] Note that \textit{act} changes only the valuation of \textit{p} and \textit{b.next}. Moreover, by definition, we have \( \text{valid}_{\tau, \text{act}} = \text{valid}_{\tau} \cup \{ p, b.next \} \). So we conclude as follows:

\[
m_{\tau, \text{act}}(\text{valid}_{\tau, \text{act}}) = m_{\tau, \text{act}}(\text{valid}_{\tau} \cup \{ p, b.next \})
= m_{\tau, \text{act}}(\text{valid}_{\tau} \setminus \{ p, b.next \}) \cup m_{\tau, \text{act}}(\{ p, b.next \})
= m_{\tau}(\text{valid}_{\tau} \setminus \{ p, b.next \}) \cup \{ b \}.
\]

\[ Ad (18). \] Follows analogously to (17).

\[ Ad (G2). \] Using (17) and \( \text{valid}_{\tau, \text{act}} = \text{valid}_{\tau} \cup \{ p, b.next \} \), we get:

\[
dom(m_{\tau, \text{act}}|_{\text{valid}_{\tau, \text{act}}}) = \text{valid}_{\tau, \text{act}} \cup \text{DVar} \cup \{ c.data | c \in m_{\tau, \text{act}}(\text{valid}_{\tau, \text{act}}) \}
= (\text{valid}_{\tau} \setminus \{ p, b.next \}) \cup \{ p, b.next \} \cup \text{DVar}
\cup \{ c.data | c \in m_{\tau}(\text{valid}_{\tau} \setminus \{ p, b.next \}) \} \cup \{ b.data \}
= dom(m_{\tau}|_{\text{valid}_{\tau} \setminus \{ p, b.next \}}) \cup \{ p, b.next, b.data \}.
\]

By definition then, we have:

\[
m_{\tau, \text{act}}|_{\text{valid}_{\tau, \text{act}}} = (m_{\tau}|_{\text{valid}_{\tau} \setminus \{ p, b.next \}})[p \mapsto b, b.next \mapsto \text{seg}, b.data \mapsto d]
= ((m_{\tau}|_{\text{valid}_{\tau}})|_{\text{valid}_{\tau} \setminus \{ p, b.next \}})[p \mapsto b, b.next \mapsto \text{seg}, b.data \mapsto d].
\]

Along the same lines, using (18), we get:

\[
m_{\sigma, \text{act}}|_{\text{valid}_{\sigma, \text{act}}} = ((m_{\sigma}|_{\text{valid}_{\sigma}})|_{\text{valid}_{\sigma} \setminus \{ p, b.next \}})[p \mapsto b, b.next \mapsto \text{seg}, b.data \mapsto d].
\]

The above equalities now allow us to conclude the claim using (P2) and (P2) together with Lemma B.37:

\[
(m_{\tau}|_{\text{valid}_{\tau}})|_{\text{valid}_{\tau} \setminus \{ p, b.next \}} = (m_{\sigma}|_{\text{valid}_{\sigma}})|_{\text{valid}_{\sigma} \setminus \{ p, b.next \}}.
\]

\[ Ad (G3). \] By Lemma B.47, we have \( \text{adr}(m_{\tau, \text{act}}|_{\text{valid}_{\tau, \text{act}}}) \subseteq \text{adr}(m_{\tau}|_{\text{valid}_{\tau}}) \cup \{ b \} \). Moreover, we have \( \mathcal{FO}(\tau, \text{act}, c) = \mathcal{FO}(\tau, c) \) and \( \mathcal{FO}(\sigma, \text{act}, c) = \mathcal{FO}(\sigma, c) \) for all \( c \in \text{Adr} \) since \textit{act} does not emit an event. By (P3) it thus remains to show that \( \mathcal{FO}(\tau, b) \subseteq \mathcal{FO}(\sigma, b) \). If \( b \in A \), then we get the desired inclusion from (P3). Otherwise, we get it from (16).

\[ Ad (G4). \] We have \( m_{\tau, \text{act}}(p) = m_{\sigma, \text{act}}(p) \). For \( q \in PVar \setminus \{ p \} \), we have \( m_{\tau, \text{act}}(q) = m_{\tau}(q) \) as well as \( m_{\sigma, \text{act}}(q) = m_{\sigma}(q) \). Hence, the claim follows from (P4).

\[ Ad (G5). \] We have \( m_{\tau, \text{act}}(b.next) = m_{\sigma, \text{act}}(b.next) \). For \( c.next \in PSel \setminus \{ b.next \} \), we obtain then \( m_{\tau, \text{act}}(c.next) = m_{\tau}(c.next) \) and \( m_{\sigma, \text{act}}(c.next) = m_{\sigma}(c.next) \). We conclude by (P5).
Ad (G6). We have \( \text{fresh}_{r, \text{act}} = \text{fresh}_r \setminus \{ b \} \) and \( \text{freed}_{r, \text{act}} = \text{freed}_r \setminus \{ b \} \). Similarly, we have \( \text{fresh}_{\sigma, \text{act}} = \text{fresh}_\sigma \setminus \{ b \} \) and \( \text{freed}_{\sigma, \text{act}} = \text{freed}_\sigma \setminus \{ b \} \). We conclude by (P6).

Ad (G7). We have \( \text{retired}_{r, \text{act}} = \text{retired}_r \) and \( \text{retired}_{\sigma, \text{act}} = \text{retired}_\sigma \). Then, the claim follows from (P7).

Case 7: \( \text{com} \equiv \text{free}(b) \)

The update is \( \text{up} = \emptyset \). By assumption, we have:

\[
\forall c \in \text{Adr} \setminus \{ b \}. \quad \mathcal{F}_\mathcal{O}(r.\text{act}, c) = \mathcal{F}_\mathcal{O}(\tau, c) \tag{19}
\]

\[
\forall c \in \text{Adr} \setminus \{ b \}. \quad \mathcal{F}_\mathcal{O}(\sigma.\text{act}, c) = \mathcal{F}_\mathcal{O}(\sigma, c) \tag{20}
\]

Ad (G2). We have \( \text{valid}_{r, \text{act}} = \text{valid}_{\sigma, \text{act}} \). To see this, consider \( \text{pexp} \in \text{valid}_{r, \text{act}} \). By definition, this means \( \text{pexp} \in \text{valid}_r \) and \( \text{pexp} \neq b.\text{next} \) and \( m_r(\text{pexp}) \neq b \). From Lemma B.37 and (P2) we obtain \( \text{pexp} \in \text{valid}_\sigma \). By (P2), \( m_\sigma(\text{pexp}) = m_r(\text{pexp}) \neq b \). Hence, \( \text{pexp} \in \text{valid}_{\sigma, \text{act}} \) holds by definition. This establishes \( \text{valid}_{r, \text{act}} \subseteq \text{valid}_{\sigma, \text{act}} \). The reverse inclusion follows analogously. Next, observe that \( m_r|_{\text{valid}_{r, \text{act}}} = m_\sigma|_{\text{valid}_{r, \text{act}}} \) because of \( m_r = m_r.\text{act} \) due to \( \text{up} = \emptyset \). Along the same lines, \( m_\sigma|_{\text{valid}_{\sigma, \text{act}}} = m_\sigma|_{\text{valid}_{\sigma, \text{act}}} \).

Now, it is sufficient to show \( m_r|_{\text{valid}_{r, \text{act}}} = m_\sigma|_{\text{valid}_{\sigma, \text{act}}} \). We conclude by:

\[
m_r|_{\text{valid}_{r, \text{act}}} = (m_r|_{\text{valid}_{r, \text{act}}})(m_\sigma|_{\text{valid}_{\sigma, \text{act}}}) = m_\sigma|_{\text{valid}_{\sigma, \text{act}}}
\]

where the first holds by \( \text{valid}_{r, \text{act}} \subseteq \text{valid}_r \), the second equality holds by (P2), the third equality is shown above, and the last equality holds by \( \text{valid}_{\sigma, \text{act}} \subseteq \text{valid}_\sigma \).

Ad (G3). Let \( c \in \text{adr}(m_r|_{\text{valid}_{r, \text{act}}}) \cup A \). We have \( c \in \text{adr}(m_r|_{\text{valid}_{r, \text{act}}}) \cup A \) by Lemma B.35 together with the same reasoning as above. Let \( h \in \mathcal{F}_\mathcal{O}(r.\text{act}, c) \). We now establish that \( h \in \mathcal{F}_\mathcal{O}(\sigma.\text{act}, c) \) holds. First consider the case \( b = c \). We get \( \text{free}(c).h \in \mathcal{F}_\mathcal{O}(r, c) \) 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 \( h \in \mathcal{F}_\mathcal{O}(\sigma.\text{act}, c) \). Now, consider \( c \neq b \). We have \( h \in \mathcal{F}_\mathcal{O}(r, c) \) by (19). From (P3) we obtain \( \text{free}(c).h \in \mathcal{F}_\mathcal{O}(\sigma, c) \). Then, we get the desired \( h \in \mathcal{F}_\mathcal{O}(\sigma.\text{act}, c) \) from (20).

Ad (G4). The update \( \text{up} \) 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 \( m_r = m_{r, \text{act}} \) and \( m_\sigma = m_{\sigma, \text{act}} \) as well as \( \text{valid}_{r, \text{act}} \subseteq \text{valid}_r \) and \( \text{valid}_{\sigma, \text{act}} \subseteq \text{valid}_\sigma \).

Ad (G6). Follows from (P6) because by definition we have \( \text{fresh}_{r, \text{act}} = \text{fresh}_r \setminus \{ b \} \) as well as \( \text{freed}_{r, \text{act}} = \text{freed}_r \cup \{ b \} \) and similarly for \( \sigma \).
Appendix C

Case 8: \( \text{com} \equiv \text{env}(b) \)

The update is \( \text{up} = [b.\text{next} \mapsto \text{seg}, b.\text{data} \mapsto d] \) for some \( d \). We have \( b \in \text{fresh}_r \cup \text{freed}_r \) by definition. By (P0), we have \( b \in \text{fresh}_r \cup \text{freed}_r \). Then, (G4), (G6) and (G7) follow immediately from (P4), (P6) and (P7). For the remaining properties, we show the auxiliary statements:

\[
\begin{align*}
\text{m}_{r, \text{act}}(\text{valid}_{r, \text{act}}) &= \text{m}_{r}(\text{valid}_r) \setminus \{b\} \quad \text{(21)} \\
\text{adr}(\text{m}_{r, \text{act}}|_{\text{valid}_{r, \text{act}}}) &= \text{m}_{r}(\text{valid}_r) \setminus \{b\} \quad \text{(22)}
\end{align*}
\]

\( \diamond \) Ad (G7). Follows from (P7) because \( \text{retired}_{r, \text{act}} = \text{retired}_r \setminus \{b\} \) and similarly for \( \sigma \).

\( \diamond \) Ad (G2). As before, Lemmas B.43 and B.48 yield \( b.\text{next} \notin \text{valid}_r \). Then, we conclude by Lemma B.35 together with \( \text{valid}_{r, \text{act}} = \text{valid}_r \) and (21) as follows:

\[
\begin{align*}
\text{adr}(\text{m}_{r, \text{act}}|_{\text{valid}_{r, \text{act}}}) &= (\text{valid}_{r, \text{act}} \cap \text{Adr}) \cup \text{m}_{r, \text{act}}(\text{valid}_{r, \text{act}}) \\
&= (\text{valid}_r \cap \text{Adr}) \cup (\text{m}_r(\text{valid}_r) \setminus \{b\}) \\
&= (\text{valid}_r \cap \text{Adr}) \cup (\text{valid}_r \cap \text{Adr}) \cup (\text{m}_r(\text{valid}_r) \setminus \{b\}) \\
&= (\text{valid}_r \cap \text{Adr}) \cup (\text{m}_r(\text{valid}_r) \setminus \{b\}) \setminus \{b\} = \text{adr}(\text{m}_{r}|_{\text{valid}_r}) \setminus \{b\}.
\end{align*}
\]

\( \diamond \) Ad (G3). Follows from (22) and (P3) together with \( \text{act} \) not emitting an event.

\( \diamond \) Ad (G5). Follows from (21) and (P5) together with \( \text{up} \) updating only the selectors of \( b \).

We get:

\[
\begin{align*}
\text{m}_{r, \text{act}}(\text{valid}_{r, \text{act}}) &= \text{m}_{r}(\text{valid}_r) \setminus \{b\} \\
\text{m}_{r, \text{act}}(\text{valid}_{r, \text{act}}) &= \text{m}_{r, \text{act}}(\text{valid}_r) \setminus \{b.\text{next}\}
\end{align*}
\]

It remains to show that \( b \notin m_r(\text{valid}_r) \). This follows from Lemmas B.43 and B.48.
We proceed by induction over the structure of \( \sigma \) we get

The above case distinction concludes the claim. ■

**Proof C.49** (Lemma B.69). Let \( \tau_1.\text{act}_{.2} \in \mathcal{O}[\mathbb{P}]_X^\gamma \) UAF with \( \text{act} = \langle \bot, \text{env}(a), \text{up} \rangle \). The update is of the form \( \text{up} = [a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d] \) for some \( d \). By definition, \( a \in \text{fresh}_r \cup \text{freed}_r \).

We proceed by induction over the structure of \( \tau_2 \). In the base case, \( \tau_2 = \epsilon \). We get \( \tau_1 \in \mathcal{O}[\mathbb{P}]_X^\gamma \) by definition. Consider \( \exp \) with \( m_{\tau_1.\text{act}}(\exp) \neq m_{\tau_1}(\exp) \). Because of the update \( \text{up} \), we know that \( \exp \in \{ a.\text{next}, a.\text{data} \} \). This means \( \exp \cap \text{Adr} = \{ a \} \). Then, \( a \in \text{fresh}_{\tau_1} \cup \text{freed}_{\tau_1} \). Hence, we get \( a \in \text{fresh}_{\tau_1.\text{act}} \cup \text{freed}_{\tau_1.\text{act}} \) by definition. The remaining properties follow immediately by definition. That is, \( \tau_1 \) satisfies the claim. For the induction step, consider \( \tau_1.\text{act}_{.2}.\text{act}' \in \mathcal{O}[\mathbb{P}]_X^\gamma \) UAF and assume we have already shown the desired correspondence between \( \tau = \tau_1.\text{act}_{.2} \) and \( \tau' = \tau_1.\text{act}' \). We now establish the correspondence between \( \tau.\text{act}' \) and \( \tau'.\text{act}' \). Note that by the semantics, \( \text{env}(a) \) does not affect the control location of threads, \( \text{ctrl}(\tau.\text{act}') = \text{ctrl}(\tau'.\text{act}') \) by definition; we do not comment on this property hereafter. Let \( \text{act}' = \langle t, \text{com}, \text{up}' \rangle \).

\[ \diamond \text{Case 1: com is an assignment} \]

We focus on the case \( \text{com} \equiv p := q.\text{next} \). The remaining cases of assignments follow analogously. Let \( b = m_{\tau}(q) \). By definition, \( b \neq \text{seg} \). Let \( c = m_{\tau}(q.\text{next}) \). The update is of the form \( \text{up}' = [p \mapsto c] \). Note that we have \( p \cap \text{Adr} = \emptyset \). We obtain \( m_{\tau}(q) = m_{\tau}(q) = b \) from induction. Moreover, \( q \in \text{valid} \) follows from \( \tau.\text{act}' \) is UAF by assumption. Lemmas B.43 and B.48 yield \( b \notin \text{fresh}_r \cup \text{freed}_r \). Hence, we get \( m_{\tau}(b.\text{next}) = c \) and thus \( \tau'.\text{act}' \in \mathcal{O}[\mathbb{P}]_X^\gamma \).

Since \( \text{act}' \) 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 \( m_{\tau.\text{act}}(p) = m_{\tau'.\text{act}}(p) \) due to the executed update \( \text{up}' \), we must have \( p \neq \exp \).

By definition, this means \( m_{\tau.\text{act}}(\exp) = m_{\tau}(\exp) \) and \( m_{\tau'.\text{act}}(\exp) = m_{\tau}(\exp) \). Then, we conclude by induction:

\[ (\exp \cap \text{Adr}) \cap (\text{fresh}_{\tau.\text{act}} \cup \text{freed}_{\tau.\text{act}}) = (\exp \cap \text{Adr}) \cap (\text{fresh}_r \cup \text{freed}_r) \neq \emptyset. \]

\[ \diamond \text{Case 2: com} \equiv p := \text{malloc} \]

Let \( b = m_{\tau.\text{act}}(p) \). By definition, \( b \in \text{fresh}_r \cup (\text{fresh}_r \cap Y) \). So \( b \in \text{fresh}_r \cup (\text{freed}_r \cap Y) \) by induction. This means \( \tau'.\text{act}' \in \mathcal{O}[\mathbb{P}]_X^\gamma \). By induction and definition, we get:

\[ \text{fresh}_{\tau.\text{act}} = \text{fresh}_r \setminus b = \text{fresh}_r \setminus b = \text{fresh}_r \]

and

\[ \text{freed}_{\tau.\text{act}} = \text{freed}_r \setminus b = \text{freed}_r \setminus b = \text{freed}_r \]

and

\[ \text{retired}_{\tau.\text{act}} = \text{retired}_r \setminus b = \text{retired}_r \setminus b = \text{retired}_r. \]

The remaining property, the memory correspondence, follows as in the previous case.
Appendix C

Case 3: \( \text{com} \equiv \text{in}: \text{func}(r_1, \ldots, r_n) \)

We have \( r_i \cap \text{Adr} = \emptyset \). Hence, \( m_r(r_i) = v_i = m_{r'}(r_i) \). Together with induction, this means:

\[
\mathcal{H}(r.\text{act}) = \mathcal{H}(r).\text{in}: \text{func}(t, v_1, \ldots, v_n) = \mathcal{H}(r').\text{in}: \text{func}(t, v_1, \ldots, v_n) = \mathcal{H}(r'.\text{act}) .
\]

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: \( \text{com} \equiv \text{re}: \text{func} \)

Analogous to the previous case.

Case 5: \( \text{com} \equiv \text{assume} \ cond \)

As before, we have \( m_r(x) = m_{r'}(x) \) for any variable \( x \in \text{PVar} \cup \text{DVar} \) that appears in \( \text{cond} \). Hence, condition \( \text{cond} \) has the same truth value after \( r \) and \( r' \). This means \( r'.\text{act} \in \mathcal{O}[P]_X^Y \). 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: \( \text{com} \equiv \text{free}(b) \)

The update is \( \text{up} = \emptyset \). By definition, we have \( \mathcal{H}(r).\text{free}(b) \in \mathcal{S}(O) \). Hence, induction gives \( \mathcal{H}(r').\text{free}(b) \in \mathcal{S}(O) \). This means \( r'.\text{act} \in \mathcal{O}[P]_X^Y \). By induction and definition, we have:

\[
\text{fresh}_{r.\text{act}} = \text{fresh}_r \setminus b = \text{fresh}_{r'} \setminus b = \text{fresh}_{r'},
\]

and

\[
\text{freed}_{r.\text{act}} = \text{freed}_r \cup b = \text{freed}_{r'} \cup b = \text{freed}_{r'},
\]

and

\[
\text{retired}_{r.\text{act}} = \text{retired}_r \setminus b = \text{retired}_{r'} \setminus b = \text{retired}_{r'}.
\]

For the remaining property, consider \( \text{exp} \) with \( m_{r.\text{act}}(\text{exp}) \neq m_{r'.\text{act}}(\text{exp}) \). Since \( m_{r.\text{act}} = m_r \) and \( m_{r'.\text{act}} = m_{r'} \), we have \( m_r(\text{exp}) \neq m_{r'}(\text{exp}) \). By induction, this means that \( \text{exp} \) is of the form \( c.\text{sel} \) and \( c \in \text{fresh}_r \cup \text{freed}_r \). By the above, \( c \in \text{fresh}_{r.\text{act}} \cup \text{freed}_{r.\text{act}} \). This concludes the desired memory correspondence.

Case 7: \( \text{com} \equiv \text{env}(b) \)

We have \( \text{up}' = [b.\text{next} \leftrightarrow \text{seg}, b.\text{data} \mapsto d] \) for some \( d \). By definition, \( b \in \text{fresh}_r \cup \text{freed}_r \). This means \( b \in \text{fresh}_{r'} \cup \text{freed}_{r'} \) by induction. So \( r'.\text{act} \in \mathcal{O}[P]_X^Y \). The remaining properties follow similarly to the previous case.

Case 8: \( \text{com} \in \{ \text{skip}, \text{beginAtomic}, \text{endAtomic}, @\text{inv} \bullet \} \)

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 $\tau \in \mathcal{O}[P]^Y_X$ UAF and let $a \in \text{retired}_\tau \cap \text{freed}_\tau$. We show that there is a double retire in $\mathcal{O}[P]^Y_X$. To that end, we first show that $\tau$ is of the following form:

$$
\tau = \tau_1.\{t_1, \text{retire}(p_1), up_1\}.\tau_2.\{t_2, \text{free}(a), up_2\}.\tau_3.\{t_3, \text{retire}(p_3), up_3\}.\tau_4
$$

with $m_{\tau_1}(p_1) = a = m_{\tau_1,\tau_2}(p_3)$ and $a \not\in \text{frees}_{\tau_3} \cup \text{freed}_{\tau_3} \cup \text{freed}_{\tau_4}$.

