University of Kaiserslautern
Department of Computer Science

Master's Thesis

Thread-Modular Reasoning for Heap-Manipulating Programs: Exploiting Pointer Race Freedom

Author:
Sebastian Wolff

submitted on:
22.10.2015

Supervisor
Prof. Dr. rer. nat. Roland Meyer

Reviewer
Dr. rer. nat. Annette Bieniusa

Abstract

Correctness of software artefacts has become a major requirement in software development processes for all sorts of systems. For parallel programs, however, testing becomes insufficient as it cannot observer all possible interleavings of threads. Hence, we employ model checking to automatically explore all behaviours of a concurrent system and thus allow proving its correctness. Today, a core problem of model checking efforts is the dynamically evolving heap in presence of low-level memory operations that stem from explicit memory management as known from C. Most approaches presently employ thread-modularity to handle an unbounded number of threads. The scalability of this methodology, however, suffers severely in the presence of explicit memory management. To overcome this limitation, a novel approach has been proposed which introduces a concept of ownership for explicitly managed memory if certain kinds of programming bugs, so-called pointer races, are absent. In the present thesis we extend an existing thread-modular analysis for concurrent data structures to integrate the novel notion of pointer race free programs. We show that relying on those programs is reasonable in the sense that even performance-critical non-blocking algorithms can be handled. Moreover, we substantiate the usefulness of this novel notion by providing experimental evidence for a speed-up of two orders of magnitude compared to classical thread-modular reasoning for explicit memory management.

Zusammenfassung

Die Korrektheit von Softwareartefakten ist immer mehr ein essentieller Bestandteil von Softwareentwicklungsprozessen geworden. Für nebenläufige Programme erweist sich das etablierte Testen gewisser Programmabläufe allerdings als unzureichend, da die Vielzahl an möglichen Ausführungsabfolgen verschiedener Threads nicht ausreichend getestet werden kann. Deshalb greifen wir auf das Model-Checking zurück, eine Technik die in der Lage ist automatisch alle möglichen Ausführungen eines parallelen Programms zu analysieren. Ein Kernproblem dabei ist die korrekte Handhabe von dynamisch alloziertem Speicher (Heap), wenn explizite Speicherverwaltung wie in C verwendet wird. Die meisten Model-Checking-Verfahren verwenden eine Thread-modulare Vorgehensweise um Korrektheit für eine beliebige Anzahl an Threads nachzuweisen. Das Problem hierbei ist, dass dieses Verfahren für explizite Speicherverwaltung schlecht skaliert. Um dem entgegen zu wirken wurde der Begriff von sogenannten Pointer Races geprägt. Es wurde gezeigt, dass Programme die keine Pointer Races beinhalten selbst unter expliziter Speicherverwaltung einem gewissen Ownerhship Prinzip folgen. In der vorliegenden Arbeit erweitern wir eine bestehende Thread-modulare Analyse für nebenläufige Datenstrukturen, sodass sie den Fakt, dass Programme als frei von Pointer Races angenommen werden, ausnutzt. Wir zeigen, dass diese Annahme begründet ist, d.h. wir zeigen dass selbst hochgradig nebenläufige Algorithmen, welche nicht-blockierende Synchronisation verwenden, mit dieser Methode behandelt werden können. Des Weiteren zeigen wir, dass die neuartige Analyse nützlich ist, da sie die Laufzeit, verglichen mit der bisherigen Thread-modularen Analyse für explizite Speicherverwaltung, um das hundertfache beschleunigt.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

This introduction provides a motivation why we are interested in verification in general. Moreover, we motivate the dedication to automated verification of concurrent data structures. Afterwards, we shortly discuss some foundations including non-blocking algorithms and their correctness criteria as well as common pitfalls that are associated with them. Then, we present a restricted programming language that is used throughout this thesis. After the discussion of related work, we continue with the contributions and the structure of the present thesis.

1.1 Motivation

In the last decades we have seen a tremendous increase of electronic devices in every imaginable aspect of our every day lives. Our society has grown dependent on software systems. Those systems no longer solely improve quality of live, but can oftentimes decide between life and death of their bearer. There are a lot of examples where software failures caused horrific disasters. Among those incidents are some with pure economical losses, like the destruction of the Ariane 5 carrier rocket in 1996 [15]. But unfortunately there are scenarios where software systems endanger people. For an example, consider the car industry. During the last decade some car manufactures decided to replace the traditional, mechanical throttle control with an electronic one. That is, the amount of fuel injected into a cars engine is controlled by a computer system reacting to the input given through the acceleration pedal. Unfortunately, this control unit reportedly failed to work properly in some cases due to software bugs. This resulted in uncontrolled acceleration and a total of ninety people rushing to death in dense traffic [4, 5, 8, 14]. Examples like this should teach us parsimonious and cautions use of embedded systems, but we still tend to deliberately accept the risks for the sake of our own convenience.