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 free of $a$ in $\tau$, 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 $a \not\in \text{retired}_\tau$. Let $act_3$ be the first retirement of $a$ after $act_2$. By Lemma B.46, we must have $a \in \text{retired}_{\tau_3}$. That is, there must be a retirement of $a$ prior to $act_2$. Let $act_1$ be the latest such retirement. Next, observe that $a \not\in \text{freed}_{\tau_3}$ must hold. If this was not the case, we would get $a \not\in \text{retired}_{\tau_3}$ since $a$ is not retired in $\tau_3$ by choice of $act_1$. We already showed $a \in \text{retired}_{\tau_3}$ so that we obtain $a \not\in \text{freed}_{\tau_3}$ indeed. Lastly, $a \not\in \text{frees}_{\tau_3}$ and $a \not\in \text{frees}_{\tau_4}$ follows from the choice of $act_2$ being the last free of $a$ in $\tau$.

Now, consider $\gamma = \tau_1.\text{act}_1.\tau_2.\text{act}_2.\tau_3.\tau_4'$ where $\tau_3'$ corresponds to $\gamma_3$ with all actions removed that execute command $\text{env}(a)$. By Lemma B.69 we have $\gamma \in \mathcal{O}[P]^Y_X$ and $m_{\gamma}(p_3) = m_{\tau_1,\tau_2}(p_3) = a$. Note that $a \not\in \text{freed}_{\tau_3}$. Finally, construct computation $\gamma' = \tau_1.\text{act}_1.\tau_2.\tau_3'$ which corresponds to $\gamma$ up to the removal of $\text{act}_2$. Since $\text{act}_2$ 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 $a \not\in \text{freed}_{\tau_3}$. So by definition, this means $\gamma'$ performs a double retire.

It remains to show that $\gamma' \in \mathcal{O}[P]^Y_X$ holds. First, observe that $\text{act}_2$ 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 $a \not= b$ remains unaffected whether or not $\text{free}(a)$ is executed. Combining these two properties, we conclude that an action of $\tau_3'$ may no longer be enabled only if it requires $a$ to be free. Since $\tau_3$ 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 $\gamma'$ in the original prefix of $\tau$, that is, $a \not\in \text{freed}_{\tau_1,\tau_2}$. Because we have $a \not\in \text{frees}_{\tau_1} \cup \text{freed}_{\tau_1}$, we arrive at $a \not\in \text{freed}_{\tau_2}$ which contradicts the assumption. Altogether, this means that the enabledness of all actions in $\tau_3$ does not rely on $\text{act}_2$. Hence, $\gamma' \in [X]^Y$.

Proof C.51 (Theorem 7.20). Let $\mathcal{O}$ support elision and let $\mathcal{O}[P]^{one}_{\text{Adr}}$ be free from pointer races, double retires, and harmful ABAs. We proceed by induction over the structure of $\tau$. In the base case, $\tau = \epsilon$. Choose $\sigma = \tau$. This satisfies the claim. For the induction step, consider some $\tau.\text{act} \in \mathcal{O}[P]^Y_X$. PRF and assume we have already constructed, for all addresses $a \in \text{Adr}$, some $\sigma \in \mathcal{O}[P]^{(a)}_{\text{Adr}}$ with $\tau \sim \sigma$, $\tau \preceq \sigma$, and $\tau \preceq_a \sigma$. Let $a \in \text{Adr}$. Let $\text{act} = \{t, \text{com}, \text{up}\}$. We
construct some $\hat{\sigma} \in \mathcal{O}[P]\{\text{adr}\}$ such that $\tau.\text{act} \sim \hat{\sigma}$, $\tau.\text{act} \equiv_{a} \hat{\sigma}$, and $\tau.\text{act} < \hat{\sigma}$. To that end, we do a case distinction over $\text{com}$.

\[\text{Case 1:} \text{ com is an assignment}\]

Our goal is to find some $\text{act}'$ such that $\hat{\sigma} = \sigma.\text{act}'$ satisfies the requirements. We obtain $\text{act}'$ 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 $\text{com}$ contains $p.\text{sel}$ then $p \in \text{valid}_r$. So, assume $\text{com}$ contains $p.\text{sel}$ as nothing needs to be shown otherwise. We proceed as follows: we show that $\text{com}$ is enabled after $\sigma$ and then use pointer race freedom to establish the validity of $p$.

By Lemma B.50 we have $m_{\sigma}(p) \neq \bot$. That is, $m_{\sigma}(p) \in $Adr$ \cup \{\text{seg}\}$. Towards a contradiction, assume $m_{\sigma}(p) = \text{seg}$. By Lemma B.49, we have $p \in $valid$_r$. Then, $m_{\tau}(p) = \text{seg}$ follows from $\tau \sim \sigma$. This means $\tau.\text{act}$ is not enabled after $\tau$. This contradicts $\tau.\text{act} \in \mathcal{O}[P]\{\text{adr}\}$. Hence, we must have $m_{\sigma}(p) \neq \text{seg}$. So, $\text{com}$ is enabled after $\sigma$. This means there is an update $\text{up}^i$ such that $\text{act}'' = \{t, \text{com}, \text{up}^i\}$ is enabled after $\sigma$, that is, $\sigma.\text{act}" \in \mathcal{O}[P]\{\text{adr}\}$. (Note that $\text{act}''$ is not necessarily the desired $\text{act}'$.) Because $\mathcal{O}[P]\{\text{adr}\}$ is free from pointer races by assumption, we must have $p \in \text{valid}_r$. So, $\tau \in \text{valid}_r$ by $\tau \sim \sigma$ together with Lemma B.37. Altogether, we can invoke Lemma B.67 for $\tau.\text{act}$ and $\sigma$. We obtain $\text{act}'$ such that $\hat{\sigma} = \sigma.\text{act}'$ satisfies the claim.

\[\text{Case 2:} \text{ com} \equiv \text{in}:\text{func}(r_1, \ldots, r_n)\]

If $m_{\tau}(r_i) = m_{\sigma}(r_i)$ for all $1 \leq i \leq n$, then we get $\sigma.\text{act} \in \mathcal{O}[P]\{\text{adr}\}$ 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 $\text{act}$ is enabled after $\sigma$ and then show that is has the desired properties. To see enabledness of $\text{com}$, note that we have $m_{\sigma}(r_i) \in $Adr$ \cup \{\text{seg}\}$ by Lemma B.50. To the contrary, assume $m_{\sigma}(r_i) = \text{seg}$. Then, $r_i \in $valid$_s$ by Lemma B.49 and thus $m_{\tau}(r_i) = \text{seg}$ by $\tau \sim \sigma$. Since this contradicts enabledness of $\text{act}$ after $\tau$, we must have $m_{\sigma}(r_i) \neq \text{seg}$. So, $\sigma.\text{act} \in \mathcal{O}[P]\{\text{adr}\}$.

Now, we show that $\hat{\sigma} := \sigma.\text{act}$ satisfies the claim. Let $\overline{r} = r_1, \ldots, r_n$. Let $\overline{\sigma} = m_\overline{r}(\overline{r})$ and let $\overline{\sigma'} = m_\overline{r}(\overline{r})$. Let $\overline{\sigma} = \overline{v}_{r,1}, \ldots, \overline{v}_{r,n}$ and $\overline{\sigma'} = \overline{v}_{\sigma,1}, \ldots, \overline{v}_{\sigma,n}$. The main task is to show:

$$\forall c \in adr(m_{\tau.\text{act}}|_{\text{valid}^\tau_{\text{adr}}}) \cup \{a\}. F_\mathcal{O}(\tau.\text{act}, c) \subseteq F_\mathcal{O}(\sigma.\text{act}, c).$$

Let $c \in adr(m_{\tau.\text{act}}|_{\text{valid}^\tau_{\text{adr}}}) \cup \{a\}$. Because $\text{act}$ does not alter the heap nor the validity of expressions, we get $c \in adr(m_{\tau}|_{\text{valid}^\tau_{\text{adr}}}) \cup \{a\}$ by Lemma B.35. From $\tau \equiv_a \sigma$ and $\tau \preceq \sigma$ we get $F_\mathcal{O}(\tau, c) \subseteq F_\mathcal{O}(\sigma, c)$.

To see this, let $h \in F_\mathcal{O}(\mathcal{H}(\tau).\text{in}:\text{func}(t, \overline{\sigma'}), c)$ and $\mathcal{H}(\tau).\text{in}:\text{func}(t, \overline{\sigma'}) c$. So, $\text{in}:\text{func}(t, \overline{\sigma'}).h \in F_\mathcal{O}(\mathcal{H}(\tau), c)$ by Lemma B.41. Using $\tau \equiv_a \sigma$ and $\tau \preceq \sigma$ we then obtain $\text{in}:\text{func}(t, \overline{\sigma'}).h \in F_\mathcal{O}(\mathcal{H}(\sigma), c)$. Again
by Lemma B.41, we conclude \( h \in F_O(\mathcal{H}(\sigma).in:\func(t, \overline{I}), c) \). Next, recall that \( \sigma.act \) is PRF. That is, \( \sigma.act \) is not racy. From this we get:

\[
F_O(\mathcal{H}(\sigma).in:\func(t, \overline{I}), c) \subseteq F_O(\mathcal{H}(\sigma).in:\func(t, \overline{I}), c). \tag{24}
\]

To see this, we have to show that the following holds for all \( i \) with \( 1 \leq i \leq n \):

\[
(v_{\sigma,i} = c \lor r_i \in \valid_{\sigma} \lor r_i \in \DExp) \implies v_{\tau,i} = v_{\sigma,i}
\]

If \( r_i \in \valid_{\sigma} \), 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 \( r_i \in \DExp \) because this means \( r_i \in DVar \) and \( DVar \subseteq \text{dom}(m_\sigma | \valid_{\sigma}) \) by definition. So consider the case where \( v_{\sigma,i} = c \) and \( r_i \notin \valid_{\sigma} \cup \DExp \). To the contrary, assume \( c \neq a \). By the choice of \( c \), we get \( c \in \adrx(m_\sigma | \valid_{\sigma}) \). Moreover, \( c \in m_\sigma(\PVar \setminus \valid_{\sigma}) \) follows from \( m_\sigma(r) = v_{\sigma,i} = c \) and \( r_i \notin \valid_{\sigma} \). 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 \( \tau \leq_a \sigma \) and \( r_i \in \PVar \). 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:

\[
F_O(\tau.act, c) = F_O(\mathcal{H}(\tau).in:\func(t, \overline{I}), c) \subseteq F_O(\mathcal{H}(\sigma).in:\func(t, \overline{I}), c)
\]

\[
\subseteq F_O(\mathcal{H}(\sigma).in:\func(t, \overline{I}), c) = F_O(\mathcal{H}(\sigma).in:\func(t, \overline{I}), c).
\]

Lastly, note that we have \( m_\tau(r_i) = a \) iff \( m_\sigma(r_i) = a \) because \( \tau \leq_a \sigma \). By induction, this means \( a \in \retired_{\tau.act} \) iff \( a \in \retired_{\sigma.act} \) because \( \tau \sim \sigma \) and \( \tau \leq \sigma \) if it does so after \( \sigma \). Altogether, we obtain the desired \( \tau.act \sim \sigma.act, \tau.act \leq_a \sigma.act, \) and \( \tau.act \neq \sigma.act \) from the above together with the fact that action \( \tau.act \) does not change the memory nor the validity nor the fresh/freed addresses.

\begin{itemize}
  \item \textbf{Case 3:} \( \com = \re:\func \)
  
  We choose \( \delta = \sigma.act \). By \( \tau \sim \sigma \), we know that \( \tau.act \) is enabled after \( \sigma \), that is, \( \sigma.act \in O[P]^{(a)}_{\Adr} \).
  
  By definition, \( \tau.act \) emits the same event \( \re:\func(t) \) after both \( \tau \) and \( \sigma \). Then, the claim follows similarly to the previous case.

  \item \textbf{Case 4:} \( \com = p := \malloc \)
  
  Let \( b = m_\tau.act(p) \). The update is \( \up = [p \mapsto b, b.nexx \mapsto \seg, b.data \mapsto \dd] \) for some \( \dd \). By definition, we have \( b \in \fresh_{\tau} \cup \freed_{\tau} \). If \( a = b \), then \( b \in \fresh_{\sigma} \cup \freed_{\sigma} \) holds by \( \tau \leq_a \sigma \).
  
  Hence, \( \tau.act \) is enabled after \( \sigma \), i.e., \( \sigma.act \in O[P]^{(a)}_{\Adr} \). We choose \( \delta = \sigma.act \). Then, \( \delta \) satisfies the claim by Lemma B.68. So consider the remaining case of \( a \neq b \) hereafter.

  \begin{itemize}
    \item First, we show that \( b \notin \adrx(m_\sigma | \valid_{\sigma}) \) holds. To that end, we invoke the induction for \( \tau \) and \( b \). We obtain \( \delta \in O[P]^{(b)}_{\Adr} \) with \( \tau \sim \delta \) and \( \tau \leq_b \delta \). The latter gives \( b \in \fresh_{\delta} \cup \freed_{\delta} \).
    
    Lemmas B.43 and B.48 yield \( b \notin m_{\delta}(\valid_{\delta}) \) and \( b.nexx \notin \valid_{\delta} \). (Note that we cannot
  
  \end{itemize}
\end{itemize}
conclude this directly for \( \tau \) since we do not know whether or not \( \tau \) 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 \( \tau \) and \( \delta \) yields \( b \notin \text{adr}(m_\tau | \text{valid}_\tau) \). Lemma B.38 for \( \tau \) and \( \sigma \) 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 \( \sigma \). This gives \( y \in \mathcal{O}[P]^{(x)}_{\text{adr}} \) such that \( \sigma \sim y \), \( \sigma \leq_a y \), and \( b \in \text{fresh}_y \). We have \( \tau \sim \sigma \sim y \), and \( \tau \leq_q a \sigma \leq_q y \) and \( \tau < \sigma < y \). 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 \( \tau \sim y \), \( \tau \leq_a y \), and \( \tau < y \). Moreover, \( b \in \text{fresh}_y \) means that \( \text{act} \) is enabled after \( y \). That is, \( \text{act} \in \mathcal{O}[P]^{(x)}_{\text{adr}} \). We choose \( \delta = y.\text{act} \). That \( y.\text{act} \) satisfies the claim is established by Lemma B.68. For Lemma B.68 to apply, we have to show that \( F_\mathcal{O}(\tau, b) \subseteq F_\mathcal{O}(y, b) \) holds. This, in turn, follows from Lemma B.57. We show that Lemma B.57 is enabled. We already have \( b \in \text{fresh}_y \) and \( F_\mathcal{O}(\tau, a) \subseteq F_\mathcal{O}(y, a) \) where the latter holds by \( \tau \leq_a y \). So, it remains to establish \( b \notin \text{retired}_\tau \).

Towards a contradiction, assume \( b \in \text{retired}_\tau \). By \( \tau \leq_b \delta \) then, \( b \in \text{retired}_\delta \). That is, \( \delta \) is of the form \( \delta = \delta_2.\text{act}_1.\delta_1 \) with \( \text{act}_1 \) performing \( \text{in: retire}(q) \) and \( m_{\delta_2}(q) = b \) and \( b \notin \text{fresh}_{\delta_2} \). (This decomposition chooses the latest retirement of \( b \) for \( \text{act} \)—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 \( b \in \text{fresh}_\tau \). By \( \tau \leq_b \delta \) 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 \( \mathcal{O}[P]^{(b)}_{\text{adr}} \).

This contradicts the assumption of \( \mathcal{O}[P]^{(x)}_{\text{adr}} \) 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: com \equiv \text{free}(b)**

The update is \( \text{up} = \emptyset \). By definition, \( \mathcal{H}(\tau).\text{free}(b) \in \mathcal{S}(O) \). That is, \( \text{free}(b) \in F_\mathcal{O}(\tau, b) \).

If we have \( \text{free}(b) \in F_\mathcal{O}(\sigma, b) \), we get \( \sigma.\text{act} \in \mathcal{O}[P]^{(x)}_{\text{adr}} \) and \( \mathcal{H}(\sigma).\text{free}(b) \in \mathcal{S}(O) \) by definition. Then, Lemma B.56 discharges the side condition of Lemma B.68 which yields that \( \hat{\sigma} = \sigma.\text{act} \) satisfies the claim. So consider the remaining case of \( \text{free}(b) \notin F_\mathcal{O}(\sigma, b) \) hereafter. This implies \( a \neq b \) because otherwise \( \tau \leq_a \sigma \) gives \( F_\mathcal{O}(\tau, b) \subseteq F_\mathcal{O}(\sigma, b) \) and thus \( \text{free}(b) \in F_\mathcal{O}(\sigma, b) \).

By \( \tau < \sigma \) we conclude that \( b \notin \text{adr}(m_\tau | \text{valid}_\tau) \) must hold as otherwise \( \text{free}(b) \in F_\mathcal{O}(\sigma, b) \) and thus \( \text{free}(b) \in F_\mathcal{O}(\sigma, b) \).

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 \( \text{act} \) at all.

**Ad \( \tau.\text{act} \sim \sigma \)** By definition, \( \text{ctrl}(\tau) = \text{ctrl}(\tau.\text{act}) \). So, \( \text{ctrl}(\tau.\text{act}) = \text{ctrl}(\sigma) \) by \( \tau \sim \sigma \). 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 \( \tau \sim \sigma \). To arrive at this equality, we first show \( \text{valid}_\tau = \text{valid}_{\tau.\text{act}} \). The inclusion \( \text{valid}_\tau \subseteq \text{valid}_{\tau.\text{act}} \) holds by definition. To see \( \text{valid}_\tau \subseteq \text{valid}_{\tau.\text{act}} \), 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

---

**Appendix C** Proof of Meta Theory
not hold as shown before. Moreover, we must have \( m_r(pexp) \neq b \) as otherwise we would again get \( b \in \text{adr}(m_r|_{\text{valid}_r}) \) by Lemma B.35. Hence, by the definition of validity, we have \( pexp \in \text{valid}_{r, \text{act}} \). Altogether, this means we have \( \text{valid}_r = \text{valid}_{r, \text{act}} \) indeed. Then, the desired \( m_r|_{\text{valid}_r} = m_{\text{act}, r}|_{\text{valid}_{r, \text{act}}} \) follows immediately because \( m_r = m_{\text{act}, r} \) due to \( \text{up} = \emptyset \).

\[ \diamond \text{Ad } r. \text{act} < \sigma. \] Let \( c \in \text{adr}(m_{r, \text{act}}|_{\text{valid}_{r, \text{act}}}) \). We get \( c \in \text{adr}(m_r|_{\text{valid}_r}) \) by Lemma B.47. This means \( b \neq c \) holds due to the above. Moreover, we have \( \mathcal{F}_O(r, c) \subseteq \mathcal{F}_O(\sigma, c) \) by \( r < \sigma \). So it suffices to show \( \mathcal{F}_O(r, c) = \mathcal{F}_O(r. \text{act}, c) \). This follows from Lemma B.56.

\[ \diamond \text{Ad } r. \text{act} \leq_a \sigma. \] First, recall \( b \neq a \). So \( \tau \leq_a \sigma \) gives:

\[ a \in \text{retired}_{r, \text{act}} \iff a \in \text{retired}_r \iff a \in \text{retired}_\sigma \]

and

\[ a \in \text{fresh}_{r, \text{act}} \cup \text{freed}_{r, \text{act}} \iff a \in \text{fresh}_r \cup \text{freed}_r \]

\[ \iff a \in \text{fresh}_\sigma \cup \text{freed}_\sigma. \]

Similarly to \( r. \text{act} < \sigma \), we use Lemma B.56 and \( r \leq_a \sigma \) together with \( b \neq a \) holds to obtain \( \mathcal{F}_O(r. \text{act}, a) = \mathcal{F}_O(r, a) \subseteq \mathcal{F}_O(\sigma, a) \). Since \( \text{act} \) does not modify the memory, the remaining properties follow from \( \tau \leq_a \sigma \) together with \( m_r|_{\text{valid}_r} = m_{\text{act}, r}|_{\text{valid}_{r, \text{act}}} \) which we have established for \( \tau. \text{act} \sim \sigma \) above.

\[ \diamond \text{Case 6: } \text{com} \equiv \text{env}(b) \]

The update is \( \text{up} = [b. \text{next} \mapsto \text{seg}, b. \text{data} \mapsto d] \) for some \( d \). By definition, \( b \in \text{fresh}_r \cup \text{freed}_r \). If \( a = b \), then we have \( b \in \text{fresh}_{r, \text{act}} \cup \text{freed}_{r, \text{act}} \). This means \( \sigma. \text{act} \in \mathcal{O}\[P\]^{(a)}_{\text{Adr}} \). Moreover, Lemma B.68 yields that \( \hat{\sigma} = \sigma. \text{act} \) satisfies the claim. So, assume \( a \neq b \) hereafter. We first establish that \( b \notin \text{adr}(m_r|_{\text{valid}_r}) \) holds. To that end, we invoke the induction hypothesis for \( b \). This gives \( \gamma \in \mathcal{O}\[P\]^{(b)}_{\text{Adr}} \) with \( \tau \sim \gamma, \tau \leq_b \gamma \), and \( \tau < \gamma \). We get \( b \in \text{fresh}_r \cup \text{freed}_r \) because of \( \tau \leq_b \gamma \). Then, Lemmas B.35, B.43 and B.48 yield \( b \notin \text{adr}(m_r|_{\text{valid}_r}) \). Hence, Lemma B.38 together with \( \tau \sim \sigma \) yields the desired \( b \notin \text{adr}(m_r|_{\text{valid}_r}) \).

Now, choose \( \hat{\sigma} = \sigma \) and conclude as in the previous case. To see \( m_r|_{\text{valid}_r} = m_{\text{act}, r}|_{\text{valid}_{r, \text{act}}} \), note that we have:

\[ \text{dom}(m_{r, \text{act}}|_{\text{valid}_{r, \text{act}}}) = \text{valid}_{r, \text{act}} \cup \text{DVar} \cup \{ \text{c.data} \mid c \in m_{r, \text{act}}(\text{valid}_{r, \text{act}}) \} \]

\[ = \text{valid}_r \cup \text{DVar} \cup \{ \text{c.data} \mid c \in m_r(\text{valid}_r) \} \]

\[ = \text{valid}_r \cup \text{DVar} \cup \{ \text{c.data} \mid c \in m_r(\text{valid}_r) \} = \text{dom}(m_r|_{\text{valid}_r}) \]

where the first and last equality is by definition, the second equality by \( \text{valid}_{r, \text{act}} = \text{valid}_r \), and the third equality by \( b. \text{next} \notin \text{valid}_r \) by \( b \notin \text{adr}(m_r|_{\text{valid}_r}) \) together with Lemma B.35. Similarly, \( b. \text{data} \notin \text{valid}_r \) because \( b \notin m_r(\text{valid}_r) \) by \( b \notin \text{adr}(m_r|_{\text{valid}_r}) \) together with Lemma B.35. Altogether, this gives \( m_r|_{\text{valid}_r} = m_{\text{act}, r}|_{\text{valid}_{r, \text{act}}} \).
\textbf{Case 7: }com \equiv \text{assume cond} \\
By definition, \(up = \emptyset\). If \(\sigma.act \in \mathcal{O}[P]^{(a)}_{\text{adr}}\), then \(\hat{\sigma} = \sigma.act\) satisfies the claim by Lemma B.68. 
So assume \(\sigma.act \notin \mathcal{O}[P]^{(a)}_{\text{adr}}\). Since we have \(m_\tau(u) = m_\sigma(u)\) for all data variables \(u \in DVar\) by \(\tau \sim \sigma\), we must have \(cond \equiv p \triangleq q\) with \(p, q \in PVar\) and \(\triangle \in \{=, \neq\}\).

Let \(m_\tau(p) = b\). By induction, there is \(\delta \in \mathcal{O}[P]^{(b)}_{\text{adr}}\) with \(\tau \sim \delta\), \(\tau \triangle \delta\), and \(\tau \triangle \delta\). By \(\tau \triangle \delta\), the truth value of \(cond\) is the same after \(\tau\) and \(\delta\). 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 \(\delta.act \in \mathcal{O}[P]^{(b)}_{\text{adr}}\).

From Lemma B.68 we get \(\tau.act \sim \delta.act\), \(\tau.act \triangle \delta.act\), and \(\tau.act \triangle \delta.act\).

Observe that we have \(\delta = \sigma\) by Lemma B.32. Then, the absence of harmful ABAs for \(\delta.act\) and \(\sigma\) yields a computation \(\gamma \in \mathcal{O}[P]^{(a)}_{\text{adr}}\) with \(\delta.act \sim \gamma\), \(\sigma \triangle \gamma\), and \(\delta.act \triangle \gamma\). We show that \(\hat{\sigma} = \gamma\) satisfies the claim. To do so, we show an auxiliary property first.

\[m_\tau(\text{valid}_{\tau.act}) = m_\tau(\text{valid}_\tau) \tag{25}\]

\textbf{Ad (25).} By definition, \(m_\tau = m_\tau_{\tau.act}\). It remains to show \(m_\tau(\text{valid}_{\tau.act}) = m_\tau(\text{valid}_\tau)\). In the case \(\text{valid}_{\tau.act} = \text{valid}_\tau\) holds, nothing needs to be shown. Assume \(\text{valid}_{\tau.act} \neq \text{valid}_\tau\).

This means \(cond \equiv p = q\) such that wlog. \(p \in \text{valid}_\tau\) and \(q \notin \text{valid}_\tau\). So, \(m_\tau(p) = m_\gamma(q)\) must hold. This means \(m_\tau(q) \in m_\gamma(\text{valid}_\tau)\). Further, \(\text{valid}_{\tau.act} = \text{valid}_\tau \cup \{q\}\). We conclude by: \(m_\tau(\text{valid}_{\tau.act}) = m_\gamma(\text{valid}_\tau) \cup \{m_\gamma(q)\} = m_\gamma(\text{valid}_\tau)\).

\textbf{Ad }\tau.act \sim \gamma.\text{ Follows from Lemma B.32 together with }\tau.act \sim \delta.act\text{ and }\delta.act \sim \gamma.\text{ }

\textbf{Ad }\tau.act \triangle \gamma.\text{ We have }\tau.act \triangle \delta.act \triangle \gamma\text{ and }\text{adr}(m_\tau(\text{valid}_{\tau.act})) = \text{adr}(m_\delta(\text{valid}_{\delta.act})).\text{ The latter follows from }\tau.act \sim \delta.act\text{ together with Lemma B.38. Then, Lemma B.34 gives }\tau.act \triangle \gamma.\text{ }

\textbf{Ad }\tau.act \triangle_a \gamma.\text{ We have }\tau \triangle_a \sigma \triangle_a \gamma.\text{ By }\tau \sim \sigma,\text{ we have }m_\tau(\text{valid}_\tau) \notin m_\sigma(\text{valid}_\sigma).\text{ Lemma B.33 yields }\tau \triangle_a \gamma.\text{ Assume for the moment we have }\tau.act \triangle_a \tau.\text{ Then, (25) together with Lemma B.33 yields the desired }\tau.act \triangle_a \gamma.\text{ It remains to show }\tau.act \triangle_a \tau.\text{ This follows from }\mathcal{H}(\tau.act) = \mathcal{H}(\tau), m_\tau = m_{\tau.act}, \text{freed}_{\tau.act} = \text{freed}_\tau, \text{fresh}_{\tau.act} = \text{fresh}_\tau, \text{ and }\text{retired}_{\tau.act} = \text{retired}_\tau\text{ together with (25).}

\textbf{Case 8: }com \in \{\text{skip, beginAtomic, endAtomic, @inv}\}.

We immediately obtain that \(\hat{\sigma} = \sigma.act\) satisfies the claim.

The above case distinction is complete and thus concludes the induction.

\textbf{Proof C.52} (Theorem 7.21). If \(\text{good}(\mathcal{O}[P]^{(a)}_{\text{adr}})\) then, \(\text{good}(\mathcal{O}[P]^{(a)}_{\text{one}})\) by \(\mathcal{O}[P]^{(a)}_{\text{adr}} \subseteq \mathcal{O}[P]^{(a)}_{\text{one}}\).

For the reverse direction, assume \(\text{good}(\mathcal{O}[P]^{(a)}_{\text{one}})\) holds. To the contrary, assume \(\text{good}(\mathcal{O}[P]^{(a)}_{\text{adr}})\) does not hold. So there is \(\tau \in \mathcal{O}[P]^{(a)}_{\text{adr}}\) such that \(\text{good}(\tau)\) is not satisfied. By definition, this means we have \(\text{ctrl}(\tau) \cap \text{Fault} \neq \emptyset\) where \(\text{Fault}\) are the bad control locations. Theorem 7.20
yields \( \sigma \in \mathcal{O}[P]_{\text{Adr}}^{\text{one}} \) with \( \tau \sim \sigma \). The latter gives \( \text{ctrl}(\sigma) \cap \text{Fault} \neq \emptyset \). Hence, \( \text{good}(\mathcal{O}[P]_{\text{Adr}}^{\text{one}}) \) does not hold. Since this contradicts the assumption, \( \text{good}(\mathcal{O}[P]_{\text{Adr}}) \) must hold. 

**Proof C.53** (Theorem 7.22). Towards a contradiction, assume that there is a shortest computation \( \tau.\text{act} \in \mathcal{O}[P]_{\text{Adr}}^{\text{one}} \) that performs a double retire. That is, we have \( \text{act} = \langle t, \text{retire}(p), a \rangle \) with \( m_{t}(p) = a \in \text{retired}_{\tau} \). Theorem 7.20 yields \( \sigma \in \mathcal{O}[P]_{\text{Adr}}^{\{a\}} \) with \( \tau \sim \sigma \) and \( \tau \equiv_{\sigma} \sigma \). The latter gives \( m_{\sigma}(p) = a \) and \( a \in \text{retired}_{\tau} \). By Lemma B.68 and \( \tau \sim \sigma \), we have \( \sigma.\text{act} \in \mathcal{O}[P]_{\text{Adr}}^{\{a\}} \). That is, \( \mathcal{O}[P]_{\text{Adr}}^{\text{one}} \) is not free from double retires. Since this contradicts the assumption, \( \mathcal{O}[P]_{\text{Adr}} \) must be free from double retires. 

**Proof C.54** (Proposition 7.15). In the following, we rely on the following properties of \( \mathcal{O}_{\text{SMR}} \): (i) there is at most one variable \( z \) via transitions of the form \( a \), (ii) accepting locations \( l \) are reached only via transitions labeled with free \( r \), and (iii) all transitions of the form \( l \xrightarrow{\text{free}(r)} l' \) satisfy: \( l = l' \) or \( l' \) is accepting. These properties are satisfied by \( \mathcal{O}_{\text{ERR}}, \mathcal{O}_{\text{HP}}^{0} \times \mathcal{O}_{\text{HP}}, \) and \( \mathcal{O}_{\text{HP}}^{0,1} \). Let location \( l_{\text{init}} \) uniformly refer to the initial location in the aforementioned SMR automata.

\( \diamond \) **Ad Definition 7.14i.** Let \( a \neq c \neq b \). We show \( F_{\mathcal{O}_{\text{SMR}}}(h, c) = F_{\mathcal{O}_{\text{SMR}}}(h[a/b], c) \). By Lemma B.64 we have \( F_{\mathcal{O}_{\text{SMR}}}(h, c)[a/b] = F_{\mathcal{O}_{\text{SMR}}}(h[a/b], c) \). So, it is sufficient to show that \( F_{\mathcal{O}_{\text{SMR}}}(h, c)[a/b] = F_{\mathcal{O}_{\text{SMR}}}(h, c) \). By definition, this means \( h.h' \in S(\mathcal{O}_{\text{SMR}}) \) and \( \text{free}_{h'} \subseteq \{ c \} \). Because \( \mathcal{O}_{\text{SMR}} \) never leaves accepting locations, we have \( h \in S(\mathcal{O}_{\text{SMR}}) \). To the contrary, assume \( h.h' \notin F_{\mathcal{O}_{\text{SMR}}}(h, c)[a/b] \). This means \( h.h'[a/b] \notin F_{\mathcal{O}_{\text{SMR}}}(h, c) \). Consequently, we have \( h.h'[a/b] \notin S(\mathcal{O}_{\text{SMR}}) \). By definition, there are steps \( (l_{\text{init}}, \varphi) \xrightarrow{h} (l_{1}, \varphi) \xrightarrow{\text{free}(a), r = z_{a}} (l_{2}, \varphi) \) with \( l_{2} \) accepting. By \( h \in S(\mathcal{O}_{\text{SMR}}) \), we know that \( l_{1} \) is not accepting. By \( \text{free}_{h}[a/b] = \text{free}_{h'} \subseteq \{ c \} \) and \( \mathcal{O}_{\text{SMR}} \) 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 \( \mathcal{O}_{\text{SMR}} \) having only one variable \( z_{a} \) tracking addresses, we know that \( \mathcal{O}_{\text{SMR}} \) cannot distinguish \( a \) and \( b \) so that we obtain \( (l_{1}, \varphi) \xrightarrow{h} (l_{2}, \varphi) \). Since this contradicts \( h.h' \in S(\mathcal{O}_{\text{SMR}}) \), we must have the required \( h.h' \notin F_{\mathcal{O}_{\text{SMR}}}(h, c)[a/b] \). The reverse inclusion follows analogously.

\( \diamond \) **Ad Definition 7.14ii.** Let \( a \neq b \) and \( h.\text{free}(a) \in S(\mathcal{O}) \). Consider some \( \varphi \) and some program step \( (l_{\text{init}}, \varphi) \xrightarrow{h} (l_{1}, \varphi) \xrightarrow{\text{free}(a)} (l_{2}, \varphi) \). Because \( h.\text{free}(a) \in S(\mathcal{O}) \), we know \( l_{1} \) is not accepting. So, the properties of \( \mathcal{O}_{\text{SMR}} \) yield \( l_{2} = l_{1} \). Hence, \( F_{\mathcal{O}_{\text{SMR}}}(h, b) = F_{\mathcal{O}_{\text{SMR}}}(h.\text{free}(a), b) \) as required.

\( \diamond \) **Ad Definition 7.14iii.** Assume \( F_{\mathcal{O}}(h, a) \subseteq F_{\mathcal{O}}(h, a) \) and \( b \in \text{fresh}_{h'} \). We establish the following: \( F_{\mathcal{O}_{\text{SMR}}}(h, b) \subseteq F_{\mathcal{O}_{\text{SMR}}}(h', b) \). As before, it suffices to consider \( \varphi \) with \( \varphi(z_{a}) = b \).

For \( \mathcal{O}_{\text{HP}}^{0} \) and \( \mathcal{O}_{\text{HP}}^{0,1} \), note that \( b \in \text{fresh}_{h} \) immediately gives \( (l_{\text{init}}, \varphi) \xrightarrow{h} (l_{\text{init}}, \varphi) \) as \( l_{\text{init}} \) can only be left with events where the parameter is \( b \). Moreover, \( l_{\text{init}} \) 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 $O_{EBR}$. From $b \in \text{fresh}_{h\varphi}$ 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 $h \notin S(O_{EBR})$, then nothing needs to be shown as the desired inclusion is trivially true. So, assume $h \in S(O_{EBR})$. 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 $h \in S(O_{EBR})$, we get $h.h_1 \in S(O_{EBR})$. Moreover, $h.h_1 \in S(O)$ because $O_{Base}$ does not react on $h_1$ and because $h \in S(O_{Base})$. Observe that we have the step $(l_{init}, \varphi') \xrightarrow{h.h_2} (l_{init}, \varphi')$ for all valuations $\varphi'$.
- By $h.h_1 \in S(O_{Base})$ and the definition of $O_{Base}$, there must be $h_2 \in \{ e, \text{free}(a) \}$ such that $h.h_1.h_2 \in S(O_{Base})$. Since $O_{EBR}$ reaches $l_{init}$ after $h.h_1$ for all $\varphi'$ as noted above, we have $h.h_1.h_2 \in S(O)$. 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 $h.h_1.h_2.h_3 \in S(O)$.

By $h.h_1.h_2.h_3 \in S(O)$ together with $\text{fresh}_{h.h_1.h_2.h_3} \subseteq \{ a \}$, we have $h_1.h_2.h_3 \in F_O(h, a)$. Hence, we get $h_1.h_2.h_3 \in F_O(h', a)$ by the premise. This means $h_1.h_2.h_3 \in S(O)$. However, there are the following steps for $\varphi' = \{ z_t \mapsto \varphi(z_t), z_a \mapsto a \}$ on $h_1.h_2.h_3$:

$$(l_{init}, \varphi') \xrightarrow{h.h_2} (l_1, \varphi') \xrightarrow{h} (l_2, \varphi') \xrightarrow{h_2} (l_3, \varphi') \xrightarrow{h_3} (l_4, \varphi')$$

where (i) $l_1 \neq l_{init}$ 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 $O_{EBR}$ ignores $\text{in: enterQ}(\bullet)$ events other that $\varphi'(z_t)$, (iii) $l_2 = l_3$, and (iv) $l_3 = L_7$. Altogether, this means $h'.h_1.h_2.h_3 \notin S(O)$ because $L_7$ is accepting. Since this contradicts the assumption, we must have $l' \neq L_4$ as required.

\[\blacksquare\]

**Proof C.55 (Proposition 7.23).** The claim follows for $O_{Base} \times O_{EBR}$ since $\text{enterQ}$ and $\text{leaveQ}$ do not take parameters. We turn to $O = O_{Base} \times O^0_{HP} \times O^0_{HP}$. Again, $\text{unprotect}_k$ does not take parameters and thus never races. Consider some computation $r.\text{act}$ with $\text{act} = \langle t, \text{in: protect}_k(p), up \rangle$ and $m_t(p) = c$ and $H(t) = h$. Assume $\text{act}$ is a racy call. Then, there are $a, b, c \in Adr$ with $a \neq c$ such that $F_O(h.\text{in: protect}_k(b), a) \notin F_O(h.\text{in: protect}_k(c), a)$. This means there is $h' \in F_O(h.\text{in: protect}_k(b), a)$ with $h' \notin F_O(h.\text{in: protect}_k(c), a)$. We have $h \in S(O)$ because of the membership of $h'$. There are steps $(l_{init}, \varphi) \xrightarrow{h} (l, \varphi)$ with $l$ accepting. By definition, we have $\text{fresh}_{h} \subseteq \{ a \}$. Hence, for $h'$ to reach an accepting location, we must have $\varphi(z_a) = a$ due to the definition of $O$. Consequently, $\text{in: protect}_k(b)$ and $\text{in: protect}_k(c)$ are indistinguishable for $O$. 

\[\blacksquare\]
Under $\varphi$, all transitions taken by the former event can be taken by the latter event as well, and vice versa. Hence, $\text{protect}_k$ does not race. The argument is analogous in $O_{\text{Base}} \times O_{\text{HP}}$.

**Proof C.56** (Theorem B.71). Let $O$ support elision and let $O[[P]]_{Adr}$ be MPRF and DRF. Note that this means $O[[P]]_{Adr}$ is PRF. We proceed by induction over the structure of $\tau$. In the base case, we have $\tau = \epsilon$. Choosing $\sigma = \epsilon$ satisfies the claim. For the induction step, consider some $\tau.\text{act} \in O[[P]]_{Adr}$ and assume that we have already constructed $\sigma \in O[[P]]_{Adr}$ and established the following properties:

1. $\tau \sim \sigma$
2. $\tau < \sigma$
3. $\forall a \in \text{fresh}_\sigma: \mathcal{F}_O(\tau, a) \subseteq \mathcal{F}_O(\sigma, a)$
4. $\forall pexp, qexp \in \mathcal{VExp}(\tau). \ m_\tau(pexp) \neq m_\tau(qexp) \implies m_\sigma(pexp) \neq m_\sigma(qexp)$
5. $\text{retired}_\tau \subseteq \text{retired}_\sigma$
6. $\text{freed}_\tau \cap \text{adr}(m_\tau|_{\text{valid}_\tau}) = \emptyset$
7. $\text{freed}_\tau \cap \text{retired}_\tau = \emptyset$
8. $\tau$ is UAF

We construct $\hat{\sigma}$ and show the following:

1. $\hat{\sigma} \in O[[P]]_{Adr}$
2. $\tau.\text{act} \sim \hat{\sigma}$
3. $\tau.\text{act} < \hat{\sigma}$
4. $\forall a \in \text{fresh}_{\hat{\sigma}}: \mathcal{F}_O(\tau.\text{act}, a) \subseteq \mathcal{F}_O(\hat{\sigma}, a)$
5. $\forall pexp, qexp \in \mathcal{VExp}(\tau.\text{act}). \begin{pmatrix} m_{\tau.\text{act}}(pexp) \neq m_{\tau.\text{act}}(qexp) \\ \implies m_{\hat{\sigma}}(pexp) \neq m_{\hat{\sigma}}(qexp) \end{pmatrix}$
6. $\text{retired}_{\tau.\text{act}} \subseteq \text{retired}_{\hat{\sigma}}$
7. $\text{freed}_{\tau.\text{act}} \cap \text{adr}(m_{\tau.\text{act}}|_{\text{valid}_{\tau.\text{act}}}) = \emptyset$
8. $\text{freed}_{\tau.\text{act}} \cap \text{retired}_{\tau.\text{act}} = \emptyset$
9. $\tau.\text{act}$ is UAF

Let $act = \langle t, \text{com}, \text{up} \rangle$. We do a case distinction over $\text{com}$.

**Case 1:** $\text{com}$ 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 $\text{com}$ contains $p.\text{sel}$ then $p \in \text{valid}_\tau$. So, assume $\text{com}$ contains $p.\text{sel}$ as nothing needs to be shown otherwise. We proceed as follows: we show that $\text{com}$ is enabled after $\sigma$ 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 $\tau$. This
Ad (G3). Let $a \in \text{fresh}_{\sigma, \text{act}}$. By definition, $a \in \text{fresh}_v$. We obtain $\mathcal{F}_O(t, a) \subseteq \mathcal{F}_O(\sigma, a)$ from (P3). Then, the desired $\mathcal{F}_O(t, \text{act}, a) \subseteq \mathcal{F}_O(\sigma, \text{act}, a)$ follows from $\text{act}/\text{act}'$ not emitting an event, thus $\mathcal{F}_O(t, \text{act}, a) = \mathcal{F}_O(t, a)$ and $\mathcal{F}_O(\sigma, \text{act}, a) = \mathcal{F}_O(\sigma, a)$ by definition.