Due to the potential of software failures endangering many lives, we need to ensure that software systems operate *normally*. That is, we have to verify that those systems cannot show unexpected or unwanted behaviour. This imposes two challenges for their development. On the one hand, precise and complete specifications are needed. On the other hand, a methodology for verifying that a certain software artefact properly implements its specification is needed. Whereas the former challenge has to be solved with software engineering means, the latter one can be solved with mathematics and theoretical computer science. In the present thesis we want to focus on the theoretical part of formally showing that a given software artefact implements certain properties.

Although Turing proposed to use formal methods to argue about program correctness in 1949, it was not before the late 1960s that a new branch of computer

science evolved which is since dedicated to the formal verification of computer systems [28]. In the early stage of verification most approaches where *human driven*. That is, verification was about humans giving correctness proofs – on paper.

> "The task of [manual] proof construction can be quite tedious, and a good deal of ingenuity may be required." – E. M. Clarke and E. A. Emerson [9]

The above quote is from an early paper on automated verification by Clarke and Emerson. They proposed to use automated verification instead of manual proofs to establish correctness and correspondence to the specification. Additionally, they observed that this kind of verification is the only reasonable means for proving correctness of parallel programs. On the one hand, parallel programs are usually too complex to be handled by a *human checker*. On the other hand, testing is not sufficient as parallelism is inherently non-deterministic and certain failures cannot be reproduced on purpose but only observed rarely.

So the idea of automated verification is to apply a so-called *model check*. This checker has the sole task to compute proofs of correctness or to produce counterexamples. Although this task sounds rather simple, it is in fact not. Due to programming languages being Turing complete, there simply is no such model checker which can solve this task for arbitrary programs.

Despite this theoretical limitation, many approaches to model checking have been proposed which target restricted problem sets. In the present thesis we want to address the challenge of verifying certain kinds of programs: data structures. In fact, verifying data structures is very important as they are the basic building block of nearly every program. Due to the emerging parallelism in today's programs there is the need for those data structures to be thread-safe. For those reasons we want to dedicate this thesis to the automated verification of concurrent data structures. Our overall goal is to make the existing approaches more practical by significantly improving their performance.

1.2 Foundations

This section gives an introduction to the fundamental concepts and assumptions used throughout this thesis. Therefore, we discusses two major memory semantics and give an introduction to non-blocking data structures. Then, we present a wide-spread concept of correctness for concurrent programs: linearisability. Lastly, we discuss the ABA problem which corresponds to a class of programming bugs that are related to multi-threading.

1.2.1 Memory Management

Memory management refers to the specific operations of a system when it comes to handling dynamically allocated memory. There are two kinds of requests, allocation and deallocation, that can be made by a program. An allocation

request, also called `malloc`, must provide the caller with a portion of heap memory. The memory management has to guarantee that newly allocated memory remains available at least until deallocation is requested. A deallocation request, also called `free`, liberates the memory management from the guarantees given by the allocation. In practice, there are two major sorts of memory management systems, namely *Garbage Collection (GC)* and *Explicitly Managed Memory (MM)*.

Garbage Collection This memory management system gives the strongest guarantees for both `malloc` and `free`. It guarantees that an allocation request is always served with *fresh* memory. That is, the portion of memory returned by an allocation is not referenced by any other pointer in the program. Furthermore, deallocation requests made by the program are ignored. Instead, memory is automatically freed when it is not referenced any longer. Hence, segmentation faults due to malicious memory accesses are mostly prevented due to the memory management guarantees. Additionally, those guarantees are maintained even in the presence of multi-threading.

Explicitly Managed Memory This memory management system allows for more fine grained control over allocation and deallocation than garbage collection. In turn, the user has to ensure that allocated memory is freed in order to avoid memory leaks. An allocation request may give a *fresh* cell like in garbage collection, but could also decide to reallocated previously freed memory. Thus, there might be dangling pointers to a newly allocated cell after an allocation. Clearly, a program must avoid or detect those scenarios in order to prevent unexpected behaviour.