Ad (G4). Consider $\text{pexp}, \text{qexp} \in VExp(t, \text{act})$ with $m_{\tau, \text{act}}(\text{pexp}) \neq m_{\tau, \text{act}}(\text{qexp})$. We focus on the case $\text{com} = p := q, \text{next}$; 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, $up = [p \mapsto b]$. Because $\text{act}'$ is PRF, we have $q \in \text{valid}_v$. Hence, $m_{\tau}(a) = a$ by (P1). Let $c = m_{\tau}(a, \text{next})$. Then, we have $up' = [p \mapsto c]$. If $\text{pexp} \neq p \neq \text{qexp}$, then we obtain $m_{\tau, \text{act}}(\text{pexp}) = m_{\tau}(\text{pexp})$ as well $m_{\tau, \text{act}}(\text{qexp}) = m_{\tau}(\text{qexp})$ as well as $\text{pexp} \in VExp(t)$, and similarly for $\text{qexp}$, so that we conclude by (P4). So assume now wlog. $\text{pexp} \equiv p$. This means, $\text{qexp} \neq q$. We arrive at $m_{\tau}(a, \text{next}) = m_{\tau, \text{act}}(p) \neq m_{\tau, \text{act}}(\text{qexp}) = m_{\tau}(\text{qexp})$. Furthermore, $q \in VExp(t)$ holds by definition and $\text{qexp} \in VExp(t)$ 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 $m_{\tau}(a, \text{next}) \neq m_{\tau}(\text{qexp})$. From the update $up'$ we get $m_{\tau, \text{act}}(p) = m_{\tau}(a, \text{next})$ and $m_{\tau}(\text{qexp}) = m_{\tau, \text{act}}(\text{qexp})$. This concludes the desired inequality $m_{\tau, \text{act}}(p) \neq m_{\sigma, \text{act}}(\text{qexp})$.

Ad (G5). We conclude by (P5), retired$_{\tau} = \text{retired}_{\tau, \text{act}}$, and 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 freed$_{\tau, \text{act}} \cap \text{adr}(m_{\tau, \text{act}}|\text{valid}_{\tau, \text{act}}) = \emptyset$ by (P6) together with freed$_{\tau} = \text{freed}_{\tau, \text{act}}$.

Ad (G7). We conclude by (P7) as well as freed$_{\tau, \text{act}} = \text{freed}_{\tau}$ and retired$_{\tau, \text{act}} = \text{retired}_{\tau}$.

Ad (G8). We have already shown that if $\text{com}$ 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: $\text{com} \equiv \text{in:func}(r_1, \ldots, r_n)$

The update is $up = \emptyset$. We show that $\hat{\tau} = \tau, \text{act}$ is an adequate choice.

Ad (G0). We show that $\text{act}$ is enabled after $\tau$. By Lemma B.50, $m_{\tau}(r_i) \in \text{Adr} \cup \{\text{seg}\}$. To the contrary, assume $m_{\tau}(r_i) = \text{seg}$ for some $i$. Then, $r_i \in \text{valid}_v$ by Lemma B.49. So, we get $m_{\tau}(r_i) = \text{seg}$ by $\tau \sim \tau$. As this contradicts enabledness of $\text{act}$ after $\tau$, $m_{\tau}(r_i) \neq \text{seg}$ must hold. Altogether, this means $\tau, \text{act} \in \mathcal{O}[P]_{\text{Adr}}^\emptyset$. 

Proof of Meta Theory
Ad (G2) and (G3). Let the parameters to the call be \( \bar{r} = r_1, \ldots, r_n \). Moreover, let the actual arguments to the call be \( m_r(\bar{r}) = \bar{v}_r = v_{r,1}, \ldots, v_{r,n} \) and \( m_\sigma(\bar{r}) = \bar{v}_\sigma = v_{\sigma,1}, \ldots, v_{\sigma,n} \). We show:

\[
\forall b \in \text{adr}(m_r.\text{act}|\text{valid}_r) \cup \text{fresh}_\sigma.\text{act} \cdot F_O(\tau.\text{act}, b) \subseteq F_O(\sigma.\text{act}, b).
\]

Consider some \( b \in \text{adr}(m_r.\text{act}|\text{valid}_r) \cup \text{fresh}_\sigma.\text{act} \). We obtain \( b \in \text{adr}(m_r|\text{valid}_r) \cup \text{fresh}_\sigma \) as before. Now, (P2) and (P3) give \( F_O(\tau, b) \subseteq F_O(\sigma, b) \). Then, Lemma B.41 yields:

\[
F_O(\mathcal{H}(\tau).\text{in}\cdot\text{func}(t, \bar{v}_\tau), b) \subseteq F_O(\mathcal{H}(\sigma).\text{in}\cdot\text{func}(t, \bar{v}_\sigma), b). \tag{26}
\]

Next, we use the fact that \( \sigma.\text{act} \) is PRF by assumption and (G0). That is, \( \text{act} \) is not racy:

\[
F_O(\mathcal{H}(\sigma).\text{in}\cdot\text{func}(t, \bar{v}_\sigma), b) \subseteq F_O(\mathcal{H}(\sigma).\text{in}\cdot\text{func}(t, \bar{v}_\sigma), b). \tag{27}
\]

To see this, we have to show that the following holds for all \( i \) with \( 1 \leq i \leq n \):

\[
(v_{\sigma,i} = b \lor r_i \in \text{valid}_\sigma \lor r_i \in \text{DExp}) \implies v_{r,i} = v_{\sigma,i}.
\]

If \( r_i \in \text{valid}_\sigma \), then \( v_{\sigma,i} = m_\sigma(r_i) = m_r(r_i) = v_{r,i} \) because of \( \tau \sim \sigma \). Similarly, \( v_{\sigma,i} = v_{r,i} \) 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: \( b \in \text{adr}(m_r|\text{valid}_r) \) or \( b \in \text{fresh}_\sigma \). In the former case, we have \( b \in \text{adr}(m_\sigma|\text{valid}_\sigma) \) by (P1). Moreover, \( b \in m_\sigma(P\text{Var} \setminus \text{valid}_\sigma) \) follows from \( r_i \notin \text{valid}_\sigma \) and \( m_\sigma(r_i) = v_{\sigma,i} = b \). Then, Lemma B.53 yields \( b \notin \emptyset \). 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:

\[
F_O(\tau.\text{act}, b) = F_O(\mathcal{H}(\tau).\text{in}\cdot\text{func}(t, \bar{v}_\tau), b) \subseteq F_O(\mathcal{H}(\sigma).\text{in}\cdot\text{func}(t, \bar{v}_\tau), b) \\
\subseteq F_O(\mathcal{H}(\sigma).\text{in}\cdot\text{func}(t, \bar{v}_\tau), b) = F_O(\sigma.\text{act}, b).
\]

Ad (G1). We have \( m_r = m_r.\text{act} \) and \( \text{valid}_r = \text{valid}_r.\text{act} \) by definition, and similarly for \( \sigma \). Hence, the desired \( \tau.\text{act} \equiv_\sigma \sigma.\text{act} \) follows from (P1).

Ad (G4). Follows by (P4) and \( m_r = m_r.\text{act}, m_\sigma = m_\sigma.\text{act}, \) and \( \text{VExp}(\tau) = \text{VExp}(\tau.\text{act}) \).

Ad (G5). If \( \text{com} \notin \text{in}\cdot\text{retire}(p) \), we conclude by (P5) as well as \( \text{retired}_r = \text{retired}_r.\text{act} \) and \( \text{retired}_\sigma = \text{retired}_\sigma.\text{act} \). So consider \( \text{com} \equiv \text{in}\cdot\text{retire}(p) \). Towards a contradiction, assume \( p \notin \text{valid}_\sigma \). By Lemma B.51, this means \( m_\sigma(p) \in \text{frees}_\sigma \subseteq \text{freed}_\sigma \) where the inclusion holds because no addresses are reallocated in \( \sigma \). Then, \( m_\sigma.\text{act}(p) \in \text{freed}_\sigma.\text{act} \) and \( m_\sigma.\text{act}(p) \in \text{retired}_\sigma.\text{act} \). Lemma B.70 now states that \( O[P]_\text{Adr} \) contains a double
retire. Since this contradicts the assumption, we must have $p \in \text{valid}_\sigma$. Hence, obtain that $m_r(p) = m_\sigma(p) = a$ for some $a$. We conclude by (P5):

\[ \text{retired}_{r, \text{act}} = \text{retired}_r \cup \{a\} \subseteq \text{retired}_\sigma \cup \{a\} = \text{retired}_{\sigma, \text{act}}. \]

\(\diamond\) Ad (G6). We have $\text{adr}(m_r|_{\text{valid}_r}) = \text{adr}(m_{r, \text{act}}|_{\text{valid}_{\text{act}}})$ since $\text{act}$ does not affect the memory nor the validity, as noted before. Combined with $\text{freed}_r = \text{freed}_{r, \text{act}}$ and (P6) implies (G6).

\(\diamond\) Ad (G7). If $\text{com} \not\equiv \text{retire}(p)$, nothing needs to be shown and we conclude by (P7). So consider now the case $\text{com} \equiv \text{retire}(p)$. We first show $p \in \text{valid}_\sigma$. To the contrary, assume $p \notin \text{valid}_\sigma$. Let $m_r(p) = a$. From Lemma B.52 we obtain $a \in \text{freed}_\sigma$ and thus get $a \in \text{freed}_{\sigma, \text{act}}$ by definition. Moreover, we obtain $a \in \text{retired}_{\sigma, \text{act}}$. Then, Lemma B.70, which is enabled by (G0), states that $\mathcal{O}[P]_{\text{adr}}$ contains a double retire. This contradicts the assumption. Hence, $p \in \text{valid}_\sigma$ must hold. By (G1), this means $p \in \text{valid}_r$ as well as $m_r(p) = a$. That is, $a \in m_r(\text{valid}_r) \subseteq \text{adr}(m_r|_{\text{valid}_r})$ where the inclusion follows from Lemma B.35. Now, (P6) yields $a \notin \text{freed}_r$. Altogether, this gives the desired $a \notin \text{freed}_{r, \text{act}}$ by definition.

\(\diamond\) Ad (G8). Follows from (P8) together with $\text{act}$ not raising unsafe accesses.

\(\diamond\) Case 3: $\text{com} \equiv \text{re}:\text{func}$

We choose $\hat{\sigma} = \sigma.\text{act}$. By $\tau \sim \sigma$, we know that $\text{act}$ is enabled after $\sigma$, that is, $\sigma.\text{act} \in \mathcal{O}[P]_{\text{adr}}$. By definition, $\text{act}$ emits the same event $\text{re}:\text{func}(t)$ after both $\tau$ and $\sigma$. Then, the claim follows similarly to the previous case.

\(\diamond\) Case 4: $\text{com} \equiv p := \text{malloc}$ and $m_{r, \text{act}}(p) \in \text{fresh}_\sigma$

Let $a = m_{r, \text{act}}(p)$. The update is $up = [p \mapsto a, a.\text{next} \mapsto \text{seg}, a.\text{data} \mapsto d]$ for some $d$. By definition, we have $a \in \text{fresh}_r \cup \text{freed}_r$. By assumption, we have $a \in \text{fresh}_r$. We now show that $\hat{\sigma} = \sigma.\text{act}$ is an adequate choice.

\(\diamond\) Ad (G0) to (G2). From $a \in \text{fresh}_\sigma$ we immediately get $\sigma.\text{act} \in \mathcal{O}[P]_{\text{adr}}$. Then, (G0) to (G2) follow from (P1) and (P2) together with Lemma B.68. Note that Lemma B.68 is applicable because we get $\mathcal{F}_\sigma(\tau, a) \subseteq \mathcal{F}_\sigma(\sigma, a)$ from $a \in \text{fresh}_\sigma$, together with (P3).

\(\diamond\) Ad (G3). Follows from $\text{fresh}_{\sigma, \text{act}} \subseteq \text{fresh}_\sigma$, together with the fact that $\text{act}$ does not emit an event, i.e., $\mathcal{H}(\tau.\text{act}) = \mathcal{H}(\tau)$ and $\mathcal{H}(\sigma.\text{act}) = \mathcal{H}(\sigma)$.

\(\diamond\) Ad (G4). Consider $p\text{exp}, q\text{exp} \in \text{VExp}(\tau.\text{act})$ with $m_{r, \text{act}}(p\text{exp}) \neq m_{r, \text{act}}(q\text{exp})$. By definition, we have $\text{VExp}(\tau.\text{act}) = \text{VExp}(\tau) \cup \{a.\text{next}\}$. 

Appendix C Proof of Meta Theory
Case 5: \[ com \equiv p := \text{malloc} \text{ and } m_{r,act}(p) \notin \text{fresh}_\sigma \]

Let \( a = m_{r,act}(p) \). The update is \( up = [p \mapsto a, a.next \mapsto \text{seg}, a.data \mapsto d] \) for some datum \( d \).

By definition, we have \( a \in \text{fresh}_\sigma \cup \text{freed}_r \). If \( a \notin \text{freed}_r \), then \( a \notin \text{adr}(m_{r,valid}_r) \) by induction. Otherwise, \( a \in \text{fresh}_\sigma \) and we get \( a
\neq \text{adr}(m_{r,valid}_r) \) by Lemmas B.35 and B.43. Then, \( \tau \sim \sigma \) together with Lemma B.38 yields \( a \neq \text{adr}(m_{\sigma,valid}_\sigma) \). Now, we invoke Lemma B.66 for \( \sigma \) and \( a \). This gives \( \gamma \in \mathcal{O}_{\text{adr}}[P]^{\gamma}_{\text{adr}} \) with \( \tau \sim \gamma, \sigma < \gamma, a \in \text{fresh}_\gamma \), and \( \text{freed}_\sigma \subseteq \text{retired}_\tau \cup \{a\} \).

Furthermore, the lemma results in \( m_\sigma(pexp) \neq m_\sigma(pexp) \Rightarrow m_\gamma(pexp) \neq m_\gamma(pexp) \), for all \( pexp, qexp \in \text{VExp}(\sigma) \). Since \( a \notin \text{fresh}_\sigma \), by assumption, we get \( \mathcal{F}_\sigma(\sigma, b) = \mathcal{F}_\gamma(\gamma, b) \) for all \( b \in \text{fresh}_\sigma \setminus \{a\} \).

CASE 4.1: \( pexp, pexp \notin \{p, a.next\} \)

We get \( m_{r,act}(pexp) = m_\tau(pexp) \) 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, \( qexp \neq p \). Because \( a \in \text{fresh}_\sigma \), Lemma B.42 yields \( m_{\sigma,act}(pexp) \neq m_\sigma(qexp) \). Furthermore, \( m_{\sigma,act}(pexp) \neq \text{seg} \). Hence, we arrive at \( m_{\sigma,act}(pexp) \neq m_{\sigma,act}(qexp) \) because we have \( m_{\sigma,act}(qexp) \in \{\text{seg}, m_\sigma(qexp)\} \) 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 \( pexp \neq p \)

We have \( m_{r,act}(pexp) = m_{\sigma,act}(pexp) = m_{\sigma,act}(qexp) = \text{seg} \). Towards a contradiction, assume that \( m_{\sigma,act}(qexp) = \text{seg} \) holds. By \( qexp \in VExp(\tau, act) \), we have \( qexp \in PVar \) or \( qexp \equiv b.next \wedge b \in m_{\sigma,act}(\text{valid}_\tau, act) \). By (P1) this means that we have either \( qexp \in PVar \) or \( qexp \equiv b.next \wedge b \in m_{\sigma,act}(\text{valid}_\sigma, act) \). From Lemma B.49 we get \( qexp \in \text{valid}_\sigma, act \). So, \( m_{\sigma,act}(qexp) = \text{seg} \) by (P1). Since this contradicts the choice of \( m_{r,act}(pexp) \neq m_{\tau,act}(qexp) \), we must have \( m_{\sigma,act}(qexp) \neq \text{seg} \) as required.

Ad (G5). We conclude by (P5), \( \text{retired}_r = \text{retired}_{\tau,act} \) and \( \text{retired}_\sigma = \text{retired}_{\sigma,act} \).

Ad (G6). By definition, we have \( \text{valid}_{\tau,act} = \text{valid}_r \cup \{p, a.next\} \). Consequently, we get \( \text{valid}_{\tau,act} \cap \text{Adr} = (\text{valid}_r \cap \text{Adr}) \cup \{a\} \). From an invocation of Lemma B.47 we then get \( \text{adr}(m_{r,act}|_{\text{valid}_{\tau,act}}) \subseteq \text{adr}(m_{r,valid}_r) \cup \{a\} \). Combined with \( \text{freed}_{\tau,act} = \text{freed}_r \setminus \{a\} \), we conclude by (P6).

Ad (G7). We have \( \text{freed}_{\tau,act} \subseteq \text{freed}_r \) and \( \text{retired}_{\tau,act} = \text{retired}_r \). We conclude by (P7).

Ad (G8). Follows from (P8) together with \( \text{act} \) not raising unsafe accesses.
diamond Ad $\tau \sim \gamma$ and $\tau \in \gamma$. From the above, we have $\tau \sim \sigma \sim \gamma$ and $\tau < \sigma < \gamma$. By definition and Lemma B.38, we get $m_\tau(\text{valid}_\tau) \subseteq m_\sigma(\text{valid}_\sigma)$ and $\text{adr}(m_\tau|_{\text{valid}_\tau}) = \text{adr}(m_\sigma|_{\text{valid}_\sigma})$.

 Lemmas B.32 to B.34 yield $\tau \sim \gamma$ and $\tau < \gamma$.

diamond Ad $\mathcal{F}_\sigma(\tau, b) \subseteq \mathcal{F}_\sigma(\gamma, b)$ for all $b \in \text{fresh}_\gamma$. Let $b \in \text{fresh}_\gamma$. If $b \neq a$ holds, we immediately get $\mathcal{F}_\sigma(\tau, b) \subseteq \mathcal{F}_\sigma(\sigma, b) = \mathcal{F}_\sigma(\gamma, b)$ where the inclusion is due to (P3) and the equality due to the correlation between $\sigma$ and $\gamma$ from above. Otherwise, $b = a$. Then, the desired inclusion follows from Lemma B.57 for $\tau$ and $\gamma$. We need to show that Lemma B.57 is enabled. We already have $a \in \text{fresh}_\gamma$. Moreover, we have $\mathcal{F}_\sigma(\tau, c) \subseteq \mathcal{F}_\sigma(\sigma, c) = \mathcal{F}_\sigma(\gamma, c)$ for any $c \in \text{fresh}_\sigma \cap \text{fresh}_\gamma$ due to (P3) and the correlation of $\sigma$ and $\gamma$. 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, $\mathcal{F}_\sigma(\tau, b) \subseteq \mathcal{F}_\sigma(\gamma, b)$.

diamond Ad $m_\tau(\text{pexp}) \neq m_\tau(\text{qexp}) \implies m_\tau(\text{pexp}) \neq m_\tau(\text{qexp})$ 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 $\gamma$ give $m_\gamma(\text{pexp}) \neq m_\gamma(\text{qexp})$.

diamond Ad $\text{retired}_\tau \subseteq \text{retired}_\sigma$. 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 $\gamma$. 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 $\gamma$ instead of $\sigma$ because $\gamma$ satisfies the necessary properties (P1) to (P8). Further, $a \in \text{fresh}_\sigma$. Hence, we conclude as in the previous case.

diamond Case 6: $\text{com} = \text{free}(a)$ and $\mathcal{H}(\sigma).\text{free}(a) \in S(\mathcal{O})$

The update is $\text{up} = \emptyset$. By definition, $\mathcal{H}(\tau).\text{free}(a) \in S(\mathcal{O})$. That is, $\text{free}(a) \in \mathcal{F}_\sigma(\tau, a)$. By assumption, $\mathcal{H}(\sigma).\text{free}(a) \in S(\mathcal{O})$. That is, $\text{free}(a) \in \mathcal{F}_\sigma(\sigma, a)$. We choose $\hat{\sigma} = \sigma.\text{act}$.

diamond Ad (G0) to (G2). By $\mathcal{H}(\sigma).\text{free}(a) \in S(\mathcal{O})$, we have $\sigma.\text{act} \in \mathcal{O}[\mathcal{P}]^\sigma_{\text{adr}}$. Then, we conclude by Lemma B.68, which is enabled according to Lemma B.56.

diamond 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 $\mathcal{F}_\sigma(\tau.\text{act}, b) = \mathcal{F}_\sigma(\tau, b)$ and $\mathcal{F}_\sigma(\sigma.\text{act}, b) = \mathcal{F}_\sigma(\sigma, b)$. We conclude by (P3).

diamond 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).

diamond Ad (G5). By (P5) we get $\text{retired}_{\tau.\text{act}} = \text{retired}_\sigma \setminus \{a\} = \text{retired}_\sigma \setminus \{a\} = \text{retired}_\sigma$.  

230 Appendix C Proof of Meta Theory
Case 7: \( \text{com} \equiv \text{free}(a) \) and \( H(\sigma).\text{free}(a) \notin S(O) \).

The update is \( \text{up} = \emptyset \). By definition, \( H(\tau).\text{free}(a) \in S(O) \). That is, \( \text{free}(a) \in F_O(\tau, a) \).

By assumption, we have \( \text{free}(a) \notin F_O(\sigma, a) \). By \( \tau < \sigma \) from (G2) we get \( a \notin \text{adr}(m_{r|\text{val}_r}) \) as otherwise we had \( \text{free}(a) \in F_O(\sigma, a) \). By (G1), this means \( a \notin m_{\sigma}(\text{val}_{\sigma}) \). We now show that \( \hat{\sigma} = \sigma \) is an adequate choice.

Ad (G0). By induction hypothesis, \( \sigma \in O[P]\_{\text{Adr}}^\sigma \).

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 \( m_{\tau.\text{act}|\text{val}_{\tau.\text{act}}} = m_{\sigma|\text{val}_\sigma} \) it suffices to show \( m_{r|\text{val}_r} = m_{\tau.\text{act}|\text{val}_{\tau.\text{act}}} \) by (P1).

To arrive there, we first show \( \text{val}_{\tau.\text{act}} = \text{valid}_{\tau.\text{act}} \). 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 \( \text{pexp} \in \text{valid}_{\tau} \). Then, \( \text{pexp} \neq a.\text{next} \).

Moreover, we must have \( m_{\tau}(\text{pexp}) \neq a \) as otherwise we would again get \( a \notin \text{adr}(m_{r|\text{val}_r}) \) by Lemma B.35 which does not hold as shown before. Moreover, we must have \( m_{\tau}(\text{pexp}) \neq a \) as otherwise we would again get \( a \notin \text{adr}(m_{r|\text{val}_r}) \) 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 \( m_{r|\text{val}_r} = m_{\tau.\text{act}|\text{val}_{\tau.\text{act}}} \) follows because \( m_{r} = m_{\tau.\text{act}} \) due to \( \text{up} = \emptyset \).

Ad (G2). Let \( b \in \text{adr}(m_{\tau.\text{act}|\text{val}_{\tau.\text{act}}} \). We get \( b \in \text{adr}(m_{r|\text{val}_r}) \) along the same lines as in (G1). This means that \( a \neq b \). Moreover, by \( \tau < \sigma \) we have \( F_O(\tau, b) \subseteq F_O(\sigma, b) \). So it suffices to show \( F_O(\tau, b) = F_O(\tau.\text{act}, b) \). This follows from Lemma B.56.

Ad (G3). Let \( b \in \text{fresh}_\sigma \). If \( a \neq b \), we get \( F_O(\tau.\text{act}, b) \subseteq F_O(\tau, b) \) from Lemma B.56 and then conclude by (P3). Now, consider \( a = b \). As argued above, \( \text{ffree}(a) \in F_O(\tau, a) \). Moreover, (P3) gives \( F_O(\tau, b) \subseteq F_O(\sigma, b) \). That is, we have \( \text{ffree}(a) \in F_O(\sigma, a) \). Since this contradicts \( \text{ffree}(a) \notin F_O(\sigma, a) \) from above, the case cannot apply.

Ad (G4). Let \( \text{pexp}, \text{qexp} \in VExp(\tau.\text{act}) \). By definition, \( \text{valid}_{\tau.\text{act}} \subseteq \text{valid}_{\tau} \) and \( m_{\tau.\text{act}} = m_{\tau} \). Hence, we have \( \text{pexp}, \text{qexp} \in VExp(\tau) \). Moreover, \( m_{\sigma.\text{act}} = m_{\sigma} \). We conclude by (G4).

Ad (G5). We conclude by (P5) and \( \text{retired}_{\tau.\text{act}} = \text{retired}_{\tau} \setminus \{ a \} \).
\[ \text{Case 8: } com \equiv \text{env}(a) \]

The update is \( \text{up} = [a.\text{next} \mapsto \text{seg} \cdot a.\text{data} \mapsto d] \) for some \( d \). By definition, \( a \in \text{fresh}_r \cup \text{freed}_r \).

This implies \( a \notin \text{adr}(m_r|\text{valid}_r) \): if \( a \notin \text{freed}_r \) if follows from (P6) and otherwise from Lemmas B.35 and B.43. We show that \( \hat{\sigma} = \sigma \) is an adequate choice. We have:

\[
\begin{align*}
    \text{dom}(m_{r,act}|\text{valid}_{r,act}) &= \text{valid}_{r,act} \cup \text{DVar} \cup \{ b.\text{data} \mid b \in m_{r,act}(\text{valid}_{r,act}) \} \\
    &= \text{valid}_{r,act} \cup \text{DVar} \cup \{ b.\text{data} \mid b \in m_r(\text{valid}_r) \} \\
    &= \text{valid}_{r,act} \cup \text{DVar} \cup \{ b.\text{data} \mid b \in m_r(\text{valid}_r) \} = \text{dom}(m_r|\text{valid}_r)
\end{align*}
\]

where the first and last equality is by definition, the second equality by \( \text{valid}_{r,act} = \text{valid}_r \), and the third equality because \( a.\text{next} \notin \text{valid}_r \) by \( a \notin \text{adr}(m_r|\text{valid}_r) \) together with Lemma B.35. Similarly, \( a.\text{data} \notin \text{valid}_r \) because \( a \notin \text{adr}(m_r|\text{valid}_r) \) and Lemma B.35. Altogether, \( m_r|\text{valid}_r = m_{r,act}|\text{valid}_{r,act} \) and \( m_r(\text{valid}_r) = m_{r,act}(\text{valid}_{r,act}) \) and \( \text{adr}(m_r|\text{valid}_r) = \text{adr}(m_{r,act}|\text{valid}_{r,act}) \). Furthermore, we have \( \text{ctrl}(r) = \text{ctrl}(r.\text{act}) \) and \( \text{valid}_r = \text{valid}_{r,act} \) as well as \( \text{fresh}_r = \text{fresh}_{r,act} \) and \( \text{freed}_r = \text{freed}_{r,act} \). 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 \( \text{pexp}, \text{qexp} \in \text{VExp}(r.\text{act}) \) with \( m_{r,act}(\text{pexp}) \neq m_r(\text{pexp}) \).

We have \( \text{pexp}, \text{qexp} \in \text{VExp}(r) \) by definition. Note that we must have \( \text{pexp} \neq a.\text{next} \neq \text{qexp} \) because of \( a \notin \text{adr}(m_r|\text{valid}_r) \). Hence, we obtain \( m_{r,act}(\text{pexp}) = m_r(\text{pexp}) \). Then, (G4) follows from (P4). Finally, the remaining (G8) follow from (P8) since \( \text{act} \) does not raise unsafe accesses.

\[ \text{Case 9: } com \equiv \text{assume \ cond \ and } \sigma.\text{act} \in \mathcal{O}\left[ P \right]_{\mathcal{A}d_{r}}^{<} \]

The update is \( \text{up} = \emptyset \). We show that \( \hat{\sigma} = \sigma.\text{act} \) is an adequate choice. To that end, observe that we have \( m_r(\text{valid}_r) = m_{r,act}(\text{valid}_{r,act}) \) and \( \text{adr}(m_{r,act}|\text{valid}_{r,act}) = \text{adr}(m_r|\text{valid}_r) \) by Lemma B.47.

\[ \text{Ad (G0). } \text{Holds by assumption.} \]

\[ \text{Ad (G1) and (G2). } \text{Follow from Lemma B.68 together with (P1) and (P2).} \]

\[ \text{Ad (G3). } \text{Follows from (P3) together with } \text{fresh}_r \text{ is } \text{fresh}_{r,act} \text{ as well as } \mathcal{H}(r) = \mathcal{H}(r.\text{act}) \text{ and } \mathcal{H}(\sigma) = \mathcal{H}(\sigma.\text{act}). \]

\[ \text{Ad (G7). } \text{By definition, } \text{freed}_{r,act} = \text{freed}_r \cup \{ a \} \text{ and } \text{retired}_{r,act} = \text{retired}_r \setminus \{ a \}. \text{ Then, we conclude by (P7).} \]
Case 10: \[\text{com} \equiv \text{assume cond and } \sigma.\text{act} \notin \mathbb{O}[P]_{\text{Adr}}\]
The update is \(u = \emptyset\). We have \(m_\tau(u) = m_{\sigma}(u)\) for all \(u \in DVar\) by (P1). Furthermore by (P4), \(m_\tau(p) \neq m_\sigma(q)\) implies \(m_\sigma(p) \neq m_\sigma(q)\). Hence, for \(\text{act}\) to be not enabled after \(\sigma\), 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 \(m_\tau(p) \in \text{fresh}_\tau\). 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 \(\sigma\) 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: \[\text{com} \in \{\text{skip}, \text{beginAtomic}, \text{endAtomic}, \text{@inv}\}\]
We immediately obtain that \(\tilde{\sigma} = \sigma.\text{act}\) satisfies the claim.

The above case distinction is complete and thus concludes the induction. \(\blacksquare\)

Proof C.57 (Theorem B.72). We proceed by induction over the structure of \(\tau\). In the base case, we have \(\tau = \varepsilon\) and the claim follows immediately. For the induction step, consider \(\tau.\text{act} \in \mathbb{O}[P]_{\text{Adr}}\) UAF and assume that we have already constructed \(\sigma\) with the following properties:

(P1) \(\sigma \in \mathbb{O}[P]_{\emptyset}\)
(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 \(\tilde{\sigma}\) such that:

(G1) \(\tilde{\sigma} \in \mathbb{O}[P]_{\emptyset}\)
(G2) \(\text{ctrl}(\tau.\text{act}) = \text{ctrl}(\tilde{\sigma})\)
(G3) \( (\exp \cap \Adr) \cap (\fresh_{\act} \cup \freed_{\act}) = \emptyset \implies m_{\act} = m_{\sigma}\)

(G4) \(\fresh_{\act} = \fresh_{\sigma}\)

(G5) \(\freed_{\act} \cup \retired_{\act} = \retired_{\sigma}\)

(G6) \(\inv(\act) \equiv \inv(\hat{\sigma})\)

Let \(\act = \langle t, \com, \up \rangle\).

- **Case 1:** \(\com \equiv p := q.\next\)
  
  Let \(a = m_{\sigma}(q)\) and \(b = m_{\sigma}(a.\next)\). By definition, we have \(a \neq \seg\) and \(\up = [p \mapsto b]\). We choose \(\hat{\sigma} = \sigma.\act\).

  - **Ad (G1).** Since \(\act\) is UAF, we have \(q \in \valid_{\tau}\). By definition, \(q \cap \Adr = \emptyset\). Moreover, the contrapositive of Lemmas B.43 and B.48 gives \(a \notin \fresh_{\tau} \cup \freed_{\tau}\). Then, (P3) gives us \(m_{\sigma}(p) = a\) and \(m_{\sigma}(a.\next) = b\). Hence, \(\sigma.\act \in O[\P]_{\Adr}\) as required.

  - **Ad (G2).** Follows by (P2) together with executing the same action \(\act\) after both \(\tau\) and \(\sigma\).

  - **Ad (G3).** Consider \(\exp\) such that \(\exp \cap \Adr \cap (\fresh_{\act} \cup \freed_{\act}) = \emptyset\). If \(\exp \equiv p\), then \(m_{\act} = m_{\sigma}(\exp)\) due to the form of the update \(\up\). Otherwise, we have \(m_{\act} = m_{\sigma}(\exp)\) and \(m_{\sigma}(\exp) = m_{\tau}(\exp)\). Because of \(\fresh_{\act} = \fresh_{\tau}\) and \(\freed_{\act} = \freed_{\tau}\), (G3) yields \(m_{\tau}(\exp) = m_{\sigma}(\exp)\). Altogether, we arrive at the desired \(m_{\act}(\exp) = m_{\sigma}(\exp)\).

  - **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 2:** \(\com \in \{ p := q, \ p.\next := q, \ u := \op(\overline{u}), \ u := q.\data, \ p.\data := u \}\)

  Analogous to previous case.

- **Case 3:** \(\com \in \{ \in: \func(\overline{v}), \ \re: \func \ \skip \}\)

  We choose \(\hat{\sigma} = \sigma.\act\). By definition and (P2), \(\sigma.\act \in [\P]_{\Adr}^{\emptyset}\) 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 \(x \cap \Adr = \emptyset\) and (P3), we obtain that \(m_{\tau}(x) = m_{\sigma}(x)\). Consequently, \(\cond\) has the same truth value in \(\tau\) and \(\sigma\) since it contains only variables \(x\). By (P2), we obtain \(\sigma.\act \in [\P]^{\emptyset}_{\Adr}\) 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:** \(p := \malloc\)

  We choose \(\hat{\sigma} = \sigma.\act\). Let \(a = m_{\act}(p)\). By \(\tau.\act \in O[\P]_{\Adr}^{\emptyset}\), we have \(a \in \fresh_{\tau}\).
\(\text{Ad (G1).}\) We get \(a \in \text{fresh}_\sigma\) from (P4). This means \(\sigma.\text{act} \in \llbracket P \rrbracket^\sigma_\sigma\).

\(\text{Ad (G3).}\) Consider \(\text{exp}\) with \((\text{exp} \cap \text{Adr}) \cap (\text{fresh}_{t.\text{act}} \cup \text{freed}_{t.\text{act}})\). If \(\text{exp} \notin \{p, a.\text{next}\}\), we have \(m_{t.\text{act}}(\text{exp}) = m_{\sigma.\text{act}}(\text{exp})\) by up updating \(\text{exp}\). Otherwise, \(m_{t.\text{act}}(\text{exp}) = m_t(\text{exp})\) and \(m_{\sigma.\text{act}}(\text{exp}) = m_\sigma(\text{exp})\). Moreover, \(\text{fresh}_{t.\text{act}} \cup \text{fresh}_{t.\text{act}} \subseteq \text{fresh}_t \cup \text{freed}_t\). Hence, we conclude by (P3).

\(\text{Ad (G4).}\) We conclude by (P4): \(\text{fresh}_{t.\text{act}} = \text{fresh}_t \setminus \{a\} = \text{fresh}_\sigma \setminus \{a\} = \text{fresh}_{\sigma.\text{act}}\).

\(\text{Ad (G5).}\) By definition, we have:

\[
\text{freed}_{t.\text{act}} \cup \text{retired}_{t.\text{act}} = (\text{freed}_{t} \setminus \{a\}) \cup \text{retired}_t = \text{freed}_t \cup \text{retired}_t
\]

where the last equality holds by \(a \notin \text{freed}_t\), which follows from \(a \notin \text{fresh}_t\) and Lemma B.44. By definition, \(\text{retired}_{\sigma.\text{act}} = \text{retired}_\sigma\). Moreover, \(\text{freed}_t \cup \text{retired}_t = \text{retired}_\sigma\) by (G5). Hence, we conclude the desired \(\text{freed}_{t.\text{act}} \cup \text{retired}_{t.\text{act}} = \text{retired}_{\sigma.\text{act}}\).

\(\text{Ad (G2) and (G6).}\) Follows by (P2) and (P6) together with \(\text{act}\) not affecting the invariants.

\(\text{Case 6: } \text{free}(a)\)

We choose \(\hat{\sigma} = \sigma\).

\(\text{Ad (G1).}\) Follows from (P1).

\(\text{Ad (G2).}\) Follows from (G2) together with \(\text{ctrl}(\tau) = \text{ctrl}(t.\text{act})\) by definition.

\(\text{Ad (G3).}\) Follows from (G3) together with \(m_{t.\text{act}} = m_t\) and \(m_{\sigma.\text{act}} = m_\sigma\).

\(\text{Ad (G4).}\) By definition, we have \(\text{fresh}_{t.\text{act}} = \text{fresh}_t \setminus \{a\}\). By Lemma B.46, \(a \in \text{retired}_t\).

By Lemma B.44, \(a \notin \text{fresh}_t\). Hence, \(\text{fresh}_{t.\text{act}} = \text{fresh}_\sigma\) where the last equality holds by (P4).

\(\text{Ad (G5).}\) By definition, we have:

\[
\text{freed}_{t.\text{act}} \cup \text{retired}_{t.\text{act}} = (\text{freed}_{t} \cup \{a\}) \cup (\text{retired}_{t} \setminus \{a\})
\]

\[
= \text{freed}_t \cup \text{retired}_t \cup \{a\} = \text{freed}_t \cup \text{retired}_t
\]

where the last equality holds by \(a \in \text{retired}_t\) due to Lemma B.46. Finally, (G5) establishes the desired \(\text{freed}_{t.\text{act}} \cup \text{retired}_{t.\text{act}} = \text{retired}_{\sigma.\text{act}}\).

\(\text{Ad (G6).}\) Follows from (P6) together with \(\text{inv}(t.\text{act}) = \text{inv}(\tau)\).

\(\text{Case 7: } \text{env}(a)\)

We choose \(\hat{\sigma} = \sigma\). By definition, \(a \in \text{fresh}_t \cup \text{freed}_t\).

\(\text{Ad (G1).}\) Follows from (P1).
\[\checkmark \text{Ad (G2).} \text{ Follows from (G2) together with } \text{ctrl}(\tau) = \text{ctrl}(r \cdot \text{act}) \text{ by definition.}\]

\[\checkmark \text{Ad (G3).} \text{ Consider some } \text{exp} \text{ with } (\text{exp} \cap \text{Adr}) \cap (\text{fresh}_{r \cdot \text{act}} \cup \text{freed}_{r \cdot \text{act}}) = \emptyset. \text{ This means we have } \text{exp} \notin \{ a\text{.next}, a\text{.data} \}. \text{ So } m_{r \cdot \text{act}}(\text{exp}) = m_{r}(\text{exp}). \text{ Then, } \text{fresh}_{r \cdot \text{act}} = \text{fresh}_{r} \text{ as well as } \text{freed}_{r \cdot \text{act}} = \text{freed}_{r} \text{ together with (G3) yield } m_{r \cdot \text{act}}(\text{exp}) = m_{r}(\text{exp}) = m_{\sigma}(\text{exp}).\]

\[\checkmark \text{Ad (G4) to (G6).} \text{ The remaining properties follow immediately from (P4) to (P6) together with action } \text{act} \text{ not affecting the fresh/freed/retired addresses nor the invariants.}\]

\[\checkmark \text{Case 8: } \text{com} \in \{ \text{skip, beginAtomic, endAtomic} \}\]

We immediately obtain that \(\hat{s} = \sigma \cdot \text{act}\) satisfies the claim.

\[\checkmark \text{Case 9: } \text{com} \equiv @\text{inv} \bullet \]

We choose \(\hat{s} = \sigma \cdot \text{act}\). By definition and (P2), we have \(\sigma \cdot \text{act} \in \{ P \} \sigma\), satisfying (G1). Because \(\text{act}\) 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 \(\text{inv}_{\sigma}(\text{act}) \equiv \text{inv}_{\sigma}(\text{act})\) according to the definition of invariants together with (P6).

\[\checkmark \text{Case 9.1: } \text{com} \equiv @\text{inv} \text{ angel } r \]

By definition, we immediately obtain \(\text{inv}_{\sigma}(\text{act}) \equiv \exists r \cdot \text{true} \equiv \text{inv}_{\sigma}(\text{act})\).

\[\checkmark \text{Case 9.2: } \text{com} \equiv @\text{inv} \text{ p } = q \]

Let \(a = m_{r}(p)\) and \(b = m_{r}(q)\). By definition, \(\{ p, q \} \cap \text{Adr} = \emptyset\). Consequently, (G3) yields \(m_{\sigma}(p) = a\) and \(m_{\sigma}(q) = b\). Hence, \(\text{inv}_{\sigma}(\text{act}) \equiv a = b \land \text{true} \equiv \text{inv}_{\sigma}(\text{act})\) as required.

\[\checkmark \text{Case 9.3: } \text{com} \equiv @\text{inv} \text{ p in } r \]

As before, \(m_{r}(p) = a = m_{\sigma}(p)\) by (G3). Then, \(\text{inv}_{\sigma}(\text{act}) \equiv a \in r \land \text{true} \equiv \text{inv}_{\sigma}(\text{act})\).

\[\checkmark \text{Case 9.4: } \text{com} \equiv @\text{inv} \text{ active}(p) \]

As before, \(m_{r}(p) = a = m_{\sigma}(p)\) by (G3). By (G3), we have \(\text{active}(\tau) = M = \text{active}(\sigma)\) for some set of addresses \(M \subseteq \text{Adr}\). Then, \(\text{inv}_{\sigma}(\text{act}) \equiv a \in M \land \text{true} \equiv \text{inv}_{\sigma}(\text{act})\) as required.

\[\checkmark \text{Case 9.5: } \text{com} \equiv @\text{inv} \text{ active}(r) \]

As before, \(m_{r}(p) = a = m_{\sigma}(p)\) by (G3) and \(\text{active}(\tau) = M = \text{active}(\sigma)\) by (G3). Then, we arrive at the required \(\text{inv}_{\sigma}(\text{act}) \equiv r \subseteq M \land \text{true} \equiv \text{inv}_{\sigma}(\text{act})\).

The above case distinction is complete and thus concludes the induction.  