Lastly, a deallocation request marks some portion of the memory available for reallocation. For simplicity, however, we assume that accessing memory which has been allocated before, but freed, does not result in a segmentation fault. In programming languages like C, an access like the above has *undefined behaviour* [1]. That is, there are no guarantees that accessing freed memory will not raise a segmentation fault. A segmentation fault might occur, for example, if the runtime released the freed memory to the underlying operating system. In such a case, further accesses to this memory are prevented due to security reasons. In practice, however, the runtime avoids to release memory once allocated since reallocating freed memory is much less expensive than requesting memory from the operating system [6].

1.2.2 Non-Blocking Data Structures

Data structures are clearly on of the most important building blocks of programming. Hence, we seek efficient implementations. In a multi-threaded environment this means that there should be as little regions of mutual exclusion as possible. To that end, implementations have been proposed which even abandon the traditional locking and mutual exclusion principles [27]. Those data structures are called *non-blocking*. They usually rely on an atomic *Compare-And-Swap* (`CAS`) command, which is defined as follows.

Definition 1 (Compare-And-Swap) *The command* `CAS(&a,b,c)` *atomically compares a and b, and if equal sets a to c. It returns true if the swap was conducted, and false otherwise. For a C-like pseudo code representation, consider Algorithm 1.*

Algorithm 1 Compare-And-Swap.

```
bool CAS(T& a, T b, T c) {
   atomic {
      if (a == b) { a = c; return true; }
      else return false;
   }
}
```

There is one main advantage of relying solely on `CAS` as only atomic command. The x86 instruction set supports the `CMPXCHG` instruction which resembles exactly[1] the `CAS` from above [11]. Hence, slower software based atomics and locking mechanisms can be avoided.

A famous example of a non-blocking stack based on `CAS` is *Treiber's stack* [12]. Algorithm 2 gives an example implementation.

The underlying principle inherent to the above stack implementation and other non-blocking data structures is as follows. Instead of acquiring a lock to exclusively access the global data, a local *snapshot*, i.e. a copy, is generated. The following commands then operate on the local snapshot. In a multi-threaded environment, another thread may now alter the global data, effectively invalidating the local copy. To cope with this problem, a sanity check is conducted. Therefore, parts of the local snapshot are compared with the global data. If the check succeeds, the local snapshot is written back to the global state. The last two steps, herewith, employ the `CAS` operation to avoid interruption by another thread. This approach reduces the mutual exclusion to the very minimum of writing the global data during the `CAS` operation.

Compared to data structures which use locking, some work might be repeated as it is conducted on an out-dated local snapshot. Hence, one could assume that this has a negative effect on the performance of non-blocking algorithms. Experiments by [27], however, do not validate this intuition.

1.2.3 Linearisability

Intuitively, not every data structure implementation is correct when used in a multi-threaded environment. This is due to the fact that, at any time, a thread might be interrupted by another one – potentially harming assumptions made by the programmer. Hence, we need some notion of correctness that is reasonable even in highly concurrent data structures like the non-blocking ones described above.

[1] To ensure that the instruction is executed atomically, a `LOCK prefix` is required [11].

Algorithm 2 Treiber's stack, adapted from [27].

```
struct Node {
   data_type data;
   Node* next;
};

global Node* ToS; // top of stack

void push(data_type val) {
   Node* node, top;
   node = new Node();
   node->data = val;
   do {
      top = ToS;
      node->next = top;
   } until (CAS(ToS, top, node));
}

bool pop(data_type& dst) {
   Node* top, next;
   do {
      top = ToS;
      if (top == NULL) {
         return false;
      }
      next = ToS->next;
   } until (CAS(ToS, top, node));
   dst = top->data;
   free(top);
   return true;
}
```

The most common correctness criterion for concurrent data structures is *linearisability* [20, 21]. It provides the programmer with the illusion of functions taking effect instantaneously in an atomic action. To prove linearisability we need to instrument every function of a program with so-called *linearisation points*. Those points mark the very moment in the function execution where the effect of the function is known and takes place conceptually. When a linearisation point is reached, a so-called *linearisation event* is emitted. This event is of the form $f(in_1, \ldots, in_n, out)$ where f is the name of the called function, $in_1, \ldots, in_n$ are the actual parameters passed to the function invocation and out is the return value. Additionally, the event can be conditional. That is, reaching a linearisation point might only emit a linearisation event if some condition is met.