\textbf{Proof C.58 (Theorem A.10).} If \(\text{good}(O[\{ P \}]_{\text{Adr}})\) then, \(\text{good}(\{ P \} \sigma)\) follows by \(\{ P \} \sigma \subseteq O[\{ P \}]_{\text{Adr}}\). For the reverse direction, assume \(\text{good}(\{ P \} \sigma)\) holds. To the contrary, assume \(\text{good}(O[\{ P \}]_{\text{Adr}})\) does not hold. There is \(\tau \in O[\{ P \}]_{\text{Adr}}\) so that \(\text{good}(\tau)\) is not satisfied. Hence, \(\text{ctrl}(\tau) \cap \text{Fault} \neq \emptyset\) where \text{Fault} are the bad control locations. Theorem B.71 yields \(\sigma \in O[\{ P \}]_{\text{Adr}}\) with \(\tau \sim \sigma\). Note
that $\sigma$ is MPRF by assumption. Then, Theorem B.72 yields $\gamma \in \llbracket P \rrbracket^\text{Adr}_{\sigma}$ with $\text{ctrl}(\sigma) = \text{ctrl}(\gamma)$. Altogether, we arrive at $\text{ctrl}(\tau) = \text{ctrl}(\gamma)$ and thus $\text{ctrl}(\gamma) \cap \text{Fault} \neq \emptyset$. Hence, $\text{good}(\llbracket P \rrbracket^\text{Adr}_{\sigma})$ does not hold. Since this contradicts the assumption, we obtain the desired $\text{good}(\mathcal{O}\llbracket P \rrbracket^\text{Adr}_{\sigma})$.

Now, we show that $\mathcal{O}\llbracket P \rrbracket^\text{Adr}_{\sigma}$ is free from double retires. To the contrary, assume there is a shortest computation $r.\text{act} \in \mathcal{O}\llbracket P \rrbracket^\text{Adr}_{\sigma}$ with $\text{act} = \langle t, \text{in:retire}(p), \text{up} \rangle$ and $m_r(p) \in \text{retire}(\tau)$. Theorem B.71 yields $\sigma \in \mathcal{O}\llbracket P \rrbracket^\text{Adr}_{\sigma}$ with $\tau \sim \sigma$ and $\text{retire}_r \subseteq \text{retire}_\sigma$. By assumption, $\sigma$ is MPRF. If $p \notin \text{valid}_\sigma$, then $m_\sigma(p) \in \text{freed}_\sigma$ by Lemma B.52. So, $m_{\sigma,\text{act}}(p) \in \text{freed}_\sigma \cup \text{retire}_\sigma$. Then, Lemma B.70 yields a double retire in $\mathcal{O}\llbracket P \rrbracket^\text{Adr}_{\sigma}$. Since this contradicts the assumption, we must have $p \in \text{valid}_\sigma$. Hence, $m_\sigma(p) = m_{\sigma,\text{act}}(p) \in \text{retire}_\sigma$. This means $\sigma.\text{act}$ is a double retire in $\mathcal{O}\llbracket P \rrbracket^\text{Adr}_{\sigma}$. This contradicts the assumption. So, $\mathcal{O}\llbracket P \rrbracket^\text{Adr}_{\sigma}$ is free from double retires. $lacksquare$

**Proof C.59** (Theorem 8.5). Set $C\text{Var} = \emptyset$ so that $\mathcal{O}\llbracket P \rrbracket^\text{Adr}_{\sigma}$ satisfies Assumption A.9. Note that SPRF implies MPRF. Then, the claim follows from Theorem B.71. $lacksquare$

**Proof C.60** (Theorem 8.6). Set $C\text{Var} = \emptyset$ so that $\mathcal{O}\llbracket P \rrbracket^\text{Adr}_{\sigma}$ satisfies Assumption A.9. Note that SPRF implies MPRF. Then, the claim follows from Theorem B.72. $lacksquare$

**Proof C.61** (Theorem 8.7). Set $C\text{Var} = \emptyset$ so that $\mathcal{O}\llbracket P \rrbracket^\text{Adr}_{\sigma}$ satisfies Assumption A.9. Note that SPRF implies MPRF. Then, the claim follows from Theorem A.10. $lacksquare$

### C.4 Type System

**Proof C.62** (Lemma B.73). By definition. $lacksquare$

**Proof C.63** (Lemma B.74). Consider $\vdash \{ \Gamma_1 \} \text{stmt} \{ \Gamma_2 \}$ and let $\text{stmt} \equiv \text{com. stmt}$. 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 (\text{infer}), (\text{seq}), (\text{choice}), or (\text{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 $\text{stmt} \equiv \text{com. stmt}$. We do a case distinction on the first rule.

- **Case 1**: Rule (\text{seq}), part 1
  
  We have $\text{stmt} \equiv \text{stmt}_1; \text{stmt}_2$. There is $\Gamma$ 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: \( \text{stmt}_1 \neq \text{skip} \)

By definition, we have \( \text{stmt}^l \equiv \text{stmt}^l_1; \text{stmt}^l_2 \) and \( \text{stmt}^l_1 \xrightarrow{\text{com}} \text{stmt}^l_1 \). We invoke the induction hypothesis for \( \vdash \{ \Gamma_1 \} \text{stmt}_1 \{ \Gamma \} \) and \( \text{stmt}^l_1 \xrightarrow{\text{com}} \text{stmt}^l_1 \). This yields some \( \Gamma' \) such that \( \vdash \{ \Gamma_1 \} \text{com} \{ \Gamma' \} \) and \( \vdash \{ \Gamma' \} \text{stmt}^l_1 \{ \Gamma \} \). By Rule (SEQ) then, \( \vdash \{ \Gamma' \} \text{stmt}^l \{ \Gamma_2 \} \).

\* Case 2: Rule (choice)

We have \( \text{stmt} \equiv \text{stmt}^l_1 \oplus \text{stmt}^l_2 \). By definition, we have \( \text{stmt} \xrightarrow{\text{com}} \text{stmt}^l_i \) for \( i \in \{1, 2\} \) and \( \text{com} \equiv \text{skip} \). The type rules gives \( \vdash \{ \Gamma_1 \} \text{stmt}^l \{ \Gamma_2 \} \). Moreover, \( \vdash \{ \Gamma_2 \} \text{com} \{ \Gamma_2 \} \).

\* Case 3: Rule (loop)

We have \( \text{stmt} \equiv \text{stmt}^l_i \) and \( \Gamma_1 \equiv \Gamma_2 \). By definition, we have \( \text{stmt}^l \equiv \{ \text{skip}, \text{stmt}^l_1; \text{stmt}^l \} \) and \( \text{com} \equiv \text{skip} \). We immediately get \( \vdash \{ \Gamma_1 \} \text{com} \{ \Gamma_1 \} \). Moreover, \( \vdash \{ \Gamma_1 \} \text{stmt}^l \{ \Gamma_2 \} \) follows from Rule (skip) in case of \( \text{stmt}^l \equiv \text{skip} \) and from Rule (seq) otherwise.

\* Case 4: Rule (infer)

There are type environments \( \Gamma_1 \) and \( \Gamma_2 \) such that \( \Gamma_1 \leadsto \Gamma_1 \) and \( \vdash \{ \Gamma_1 \} \text{stmt} \{ \Gamma_2 \} \), and \( \Gamma_2 \leadsto \Gamma_2 \).

By induction for \( \vdash \{ \Gamma_1 \} \text{stmt} \{ \Gamma_4 \} \), there is \( \Gamma \) with \( \vdash \{ \Gamma_3 \} \text{com} \{ \Gamma \} \) and \( \vdash \{ \Gamma \} \text{stmt}^l \{ \Gamma_1 \} \).

Applying Rule (infer) we get \( \vdash \{ \Gamma_1 \} \text{com} \{ \Gamma \} \) and \( \vdash \{ \Gamma \} \text{stmt}^l \{ \Gamma_2 \} \) as desired.

The above case distinction concludes the induction.

\[\Box\]

\*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, \( (\text{pc}_{\text{init}}, \epsilon) \rightarrow^0 (\text{pc}, \tau) \). That is, \( \text{pc} = \text{pc}_{\text{init}} \) and \( \tau = \epsilon \). Let \( t \) be a thread.

By definition, \( \text{stmt}(\tau,t) = \text{skip} \) and \( \text{pc}(t) = P[t] \). The former gives \( \vdash \{ \Gamma_{\text{init}}[i] \} \text{stmt}(\tau,t) \{ \Gamma_{\text{init}}[i] \} \).

The latter gives \( \vdash \{ \Gamma_{\text{init}}[i] \} \text{pc}(t) \{ \Gamma \} \) due to the premise. So we choose \( \Gamma_1 = \Gamma_{\text{init}}[i] \) and \( \Gamma_2 = \Gamma \).

For the induction step, consider \( (\text{pc}_{\text{init}}, \epsilon) \rightarrow^\ast (\text{pc}, \tau) \rightarrow^\ast (\text{pc}^j, \tau, \text{act}) \). Let \( t \neq \bot \) be some arbitrary thread. By induction, there are \( \Gamma_{\text{init}}[i], \Gamma_2 \) with:

\( \vdash \{ \Gamma_{\text{init}}[i] \} \text{stmt}(\tau,t) \{ \Gamma_{\text{init}}[i] \} \) and \( \vdash \{ \Gamma_{\text{init}}[i] \} \text{pc}(t) \{ \Gamma_2 \} \).

First, assume \( t \neq t^j \). Then, \( \text{stmt}(\tau,\text{act}) = \text{stmt}(\tau,t) \) and \( \text{pc}(t) = \text{pc}(t) \). Thus, the claim follows by induction for \( \Gamma_1 = \Gamma_{\text{init}}[i] \). So consider \( t = t^j \) now. Then, \( \text{stmt}(\tau,\text{act}) = \text{stmt}(\tau,t) \); \text{com}.

By definition, we have \( \text{pc}(t) \xrightarrow{\text{com}} \text{pc}(t) \) and \( \text{act} \in \text{Act}(\tau,t,\text{com}) \). Lemma B.74 yields \( \Gamma_1 \) with:

\( \vdash \{ \Gamma_{\text{init}}[i] \} \text{stmt}(\tau,t) \{ \Gamma_{\text{init}}[i] \} \) and \( \vdash \{ \Gamma_{\text{init}}[i] \} \text{pc}(t) \{ \Gamma_2 \} \).

Altogether, we get \( \vdash \{ \Gamma_{\text{init}}[i] \} \text{stmt}(\tau,\text{act}) \{ \Gamma \} \) and \( \vdash \{ \Gamma_{\text{init}}[i] \} \text{pc}(t) \{ \Gamma_2 \} \) as required.

\[\Box\]

\*Proof C.65 (Lemma B.76)\* Let \( \tau, \tau^j \in \text{Ctrl}[P]_{\text{adr}} \). Let \( t, t^j \) be threads with \( t \neq t^j = \text{thr}(\text{act}) \). By definition of the semantics, there is a step \( (\text{pc}, \tau) \rightarrow (\text{pc}^j, \tau, \text{act}) \) for some \( \text{pc} \in \text{ctrl}(\tau) \). If we have \( \text{stmt}(\tau,t) = \text{skip} \), nothing needs to be shown. So assume \( \text{stmt}(\tau,t) \neq \text{skip} \). That is, \( t \) has contributed actions to \( \tau \). By Assumption 8.11, \( t \) must have entered an atomic block to do so. As \( t \neq \cdots \)
Applying the same argument inductively on $t$, we obtain that the last primitive command of $t$ must have been followed by an endAtomic. Hence, $stmt(t, t) = stmt; endAtomic$ for some sequence of statements $stmt$, as required.

**Proof C.66** (Lemma B.77). Let $r.act \in \mathcal{O}[\mathbb{P}]^{\mathbb{A}_{adr}}$. Let $t, t'$ be threads with $t \neq t' = \text{thrd}(act)$. Moreover, let $x \in \mathbb{PVar} \cup \mathbb{AVar}$. Assume $\vdash \{ \Gamma_{\text{init}}^{[r]} \} \cdot stmt(t, t) \{ \Gamma \}$. There are two cases.

- **Case 1:** $stmt(t, t) = \text{skip}$

  By the type rules there is $\Gamma'$ with:

  $$\Gamma_{\text{init}}^{[r]} \sim \Gamma' \quad \text{and} \quad \vdash \{ \Gamma' \} \cdot \text{skip}(\Gamma'). \quad \text{and} \quad \Gamma' \sim \Gamma.$$

  By definition, we have $\Gamma_{\text{init}}^{[r]}(x) = \emptyset$. We obtain $\neg \text{isValid}(\Gamma_{\text{init}}^{[r]}(x))$. Hence, $\neg \text{isValid}(\Gamma'(x))$ and $\neg \text{isValid}(\Gamma(x))$ follow by type inference. We conclude $\Gamma(x) \cap \{ \text{A, L, S} \} = \emptyset$.

- **Case 2:** $stmt(t, t) = stmt; endAtomic$

  By the typing rules there are $\Gamma_1, \Gamma_2, \Gamma_3$ with:

  $$\vdash \{ \Gamma_{\text{init}}^{[r]} \} \cdot stmt(\Gamma_1) \quad \text{and} \quad \Gamma_1 \sim \Gamma_2 \quad \text{and} \quad \vdash \{ \Gamma_2 \} \cdot \text{endAtomic}(\Gamma_3) \quad \text{and} \quad \Gamma_3 \sim \Gamma$$

  where the derivation $\vdash \{ \Gamma_2 \} \cdot \text{endAtomic}(\Gamma_3)$ is due to Rule (END). This means, that we have $\Gamma_3 = \text{rm}(\Gamma_2)$. By definition, $\text{A} \notin \Gamma_3(x)$. Hence, type inference provides $\text{A} \notin \Gamma(x)$ as desired. Consider $x \notin \text{local}_r$ now. There are two cases. First, assume $x \in \text{shared}_r$. Then, we get $\Gamma_3(x) = \emptyset$ by definition of Rule (END). Second, assume $x \notin \text{shared}_r$. That is, $x$ is local to another thread $t'' \neq t$, i.e., $x \in \text{local}_{r''}$. 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 $r.act \in \mathcal{O}[\mathbb{P}]^{\mathbb{A}_{adr}}$. Let $t, t'$ be threads with $t \neq t' = \text{thrd}(act) \neq \bot$. Consider $p \in \mathbb{PVar} \cap \text{local}_r$. By definition, $\text{local}_r \cap \text{local}_{r'} = \emptyset$. Due to the semantics, $p$ does not occur in $\text{com}(act)$. Hence, $p \in \text{valid}_t \iff p \in \text{valid}_{t', \text{act}}$. Further, $m_t(p) = m_{t, \text{act}}(p)$. So every valid alias created by $act$ requires a valid alias in $r$. This is not possible since $\text{noalias}_r(p)$.

**Proof C.68** (Lemma B.79). Let $r.act \in \mathcal{O}[\mathbb{P}]^{\mathbb{A}_{adr}}$ UAF with $\text{act} = \langle t, \text{@inv } p = q, \emptyset \rangle$ and $\text{inv}(r.\text{act})$. The latter gives $m_t(p) = m_t(q)$. Then, Lemma B.53 gives $p, q \in \text{valid}_r$ as required.

**Proof C.69** (Lemma B.80). Let $r.act \in \mathcal{O}[\mathbb{P}]^{\mathbb{A}_{adr}}$ UAF such that $\text{act} = \langle t, \text{@inv } \text{active}(p), ap \rangle$ and $\text{inv}(r.\text{act})$. By definition, this means $m_t(p) \in \text{active}(r)$. So, $m_t(p) \notin \text{freed}_r$. Lemma B.52
This concludes the claim. \[ \mathcal{P} \] Let \( \text{Lemma B.82} \)

Proof C.70 (Lemma B.81). Let \( \tau \in \mathcal{O}[P]^{\sim}_{\text{Adr}} \) such that \( \text{act} = \langle t, \# \text{inv active}(r), \text{up} \rangle \) and \( \text{inv}(\tau.\text{act}) \). By definition, we have \( \text{repr}_r(r) \subseteq \text{active}(\tau) \). Hence, \( \text{repr}_r(r) \cap \text{freed}_r = \emptyset \) holds. Moreover, we also have \( \text{repr}_r(\tau.\text{act}) \subseteq \text{active}(\tau.\text{act}) \) by definition. Let \( a \in \text{repr}_r(\tau.\text{act}) \). This means \( a \in \text{active}(\tau.\text{act}) \). That is, we have \( a \notin \text{retired}_r \). Then, the remaining property follows from Lemma B.45.

Proof C.71 (Lemma B.82). Let \( \tau \in \mathcal{O}[P]^{\sim}_{\text{Adr}} \). Let \( \Gamma, \Gamma' \) be two type environments with \( \Gamma \sim \Gamma' \).

Let \( p \in PVar \) be a pointer with \( a = m_r(p) \) and let \( r \in AVar \) be a ghost variable and \( b \in \text{repr}_r(r) \).

- Assume \( \text{isValid}(\Gamma(p)) \implies p \in \text{valid}_r \).
  By \( \Gamma \sim \Gamma' \), we have \( \text{isValid}(\Gamma'(p)) \implies \text{isValid}(\Gamma(p)) \).
  Hence, we obtain the desired \( \text{isValid}(\Gamma'(p)) \implies p \in \text{valid}_r \).
- Assume \( \text{isValid}(\Gamma(r)) \implies b \notin \text{freed}_r \).
  By \( \Gamma \sim \Gamma' \), we have \( \text{isValid}(\Gamma'(r)) \implies \text{isValid}(\Gamma(r)) \).
  Hence, we obtain the desired \( \text{isValid}(\Gamma'(r)) \implies b \notin \text{freed}_r \).
- Assume \( \mathbb{L} \in \Gamma(p) \implies \text{noalias}_s(p) \).
  By \( \Gamma \sim \Gamma' \), we have \( \mathbb{L} \in \Gamma'(p) \implies \mathbb{L} \in \Gamma(p) \).
  Hence, we obtain the desired \( \mathbb{L} \in \Gamma'(p) \implies \text{noalias}_s(p) \).
- Assume \( \text{reach}_{t,a}^O(\mathcal{H}(r)) \subseteq \text{Loc}(\Gamma(p)) \). By \( \Gamma \sim \Gamma' \), we have \( \text{Loc}(\Gamma(p)) \subseteq \text{Loc}(\Gamma'(p)) \).
  Hence, we obtain the desired \( \text{reach}_{t,a}^O(\mathcal{H}(r)) \subseteq \text{Loc}(\Gamma'(p)) \).
- Assume \( \text{reach}_{t,b}^O(\mathcal{H}(r)) \subseteq \text{Loc}(\Gamma(r)) \). By \( \Gamma \sim \Gamma' \), we have \( \text{Loc}(\Gamma(r)) \subseteq \text{Loc}(\Gamma'(r)) \).
  Hence, we obtain the desired \( \text{reach}_{t,b}^O(\mathcal{H}(r)) \subseteq \text{Loc}(\Gamma'(r)) \).

This concludes the claim.

Proof C.72 (Theorem B.83). Let \( \tau \in \mathcal{O}[P]^{\sim}_{\text{Adr}} \). We proceed by induction over the structure of \( \tau \) in order to show the following:

\[
\text{freed}_r \cap \text{retired}_r = \emptyset \\
\forall t \forall \Gamma. \quad \vdash \{ \Gamma[t] \} \text{stmt}(r, t) \{ \Gamma \} \quad \Rightarrow \quad \forall p,r.
\begin{align*}
\text{reach}_{t,m_r(p)}^O(\mathcal{H}(r)) & \subseteq \text{Loc}(\Gamma(p)) \land \text{isValid}(\Gamma(p)) \implies p \in \text{valid}_r \\
& \land \mathbb{L} \in \Gamma(p) \implies \text{noalias}_s(p) \\
& \land \forall a \in \text{repr}_r(r). \text{reach}_{t,a}^O(\mathcal{H}(r)) \subseteq \text{Loc}(\Gamma(r)) \\
& \land \text{isValid}(\Gamma(r)) \implies \text{repr}_r(r) \cap \text{freed}_r = \emptyset
\end{align*}
\]
Base Case. We have $\tau = \epsilon$. Let $t, \Gamma$ with $\vdash \{ \Gamma^{[r]}_{\text{init}} \} \ stmt(t, t) \ {\Gamma}$. By definition, $\stmt(r, t) = \text{skip}$ as well as $\text{freed}_{r} \cap \text{retired}_{r} = \emptyset$. Let $p \in \text{PVar}$ and $r \in \text{AVar}$. By definition, $p \in \text{valid}_{r}$. This gives the desired implication $\text{isValid}(\Gamma(p)) \implies p \in \text{valid}_{r}$. By the type rules we have:

$$\Gamma^{[r]}_{\text{init}} \vdash \Gamma \quad \text{and} \quad \vdash \{ \Gamma^{[r]}_{\text{init}} \} \ \text{skip} \ \{ \Gamma^{[r]}_{\text{init}} \} \quad \text{and} \quad \Gamma^{[r]}_{\text{init}} \vdash \Gamma.$$ 

Since $\bot \notin \Gamma^{[r]}_{\text{init}}(p)$, we get $\bot \notin \Gamma(p)$ by the definition of type inference. So, we satisfy the implication $\bot \in \Gamma(p) \implies \text{noalias}_{r}(p)$. Moreover, $\text{freed}_{r} = \emptyset$. So, $\text{isValid}(\Gamma(r)) \implies a \notin \text{freed}_{r}$ holds for all $a \in \text{Adr}$. From Lemma B.73, we get $\Gamma^{[r]}_{\text{init}} \vdash \Gamma$. That is $\text{Loc}(\Gamma^{[r]}_{\text{init}}(p)) \subseteq \text{Loc}(\Gamma(p))$. By definition, $\Gamma^{[r]}_{\text{init}}(p) = \emptyset$. That is, $\text{reach}^{O}_{r, m_{r}}(\Gamma(r)) \subseteq \text{Loc}(O) = \text{Loc}(\emptyset) = \text{Loc}(\Gamma^{[r]}_{\text{init}}(p))$. Similarly for $r$. Altogether, this concludes the base case.

Induction Step. Consider $\tau, \text{act}$ UAF with $\text{act} = \{ t', \text{com}, \text{up} \}$ and $\text{inv}(\tau, \text{act})$. Let $t$ be some arbitrary thread. We establish the claim for $t$. To that end, we do a case distinction over thread $t'$ executing $\text{act}$.

\[ \diamond \text{ Case 1: } t = t' \text{ and } t' \neq \bot \]

By definition, we have $\stmt(\tau, t) = \stmt(\tau, t); \text{com}$ and $\text{freed}_{\tau, \text{act}} \subseteq \text{freed}_{\tau}$. First, we establish that $\text{freed}_{\tau, \text{act}} \cap \text{retired}_{\tau, \text{act}} = \emptyset$ holds. If $\text{retired}_{\tau, \text{act}} \subseteq \text{retired}_{\tau}$, then the claim follows by induction. Otherwise, we have $\text{retired}_{\tau, \text{act}} = \text{retired}_{\tau} \cup \{ a \}$ and $\text{com} \equiv \text{in:retire}(q)$ with $m_{r}(q) = a$. Since $\vdash \{ \Gamma_{1} \} \ \text{com} \ \{ \Gamma_{2} \}$ holds, we know $q \in \text{valid}_{r}$. By the contrapositive of Lemma B.52, we get $a \notin \text{freed}_{r}$. So by induction, $\text{freed}_{\tau, \text{act}} \cap \text{retired}_{\tau, \text{act}} = \emptyset$.

Now, Assume that $\tau, \text{act}$ can be typed for $t$ as nothing needs to be shown otherwise. That is, assume there is $\Gamma_{3}$ such that:

$$\vdash \{ \Gamma^{[r]}_{\text{init}} \} \ \stmt(\tau, t) \ \{ \Gamma_{3} \}.$$ 

Due to the type rules and the above equality, we know that there are $\Gamma_{1}, \Gamma_{2}$ with:

$$\vdash \{ \Gamma^{[r]}_{\text{init}} \} \ \stmt(\tau, t) \ \{ \Gamma_{1} \} \quad \text{and} \quad \Gamma_{0} \vdash \Gamma_{1} \quad \text{and} \quad \vdash \{ \Gamma_{1} \} \ \text{com} \ \{ \Gamma_{2} \} \quad \text{and} \quad \Gamma_{2} \vdash \Gamma_{3}$$

where $\vdash \{ \Gamma_{1} \} \ \text{com} \ \{ \Gamma_{2} \}$ is derived by neither (seq), (choice), (loop), nor (infer). Induction yields the claim for $\Gamma_{0}$. So by Lemma B.82 the claim also holds for $\Gamma_{1}$. If the claim holds for $\Gamma_{2}$, then the claim follow for $\Gamma_{3}$ from Lemma B.82 again. It remains to show that the claim holds for $\Gamma_{2}$ relying on $\Gamma_{1}$. Let $p \in \text{PVar}$ and $r \in \text{AVar}$ be arbitrary. Let $m_{r, \text{act}}(p) = c_{p}$ and let $c_{r} \in \text{repr}_{\text{act}}(r)$. We show:

\begin{align*}
\text{(G1)} \quad & \text{reach}^{O}_{p, t}(\text{H}(\tau, \text{act})) \subseteq \text{Loc}(\Gamma_{2}(p)) \\
\text{(G2)} \quad & \text{reach}^{O}_{t, c_{r}}(\text{H}(\tau, \text{act})) \subseteq \text{Loc}(\Gamma_{2}(r)) \\
\text{(G3)} \quad & \text{isValid}(\Gamma_{2}(p)) \implies p \in \text{valid}_{r, \text{act}} \\
\text{(G4)} \quad & \bot \in \Gamma_{2}(p) \implies \text{noalias}_{r, \text{act}}(p)
\end{align*}

\[ \diamond \text{ Case 2: } t = t' \quad \text{and} \quad t' \neq \bot \]

Finally, $\vdash \{ \Gamma^{[r]}_{\text{init}} \} \ \stmt(\tau, t) \ \{ \Gamma_{3} \}$.
We do a case distinction over the type rule applied to derive \( \vdash \{ \Gamma_1 \} \ com \ \{ \Gamma_2 \} \).

**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 \( H(r) = H(r . act) \). Now, assume \( isValid(\Gamma_2(p)) \). By \( \Gamma_2(p) \subseteq \Gamma_1(p) \), we have \( isValid(\Gamma_1(p)) \). Induction yields \( p \notin valid_r \). Since \( valid_r = valid_{r . act} \), 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 \( \text{repr}_r = \text{repr}_{r . act} \) gives \( c_r \notin freed_r \). Due to \( freed_r = freed_{r . act} \), we arrive at (G5). If \( L \in \Gamma_2(p) \), then \( L \in \Gamma_1(p) \) and thus \( noalias_r(p) \) by induction. Since \( m_r = m_{r . act} \) 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 \( m_r = m_{r . act} \) and \( valid_r = valid_{r . act} \).

Moreover, we have \( freed_r = freed_{r . act} \) and \( \text{repr}_r = \text{repr}_{r . act} \) as well as \( H(r) = H(r . act) \). Hence, (G1) to (G5) follow by induction.

**Case 1.3: Rule (**assign1**)**

We have \( freed_r = freed_{r . act} \) and \( \text{repr}_r \equiv \text{repr}_{r . act} \). Hence, (G5) holds by induction because of \( \Gamma_2(r) = \Gamma_1(r) \). If \( L \in \Gamma_2(p) \), then \( p \) cannot appear in \( com \) since Rule (**assign1**)** would have removed \( L \). 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 \( \Gamma_2(p) = \Gamma_1(q) \setminus \{ L \} \). So, induction gives \( q \notin valid_r \). We conclude \( p \in valid_{r . act} \) as required. Second, \( p \) is not assigned to by \( com \). Then, \( \Gamma_2(p) \subseteq \Gamma_1(p) \). Hence, \( p \in valid_r \) by induction and thus \( p \in valid_{r . act} \). This concludes (G3). Note that \( H(r . act) = H(r) \). 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 \( \Gamma_2(p) = \Gamma_1(p) \setminus \{ L \} \). 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 \( \Gamma_2(p) = \Gamma_1(q) \setminus \{ L \} \). Furthermore, \( m_{r . act}(p) = m_r(q) = c_p \). By induction, we have \( reach_{r . act}(H(r)) \subseteq Loc(\Gamma_1(q)) \). Hence, \( reach_{r . act}(H(r . act)) \subseteq Loc(\Gamma_1(q) \setminus \{ L \}) \). We get the desired \( reach_{r . act}(H(r . act)) \subseteq Loc(\Gamma_2(q)) \) 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 \( \text{repr}_r = \text{repr}_{r . act} \), \( valid_r = valid_{r . act} \), and \( freed_r = freed_{r . act} \)
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. $\text{com} \equiv \text{assume} \ C = q$ with $C \in CVar$. By the semantics, we have $m_\tau (C) = m_\tau (q)$. Lemma B.54 yields $q \in \text{valid}_r$. Lemma B.51 and Assumption A.9 gives $C \in \text{valid}_r$. Hence, we get $\text{valid}_r = \text{valid}_{r.'act}$. Moreover, we have $\text{freed}_r = \text{freed}_{r.'act}$ and $\text{repr}_r = \text{repr}_{r.'act}$. We show the required properties. By definition, we have $\Gamma_1 (r) = \Gamma_1 (r)$. Hence, (G5) follows by induction. If $\mathbb{L}_r \in \Gamma_2 (p)$, then $\mathbb{L}_r \in \Gamma_1 (p)$ due to the type rule. This means $\text{noalias}_r (p)$ by induction. Note that $m_\tau = m_{\tau.'act}$. So we get $\text{noalias}_{r.'act}(p)$ by definition together with the above $\text{valid}_r = \text{valid}_{r.'act}$. This establishes (G4). If $\text{isValid}(\Gamma_2 (p))$, then there are two cases. First, $\text{isValid}(\Gamma_1 (p))$ holds. This means we have $p \in \text{valid}_r$ by induction. As stated above, this results in $p \in \text{valid}_{r.'act}$. Second, $\neg \text{isValid}(\Gamma_1 (p))$ holds. Then, $p$ is validated by $\text{com}$. For this to happen, $p$ must appear in $\text{com}$, i.e., $p \in \{ C, q \}$. So, $p \in \text{valid}_r$ as before. This gives (G3). Since $\mathcal{H}(\tau.'act) = \mathcal{H}(\tau)$ and $\text{repr}_r = \text{repr}_{r.'act}$, (G2) continues to hold by induction since $r$ is not affected. It remains to establish (G1). If $p$ does not occur in $\text{com}$ nothing needs to be show. So assume $p$ appears in $\text{com}$. By the the type rule, we have $\Gamma_2 (p) = (\Gamma_1 (C) \land \Gamma_1 (q)) \setminus \{ \mathbb{L} \}$. 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:

$$\text{reach}_{\Gamma_1}^{c_p}(\mathcal{H}(\tau)) \subseteq \text{Loc}(\Gamma_1 (p)) \cap \text{Loc}(\Gamma_1 (q)) = \text{Loc}(\Gamma_1 (p) \land \Gamma_2 (p)) = \text{Loc}(\Gamma_2 (p)).$$

**Case 1.7: Rule (assume1)**

Analogous to the previous case.

**Case 1.8: Rule (equal)**

If $\text{isValid}(\Gamma_2 (p))$, then there are two cases. First, $\text{isValid}(\Gamma_1 (p))$ holds. By induction, this means $p \in \text{valid}_r$. Then, we get $p \in \text{valid}_{r.'act}$ because $\text{valid}_r = \text{valid}_{r.'act}$ by definition. Second, $\neg \text{isValid}(\Gamma_1 (p))$ holds. Then, $p$ is validated by $\text{com}$. For this to happen, $\text{com}$ must be of the form $\text{com} \equiv \text{@} \text{inv} \ p = q$ with $\text{isValid}(q)$. By induction, we have $q \in \text{valid}_r$. Then, B.79 gives $p \in \text{valid}_r$. As before, we get $p \in \text{valid}_{r.'act}$. This concludes (G3). The remaining properties follow analogously to the previous case for Rule (assume1).

For (G1) note that $\text{inv}(\tau.'act)$ gives $m_\tau (p) = m_\tau (q)$ and thus $\text{inv}(\tau) \iff \text{inv}(\tau.'act)$. That is, $\text{repr}_r = \text{repr}_{r.'act}$.

**Case 1.9: Rule (active) for pointers**

If $\text{isValid}(\Gamma_2 (p))$, then there are two cases. First, $\text{isValid}(\Gamma_1 (p))$ holds. By induction, this means $p \in \text{valid}_r$. Then, we get $p \in \text{valid}_{r.'act}$ because $\text{valid}_r = \text{valid}_{r.'act}$ by definition.
Appendix C

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 \( r \cdot act \in O[[P]]_{adv} \). So \( act \) allocates a fresh address. Hence, \( freed_r = freed_{\cdot act} \).

Moreover, \( repr_r \equiv repr_{\cdot act} \). 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, \( c_p \notin retired_r \) and thus \( c_p \notin \text{valid}_{\cdot act} \). Then, \( \text{reach}^{G}_{\cdot act}(\mathcal{H}(\cdot act)) \subseteq \text{Loc}(\mathcal{A}) \). Induction yields \( \text{reach}^{G}_{\cdot act}(\mathcal{H}(\cdot act)) \subseteq \text{Loc}(\mathcal{G}_1(p)) \).

Hence, we conclude (G1) by definition as follows:

\[
\text{reach}^{G}_{\cdot act}(\mathcal{H}(\cdot act)) \subseteq \text{Loc}(\mathcal{G}_1(p)) \cap \text{Loc}(\mathcal{A}) \subseteq \text{Loc}(\mathcal{G}_1(p) \land \mathcal{A}) = \text{Loc}(\mathcal{G}_2(p)).
\]

Case 1.12: Rule (enter)

By definition, \( m_r = m_{r \cdot act} \) and \( repr_r = repr_{\cdot act} \). We have \( \text{reach}^{G}_{\cdot act}(\mathcal{H}(\cdot)) \subseteq \text{Loc}(\mathcal{G}_1(p)) \) by induction. Type inference \( \Gamma_1, \text{com} \leadsto \Gamma_2 \) results in \( post_{\cdot com}(\text{Loc}(\Gamma_1(p))) \subseteq \text{Loc}(\Gamma_2(p)) \).

Hence, we get the desired \( \text{reach}^{G}_{\cdot act}(\mathcal{H}(\cdot \cdot act)) \subseteq \text{Loc}(\Gamma_2(p)) \). This concludes (G1). We conclude (G2) along the same lines. Now, assume \( \text{isValid}(\mathcal{G}_2(p)) \).

Second, \( \neg\text{isValid}(\mathcal{G}_1(p)) \) holds. That is, \( p \) is validated by \( \text{com} \). Hence, \( \text{com} \) must be of the form \( \text{com} \equiv @\text{inv active}(p) \). Since the invariants hold by assumption, \( \text{inv}(r \cdot act) \), we can invoke Lemma B.80. It gives \( p \in \text{valid}_{\cdot act} \). Altogether, this concludes (G3).

If we have \( L \in \Gamma_2(p) \), then \( L \in \Gamma_1(p) \) due to the type rule. Hence, the induction hypothesis together with \( m_r = m_{r \cdot act} \) and \( \text{valid}_r = \text{valid}_{\cdot act} \) gives (G4). For (G5) observe that we have \( \Gamma_2(r) = \Gamma_r(r) \) and \( \text{repr}_r(r) = \text{repr}_{\cdot act}(r) \) because \( \text{inv}(r \cdot act) \). So (G5) follows by induction. By definition, \( \mathcal{H}(\cdot \cdot act) = \mathcal{H}(r) \). Since \( r \) is not affected, (G2) follows by induction. We show (G1). If \( \text{com} \) does not contain \( p \), nothing needs to be show. Otherwise, \( \text{com} \) is of the form \( \text{com} \equiv @\text{inv active}(p) \). From Lemma B.80 we get \( \text{reach}^{G}_{r \cdot act}(\mathcal{H}(\cdot \cdot act)) \subseteq \text{Loc}(\mathcal{A}) \). Induction yields \( \text{reach}^{G}_{r \cdot act}(\mathcal{H}(\cdot \cdot act)) \subseteq \text{Loc}(\mathcal{G}_1(p)) \).

Hence, we conclude (G1) by definition as follows:

\[
\text{reach}^{G}_{r \cdot act}(\mathcal{H}(\cdot \cdot act)) \subseteq \text{Loc}(\mathcal{G}_1(p)) \cap \text{Loc}(\mathcal{A}) \subseteq \text{Loc}(\mathcal{G}_1(p) \land \mathcal{A}) = \text{Loc}(\mathcal{G}_2(p)).
\]
by the definition of type inference. So, (G3) follows by induction and valid \( r = valid_{r,act} \). Similarly, isValid(\( \Gamma_2(r) \)) implies isValid(\( \Gamma_1(r) \)). Then, (G5) follows by induction together with freed \( r = freed_{r,act} \) and repr \( r = repr_{r,act} \). Lastly, assume \( L \in \Gamma_2(p) \). Then \( L \in \Gamma_1(p) \) by definition. We obtain the required noalias \( r,act(p) \) by induction and \( m_r = m_{r,act} \). This concludes (G4).

\[ \diamond \text{Case 1.13: Rule (exit)} \]

Analogously to the previous case for Rule (enter).

\[ \diamond \text{Case 1.14: Rule (angel)} \]

By definition, \( m_r = m_{r,act} \), valid \( r = valid_{r,act} \), and freed \( r = freed_{r,act} \). 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 shown because repr \( r,act(r) = repr_r(r) \) holds due to the fact that annotations cannot correlate different angels. Otherwise, \( \Gamma_2(r) = \emptyset \). This concludes (G2).

\[ \diamond \text{Case 1.15: Rule (member)} \]

By definition, \( m_r = m_{r,act} \) and valid \( r = valid_{r,act} \). If isValid(\( \Gamma_2(p) \)), then isValid(\( \Gamma_1(p) \)) holds since \( \Gamma_2(p) = \Gamma_1(p) \). Moreover, we have inv(\( r,act \)) \( \rightleftarrows inv(r) \) by construction of inv(\( \bullet \)). This means repr \( r,act(r) \) \( \subseteq repr_r(r) \). So, we get \( c_r \notin freed_r \) by induction. This concludes (G5) because of freed \( r = freed_{r,act} \). If \( L \in \Gamma_2(p) \), then \( L \in \Gamma_1(p) \) since angels cannot acquire guarantee \( L \) 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, \( p \in valid_r \) by induction and thus \( p \in valid_{r,act} \). 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 \( m_r(p) \in repr_{r,act}(r') \) from inv(\( r,act \)). Further, isValid(\( \Gamma_1(r') \)) gives isValid(\( \Gamma_2(r') \)). Hence, \( m_r(p) \notin freed_{r,act} \) follows from the already established (G5). The contrapositive of Lemma B.52 yields \( p \in valid_{r,act} \). This establishes (G3). By definition, we have \( \mathcal{H}(r,act) = \mathcal{H}(r) \) 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 \( c_p \in repr_r(r') \). Induction gives:

\[ \text{reach}_{c_p}^{C}(\mathcal{H}(r)) \subseteq \text{Loc}(\Gamma_1(p)) \cap \text{Loc}(\Gamma_1(r')) = \text{Loc}(\Gamma_1(p) \land \Gamma_1(r')) = \text{Loc}(\Gamma_2(p)) . \]

This concludes (G1) because \( \mathcal{H}(r,act) = \mathcal{H}(r) \).
Case 2: $t \neq t'$ and $t' \neq \bot$

We conclude $\text{freet}_{r, \text{act}} \cap \text{retired}_{r, \text{act}} = \emptyset$ as in the previous case. Consider now some $t, \Gamma_4$ such that $\vdash (\Gamma_4^t) \text{stmt}(r, t) \{ \Gamma_4 \}$. By definition, we have:

$$\text{stmt}(r, t) = \text{stmt}(r, t') \quad \text{and} \quad \vdash (\Gamma_4^t) \text{stmt}(r, t) \{ \Gamma_4 \}.$$ 

The induction hypothesis applies to $\vdash (\Gamma_4^t) \text{stmt}(r, t) \{ \Gamma_4 \}$. We have to show that the desired properties are stable under interference.

Consider $p \in PVar$. If $L \in \Gamma_4(p)$, then $p \in \text{local}_r$ by the contrapositive of Lemma B.77. By induction, we have $\text{noalias}_{r, \text{act}}(p)$. Then, Lemma B.78 gives $\text{noalias}_{r, \text{act}}(p)$. If $\text{isValid}(\Gamma_4(p))$, then $p \in \text{local}_r$ by the contrapositive of Lemma B.77 and $p \in \text{valid}_r$ by induction. Lemma B.78 gives $p \in \text{valid}_r$.

Consider $r \in AVar$. If $\text{isValid}(\Gamma_4(r))$, then $r \in \text{local}_r$ by the contrapositive of Lemma B.77. By induction, we get $\text{repr}_{r, \text{act}}(r) \cap \text{freet}_r = \emptyset$. By $t \neq t'$, we know that $r$ cannot occur in $\text{com}_r$. Consequently, we have $\text{repr}_{r, \text{act}}(r) = \text{repr}_{r, \text{act}}(r)$. Observe that we have $\text{freet}_{r, \text{act}} \subseteq \text{freet}_r$ due to $t' \neq \bot$. Hence, we obtain $\text{repr}_{r, \text{act}}(r) \cap \text{freet}_{r, \text{act}} = \emptyset$ as required.