For an example, consider Treiber's stack from above. The first linearisation point is the `CAS` operation in the *push* function. It emits $push(val)$ whenever the Compare-And-Swap succeeds. The second linearisation point is the statement `top = ToS` at the beginning of *pop*. It conditionally emits $pop(\varnothing)$ if `ToS == NULL` holds indicating that the stack is empty. Note here that one cannot emit this

linearisation event after the later `if` although it checks the same condition. This is due to the fact that the result of the function call is defined by the value just read. That is, *pop* returns *false* regardless what other interfering threads do. In fact, when the function returns the stack might not be empty any more since another thread could have pushed some value. Nevertheless, we consider the returned value to be correct as the stack was empty at the moment when the function took place logically. The last linearisation point is, similar to the first, the `CAS` in *pop*. It emits $pop(top.data)$ whenever the `CAS` succeeds.

With an instrumented program as above, we can prove linearisability by observing the emitted linearisation events. Intuitively, a concurrent program is linearisable if every possible sequence of emitted events could have been emitted by a single threaded program. Since we assume an instrumented program, there is no need to track all possible histories, that is call and return events of functions, as done in the original definition from [21]. This leads to a simpler, more natural definition of linearisability as follows [3, 31].

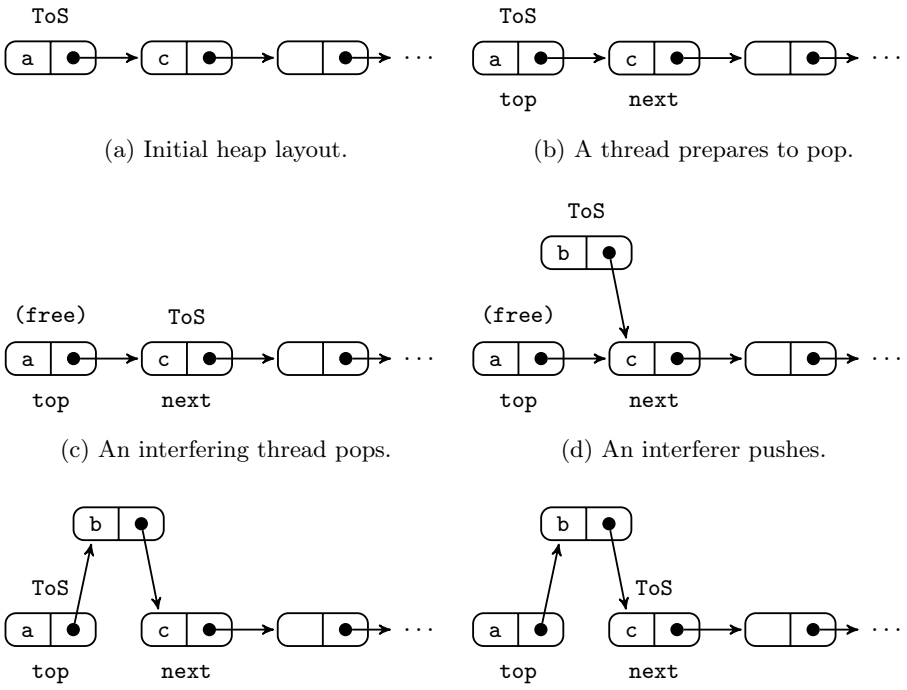
Definition 2 (Linearisability) *An instrumented program P is linearisable if $\Sigma_{P[n]} \subseteq \Sigma_{P[1]}$ holds for all $n \in \mathbb{N}$, where $\Sigma_{P[n]}$ denotes the set of all sequences of linearisation events that can be emitted by some run of P with n concurrently working threads.*

Applying this definition to Treiber's stack, we find that it is linearisable for garbage collection but not for explicit memory management. The following section discusses this problem and presents a solution making Treiber's stack linearisable in both environments.

1.2.4 The ABA Problem

For this section, recall Treiber's stack from Algorithm 2. The implementation is correct, i.e. linearisable, when executed in a garbage collected environment. For explicitly managed memory, however, this is not true. In the latter case, Treiber's stack suffers from the so-called *ABA problem*. It arises as the `CAS` operation cannot distinguish between dangling and valid pointers. That is, a `CAS` operation might succeed although it should not.

A precise description of the ABA problem in Treiber's stack is in order. Therefore, we go along an example execution the heap of which is depicted in Figure 1.1. The initial heap layout (Figure 1.1a) is as follows. The cells denoted by a and c are the topmost and second topmost cells, respectively. The global variable `ToS` points to the topmost element, i.e. a. Now, a *victim* thread prepares to pop a from the stack. Therefore, the pointers `top` and `next` are positioned to point to a and c, respectively (Figure 1.1b). But instead of performing the following `CAS`, the victim thread might be interrupted by an *interfering* thread. Assume that the interfering thread executes the following commands. It pops a (Figure 1.1c) and pushes b (Figure 1.1d). Then, another push follows. Unfortunately, the previously freed cell a is reallocated (Figure 1.1e). Since `ToS` has now come back to a, the victim thread cannot detect that its local copy has



(a) Initial heap layout.