Let $x \in \text{dom}(\Gamma_4)$. If $x \in \text{shared}$, Lemma B.76 yields $\Gamma_4(x) = \emptyset$. So, $\text{Loc}(\Gamma_4(x)) = \text{Loc}(\emptyset)$ entails the remaining properties. Assume $x \notin \text{shared}$ hereafter. This means $x \in \text{local}_r$. Consider $a \in m_{r, \text{act}}(x)$; to be precise, we mean $a = m_{r, \text{act}}(x)$ if $x \in PVar$ and $a \in \text{repr}_{r, \text{act}}(x)$ if $x \in AVar$. Because $x$ is local to $t$, it cannot appear in $\text{com}_r$ due to the semantics. Hence, we have $a \in m_x(x)$. By induction, we have $\text{reach}_{\text{loc}}(\mathcal{H}(r)) \subseteq \text{Loc}(\Gamma_4(x))$. We establish the required $\text{reach}_{\text{loc}}(\mathcal{H}(r, \text{act})) \subseteq \text{Loc}(\Gamma_4(x))$. If $\mathcal{H}(r, \text{act}) = \mathcal{H}(r)$, then the claim follows by induction. Otherwise, $\mathcal{H}(r, \text{act})$ is of the form $\mathcal{H}(r, \text{act}) = \text{h.ev}t$ with $\mathcal{H}(r) = h$. Since $t \neq t'$, we know that $\text{ev}t$ is not an event of $t$, that is, $\text{ev}t_{1, t} = c$. By the definition of closedness under interference, we have for $S$ and all $E_L$:

$$\text{reach}_{\text{loc}}(h) \subseteq \text{Loc}(S) \implies \text{reach}_{\text{loc}}(h, \text{ev}t) \subseteq \text{Loc}(S)$$

and

$$\text{reach}_{\text{loc}}(h) \subseteq \text{Loc}(E_L) \implies \text{reach}_{\text{loc}}(h, \text{ev}t) \subseteq \text{Loc}(E_L).$$

By induction, this means that $G \in \Gamma_4(x)$ implies $\text{reach}_{\text{loc}}(h, \text{ev}t) \subseteq \text{Loc}(G)$ for all guarantees $G \in \{ S, \ldots, E_{L_4} \}$. By Lemma B.77, we have $A \notin \Gamma_4(x)$. It remains to show: $L \in \Gamma_4(x)$ implies $\text{reach}_{\text{loc}}(h, \text{ev}t) \subseteq \text{Loc}(L)$. So assume $L \in \Gamma_4(x)$. Then we have $x \in PVar$ since the type rules do not allow angels to carry $L$. By induction, we have $\text{reach}_{\text{loc}}(h) \subseteq \text{Loc}(L)$. Towards a contradiction, assume $\text{reach}_{\text{loc}}(h, \text{ev}t) \notin \text{Loc}(L)$. By definition, this means that $\text{ev}t$ makes $O_{Base}$ leave its initial location. To do that, $\text{ev}t$ must be of the form $\text{ev}t = \text{in}\text{:retire}(t', a)$. That is, $\text{com} \equiv \text{in}\text{:retire}(t', a)$ with $m_x(q) = a$. From the already established $\text{freet}_{r, \text{act}} \cap \text{retired}_{r, \text{act}} = \emptyset$, we conclude that $a \notin \text{freet}_{r, \text{act}}$. Hence, we get $a \notin \text{freet}_r$. The contrapositive of Lemma B.52 gives $q \in \text{valid}_r$. We get $\neg \text{noalias}_{r}(x)$.
because of \( x \neq q \). However, induction together with \( \llbracket L \rrbracket \in \Gamma_4(x) \) gives noalias\(_4(x)\). Since this resembles a contradiction, we must have \( \text{reach}_{t,a}^O(h,\text{evt}) \subseteq \text{Loc}(\llbracket L \rrbracket) \) as required.

Combining the above results for individual guarantees yields: \( \text{reach}_{t,a}^O(H(\tau)) \subseteq \text{Loc}(\Gamma_4(x)) \) implies \( \text{reach}_{t,a}^O(H(\tau,\text{act})) \subseteq \text{Loc}(\Gamma_4(x)) \). Hence, we get \( \text{reach}_{t,a}^O(H(\tau,\text{act})) \subseteq \text{Loc}(\Gamma_4(x)) \) by induction, as required.

\begin{itemize}
\item \textbf{Case 3:} \( t' = \bot \)
\end{itemize}

We have \( \text{com} \equiv \text{free}(a) \) or \( \text{com} \equiv \text{env}(a) \). In the latter case, we get \( m_r(p) = m_{r,\text{act}}(p) \) for all pointers \( p \in \text{PVar} \), \( \text{valid}_r = \text{valid}_{r,\text{act}} \), \( H(\tau) = H(\tau,\text{act}) \), \( \text{freed}_r = \text{freed}_{r,\text{act}} \), and \( \text{repr}_r = \text{repr}_{r,\text{act}} \). Moreover, \( m_r(\text{valid}_r) \subseteq m_{r,\text{act}}(\text{valid}_{r,\text{act}}) \) by Lemma B.47 results in \( \text{noalias}_4(p) \Rightarrow \text{noalias}_{r,\text{act}}(p) \) for all pointers \( p \in \text{PVar} \). Hence, the claim follows by induction.

Consider now \( \text{com} \equiv \text{free}(a) \). The update is \( \text{up} = \emptyset \). We have \( \text{freed}_{r,\text{act}} = \text{freed}_r \cup \{ a \} \) as well as \( \text{retired}_{r,\text{act}} = \text{retired}_r \setminus \{ a \} \). Hence, we obtain \( \text{freed}_{r,\text{act}} \cap \text{retired}_{r,\text{act}} = \emptyset \) as required.

Let \( t, \Gamma \) with \( \vdash \{ \Gamma_\text{init} \} \text{stmt}(\tau, t) \{ \Gamma \} \). By definition, we have \( \text{stmt}(\tau, \text{act}, t) = \text{stmt}(\tau, t) \).

That is, \( \vdash \{ \Gamma_\text{init} \} \text{stmt}(\tau, \text{act}, t) \{ \Gamma \} \). We show that \( \Gamma \) satisfies the claim. Let \( H(\tau) = h \). This means \( H(\tau, \text{act}) = h.\text{free}(a) \). By the semantics, we have \( h.\text{free}(a) \in S(\Omega) \).

Consider \( p \in \text{PVar} \). If \( \llbracket L \rrbracket \in \Gamma(p) \), then \( \text{noalias}_{r,\text{act}}(p) \). Since \( m_r = m_{r,\text{act}} \) and \( \text{valid}_{r,\text{act}} \subseteq \text{valid}_r \), we obtain \( \text{noalias}_{r,\text{act}}(p) \). If \( \text{isValid}(\Gamma(p)) \), then \( \{ A, L, S \} \cap \Gamma(p) \neq \emptyset \). By induction, we have \( p \in \text{valid}_r \) and \( \text{reach}_{t,m_r(p)}^O(h) \subseteq \text{Loc}(\Gamma(p)) \). Consider \( \llbracket A \rrbracket \in \Gamma(p) \) or \( \llbracket L \rrbracket \in \Gamma(p) \). By definition of the meaning of types, \( \text{reach}_{t,m_r(p)}^O(h) \subseteq \{ L_2 \} \times \text{Loc}(\Omega_{\text{SMR}}) \). Hence, \( m_r(p) \) cannot be freed according to \( \Omega_{\text{Base}} \) as otherwise it would reach its accepting location and thus contradict \( h.\text{free}(a) \in S(\Omega) \). We get \( p \in \text{valid}_{r,\text{act}} \). Consider now \( S \in \Gamma(p) \). By definition, \( \text{reach}_{t,m_r(p)}^O(h) \subseteq \text{SafeLoc}(\Omega) \). As before, this means \( m_r(p) \) cannot be freed. Altogether, we get \( p \in \text{valid}_{r,\text{act}} \) as required.

Consider \( r \in \text{AVar} \). If \( \text{isValid}(\Gamma(r)) \), then \( \text{repr}_r(r) \cap \text{freed}_r = \emptyset \) by induction. By definition, we get \( \text{repr}_{r,\text{act}}(r) \cap \text{freed}_{r,\text{act}} = \emptyset \). To arrive at \( \text{repr}_{r,\text{act}}(r) \cap \text{freed}_{r,\text{act}} = \emptyset \), it suffices to establish \( a \notin \text{repr}_r(r) \). Towards a contradiction, assume \( a \in \text{repr}_r(r) \). Then, induction gives \( \text{reach}_{t,a}^O(h) \subseteq \text{Loc}(\Gamma(r)) \). As before, however, we get \( h.\text{free}(a) \notin S(\Omega) \) from \( \text{isValid}(\Gamma(r)) \). Hence, \( a \notin \text{repr}_r(r) \) must hold as desired.

For the remaining properties, consider some \( x \in \text{dom}(\Gamma_4) \). Let \( b = m_{r,\text{act}}(x) = m_r(x) \) if \( x \in \text{PVar} \) and \( b = \text{repr}_{r,\text{act}}(x) = \text{repr}_r(x) \) if \( x \in \text{AVar} \). By the definition of \( \Omega_{\text{Base}} \) and guarantees \( A, \llbracket L \rrbracket \), we have:

\[
\text{reach}_{t,a}^O(h) \subseteq \text{Loc}(\llbracket A \rrbracket) \implies \text{reach}_{t,b}^O(h.\text{free}(a)) \subseteq \text{Loc}(\llbracket A \rrbracket) \quad \text{if } a \neq b
\]

and

\[
\text{reach}_{t,a}^O(h) \subseteq \text{Loc}(\llbracket L \rrbracket) \implies \text{reach}_{t,b}^O(h.\text{free}(a)) \subseteq \text{Loc}(\llbracket L \rrbracket) \quad \text{if } a \neq b .
\]
By the definition of interference freedom and guarantees $\mathbb{S}, \mathbb{E}_L$, we have:

$$\text{reach}^\mathcal{O}_{t,b}(h) \subseteq \text{Loc}(\mathbb{E}_L) \implies \text{reach}^\mathcal{O}_{t,b}(h.f\text{ree}(a)) \subseteq \text{Loc}(\mathbb{E}_L)$$

and $$\text{reach}^\mathcal{O}_{t,b}(h) \subseteq \text{Loc}(\mathbb{S}) \implies \text{reach}^\mathcal{O}_{t,b}(h.f\text{ree}(a)) \subseteq \text{Loc}(\mathbb{S}) .$$

Recall from before that $\{a, b, c, d\} \cap \Gamma(x) \neq \emptyset$ implies $a \neq b$. Hence, the above properties entail the desired $\text{reach}^\mathcal{O}_{t,b}(h.f\text{ree}(a)) \subseteq \text{Loc}(\Gamma(x))$ because we have $\text{reach}^\mathcal{O}_{t,b}(h) \subseteq \text{Loc}(\Gamma(x))$ by induction together with $m_r = m_{r,\text{act}}$ and $\text{repr}_r = \text{repr}_{r,\text{act}}$.

The above case distinction is complete and thus concludes the induction. 

\textbf{Proof C.73} (Corollary B.84). Let $\vdash \{ \Gamma_{\text{init}} \} \ \text{stmt}(\tau, t) \ \{ \Gamma \}$ and $\text{inv}(\mathbb{P}\mathbb{S})$. Towards a contradiction, assume the claim does not hold. That is, there is a shorted prefix $\sigma.\text{act}$ such that $\tau$ would be a pointer race. Lemma B.75 yields $\Gamma_3$ with $\vdash \{ \Gamma_{\text{init}} \} \ \text{stmt}(\sigma.\text{act}, t) \ \{ \Gamma_3 \}$. We have $\text{stmt}(\sigma.\text{act}, t) = \text{stmt}(\sigma, t) \ \text{com}$. So by the type rules there are $\Gamma_0, \Gamma_1, \Gamma_2$

$$\vdash \{ \Gamma_{\text{init}} \} \ \text{stmt}(\sigma, t) \ \{ \Gamma_0 \} \ \text{and} \ \Gamma_0 \sim \Gamma_1 \ \text{and} \ \vdash \{ \Gamma_1 \} \ \text{com} \ \{ \Gamma_2 \} \ \text{and} \ \Gamma_2 \sim \Gamma_3 .$$

We show that $\text{act}$ does not raise a pointer race.

\textbf{Case 1:} $\text{act}$ is an unsafe access

Then, $\text{com}$ contains $p.\text{next}$ or $p.\text{data}$ with $p \notin \text{valid}_\sigma$. That is, $\vdash \{ \Gamma_1 \} \ \text{com} \ \{ \Gamma_2 \}$ is derived using on of the following rules: (ASSIGN2), (ASSIGN3), (ASSIGN5), or (ASSIGN6). Since the derivation is defined, we must have $\Gamma_1(p) = T$ with $\text{isValid}(T)$. By Theorem B.83, we have $p \in \text{valid}_\sigma$. Since this contradicts the assumption of $\text{act}$ raising an unsafe access, this case cannot apply.

\textbf{Case 2:} $\text{act}$ is a racy call

Then, $\text{com} \equiv \text{in:}\text{func}(\tau)$. That is, $\vdash \{ \Gamma_1 \} \ \text{com} \ \{ \Gamma_2 \}$ is derived using Rule (ENTER). Consider $m_a(\tau) = \overline{a}$ with $\tau = r_1, \ldots, r_k, \ldots, r_n$ and $\overline{a} = \overline{a}_1, \ldots, \overline{a}_n$. Wlog. let $r_1, \ldots, r_k \in \text{PExp}$ and $r_{k+1}, \ldots, r_n \in \text{DExp}$. That $\text{act}$ is a racy call means that there are $\overline{w} = w_1, \ldots, w_n$ and $c$ with:

$$\forall i. \ (v_i = c \lor r_i \in \text{valid}_\sigma \lor r_i \in \text{DExp}) \implies v_i = w_i$$

and

$$\mathcal{F}_\mathcal{O}(h.\text{in:}\text{func}(\tau, \overline{w}), c) \notin \mathcal{F}_\mathcal{O}(h.\text{in:}\text{func}(\tau, \overline{a}), c) .$$
Let \( T_i = \Gamma(r_i) \) for \( 1 \leq i \leq k \). From Theorem B.83 we get that \( r_i \notin valid \implies \neg isValid(T_i) \). Hence:

\[
\forall i. \ (v_i = c \lor r_i \in valid \lor r \in DExp) \implies v_i = w_i
\]

implies \( \forall i. \ v_i \neq w_i \implies (v_i \neq c \land r_i \notin valid \land r_i \notin DExp) \)

implies \( \forall i. \ v_i \neq b_i \implies (v_i \neq c \land \neg isValid(T_i) \land r_i \notin DExp) \)

implies \( \forall i. \ (v_i = c \land isValid(T_i) \lor r_i \in DExp) \implies v_i = w_i \).

Moreover, Theorem B.83 gives \( \text{reach}_{\text{in}}^\omega(\sigma) \subseteq \text{Loc}(\Gamma(r_i)) \) for all \( r_i \notin DExp \). Hence, by definition, we arrive at \( \text{SafeCall}(\Gamma_i, \text{func}(\tau)) = false \). As this contradicts \( \vdash \{ \Gamma_1 \} \text{com} \{ \Gamma_2 \} \), this case cannot apply.

The above case distinction is complete. Since it yields a contradiction in each case, \( \tau \) must be PRF.

**Proof C.74** (Theorem B.85). Let \( \vdash \{ \Gamma_{\text{init}} \} P \{ \Gamma_p \} \) and \( inv(\llbracket P \rrbracket^\omega_\sigma) \). Towards a contradiction, assume the claim does not hold. That is, there is a shortest computation \( r.\text{act} \in \mathcal{O}[\llbracket P \rrbracket^\omega_\text{Adr} \) such that \( r.\text{act} \) contains moderate pointer race. By minimality, \( \tau \) is MPRF. Hence, there is some \( \sigma \in \llbracket P \rrbracket^\omega_\sigma \) with \( inv(\sigma) \implies inv(\tau) \) by Theorem B.72. Assumption \( inv(\llbracket P \rrbracket^\omega_\sigma) \) thus implies \( inv(\tau) \). Let \( act = \langle t, \text{com}, \text{up} \rangle \). If \( com \equiv \oplus inv \bullet \), then \( act \) cannot raise a pointer race and we have \( r.\text{act} \) PRF. Otherwise, we have \( inv(r.\text{act}) \) by definition and thus obtain \( r.\text{act} \) PRF by Corollary B.84. Consequently, \( r.\text{act} \) must be an unsafe assumption. That is, there is some \( \text{com}^I \in \text{next-com}(r.\text{act}) \) with \( \text{com}^I \equiv \text{ass}um e \ p = q \) and \( \{ p, q \} \notin valid_{r.\text{act}} \). By definition, this means there is \( p c \in ctr l(r.\text{act}) \) with \( pc(t^I) \xrightarrow{\text{com}^I} \bullet \) for some thread \( t^I \). By Lemma B.75, there are \( \Gamma_1, \Gamma_3 \) with \( \vdash \{ \Gamma_1 \} pc(t^I) \{ \Gamma_3 \} \). Then, Lemma B.74 yields \( \Gamma_2 \) such that \( \vdash \{ \Gamma_1 \} \text{com}^I \{ \Gamma_2 \} \). This derivation is due to Rule (\text{assume1}) or (\text{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 \( \text{com}^I \) is not an unsafe assumption. The case cannot apply. The derivation must be due to Rule (\text{assume1}). By definition of the type rule, \( \Gamma_1(p) = T \) with \( isValid(T) \) and \( \Gamma_1(q) = T^I \) with \( isValid(T^I) \). Theorem B.83, which is enabled because \( r.\text{act} \) is PRF, gives \( \{ p, q \} \subseteq valid_{\tau} \). That is, \( r.\text{act} \) is not prone to an unsafe assumption. Overall, this contradicts \( r.\text{act} \) being a moderator pointer race. Hence, we conclude that \( \mathcal{O}[\llbracket P \rrbracket^\omega_\text{Adr} \) is MPRF as required.

**Proof C.75** (Theorem B.86). Furthermore, assume \( \vdash P \) and \( inv(\llbracket P \rrbracket^\omega_\sigma) \). To the contrary, assume that the overall claim does not hold. That is, there is \( r.\text{act} \in \mathcal{O}[\llbracket P \rrbracket^\omega_\text{Adr} \) with \( act = \langle t, \text{com}, \text{up} \rangle \) such that \( com \equiv \text{in}v_{\text{retire}}(p) \) and \( m_r(p) = a \) and \( a \in \text{retired}_r \). Lemma B.75 for \( r.\text{act} \) now yields some \( \Gamma_3 \) with \( \vdash \{ \Gamma_3 \} \text{stmt}(r.\text{act}, t) \{ \Gamma_3 \} \). By definition, \( \text{stmt}(r.\text{act}, t) = \text{stmt}(\tau, t); \text{com} \). The type rules give:

\[
\vdash \{ \Gamma_3 \} \text{stmt}(\tau, t) \{ \Gamma_0 \} \quad \text{and} \quad \Gamma_0 \leadsto \Gamma_1 \quad \text{and} \quad \vdash \{ \Gamma_1 \} \text{com} \{ \Gamma_2 \} \quad \text{and} \quad \Gamma_2 \leadsto \Gamma_3
\]
for some $\Gamma_0, \Gamma_1, \Gamma_2$ where the derivation $\vdash \{ \Gamma_1 \}$ is due to Rule (enter). By definition, this means we have $A \in \Gamma_1(p)$. Note that $\tau$.act is MPRF and thus UAF by Theorem B.85. Then, Theorem B.83 yields $reach^{\mathcal{O}}_{\tau, a}(r) \subseteq Loc(\Gamma_1(p))$. In particular, $reach^{\mathcal{O}}_{\tau, a}(r) \subseteq Loc(A)$. Hence, we arrive at $(L_2, \varphi) \xrightarrow{\mathcal{H}(r)} (L_2, \varphi)$ for $\varphi = \{ z_a \mapsto a \}$. Now, Lemma B.45 gives $a \notin \text{retired}_r$. This, however, contradicts the assumed $a \in \text{retired}_r$.

**Proof C.76** (Theorem B.87). Furthermore, assume $\vdash P$ and $\text{inv}(\tau)$. To the contrary, assume the overall claim does not hold. Then, there is a shortest $\tau$.act $\in \mathcal{O}[P]_{\text{Adr}}$ with $\neg\text{inv}(\tau$.act). By minimality, we have $\text{inv}(\tau)$. By Theorem B.85, $\tau$.act is MPRF. Then, Theorem B.72 for $\tau$.act yields $\sigma \in \mathcal{O}[P]_{\tau}$ with $\text{inv}(\sigma) \implies \text{inv}(\tau$.act). The contrapositive implication and $\neg\text{inv}(\tau$.act) gives $\neg\text{inv}(\sigma)$. Hence, $\neg\text{inv}(\tau)$. 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 $r$ is PRF. Consider some thread $t$ and $x \in FVar \cup AVar$. Let $a \in m_t(x)$ and $T = \Gamma(x)$. We need to show that $reach_{\tau, a}^{\mathcal{O}}(r) \subseteq Loc(T)$ and $\text{isValid}(T) \implies x \in valid_r$. Both properties follow by Theorem B.83.

**Proof C.78** (Theorem 8.15). Set $CVar = \emptyset$ so that $\mathcal{O}[P]_{\text{Adr}}$ satisfies Assumption A.9. Theorem B.85 yields $\mathcal{O}[P]_{\text{Adr}}$ is MPRF. By $CVar = \emptyset$, this means $\mathcal{O}[P]_{\text{Adr}}$ SPRF.

**Proof C.79** (Theorem 8.16). Set $CVar = \emptyset$ so that $\mathcal{O}[P]_{\text{Adr}}$ satisfies Assumption A.9. Theorem B.86 yields $\mathcal{O}[P]_{\text{Adr}}$ 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.