(b) A thread prepares to pop.

(c) An interfering thread pops.

(d) An interferer pushes.

(e) An interfering push: `a` reallocated.

(f) Heap after finished pop: `b` lost.

Figure 1.1: The ABA problem in an execution of Treiber's stack.

been invalidated. Its `CAS` operation succeeds and the pointer `ToS` is redirected to cell c (Figure 1.1f). Clearly, this redirection is not desirable as it drops cell b. Moreover, it violates linearisability as the above scenario is not possible in a sequential execution.

The core problem in the above example is that the `CAS` involving `ToS` and `top` succeeds although `ToS` has been altered in between. Hence, in order to solve the ABA problem, we need to detect this manipulation and enforce that the `CAS` fails. To do so, we choose to equip the `ToS` pointer with a version counter which reflects the number of updates it has experienced [27]. Additionally, every comparison takes this counter into account and may only succeed if the versions of the compared pointers coincide. A version of Treiber's stack incorporating those counters is given in Algorithm 3.

With the above changes, we can effectively avoid the ABA problem [27]. Although we now require the Compare-And-Swap routine to compare pointers and their version counters, we can still employ hardware acceleration as described in Section 1.2.2. The x86 instruction set provides a *double width* version of the previously mentioned `CMPXCHG` instruction [11]. That is, one can compare two consecutive memory words with a single machine instruction.

Algorithm 3 Treiber's stack with version counters, adapted from [27].

```
struct VPtr {
   VNode* ptr;
   unsigned int age;
};

struct VNode {
   data_type data;
   VPtr* next;
};

global VPtr ToS; // top of stack

void init {
   ToS = NULL;
}

void push(data_type val) {
   VPtr node, top;
   node.ptr = new VNode();
   node.ptr->data = val;
   do {
      top = ToS;
      node.ptr->next = top;
   } until (dCAS(ToS, top, node));
}

bool pop(data_type& dst) {
   VPtr top, next;
   do {
      top = ToS;
      if (top.ptr == NULL) return false;
      next = ToS.ptr->next;
   } until (dCAS(ToS, top, node));
   dst = top.ptr->data;
   free(top.ptr);
   return true;
}

bool dCAS(VPtr a, VPtr b, VPtr c) {
   atomic {
      if (a.ptr == b.ptr && a.age == b.age) {
         a.ptr = c.ptr;
         a.ptr = b.ptr + 1;
         return true;
      }
      else return false;
   }
}
```

1.3 Programs

For the remainder of the present thesis we want to restrict programs in such a way that they are easy to handle but yet powerful enough to implement reasonable singly linked data structures as those described in Section 1.2. Therefore, we restrict the programs in such a way that they contain only pointer variables. Moreover, we restrict the available classes to `Node`, `VNode` and `VPtr` as used for Treiber's stack above. The formal definition of those kind of programs is given in Figure 1.2. In order to allow uniform implementations that are independent of the fact whether raw pointers or pointers with version counters are used, we employ the two type aliases `RawPtr` and `VPtr` and make use of the pointer selectors `next`, `data` and `age`. Depending on the used pointer type the selectors are syntactic sugar for the following expressions.

`RawPtr` Using a pointer variable which is defined with the type alias `RawPtr` corresponds to using raw pointers of type `Node`. For such pointers `p` the selectors correspond to `p->next` and `p->data`, respectively. Since raw pointers do not offer version counters, using the `age` selector is not allowed.

`VPtr` Using pointers with version counters corresponds to using objects of type `VPtr` and `VNode`. For those kind of special pointers `p` the selectors from above correspond to `p.ptr->next`, `p.ptr->data` and `p.age`, respectively. Moreover, we may simply use `p` in assignments instead of `p.ptr`.

Regarding data fields in our programming language, we use the type place holder `data_type`. We do not restrict the domain $\mathcal{D}$ of this type. For convenience, however, we assume that there are distinguished values $\varnothing \in \mathcal{D}$ and $\bot \in \mathcal{D}$ representing the absence of data and corrupt data, respectively. We will use the former value to represent emptiness of data structures and the latter one in cases where a value is not defined, e.g. when a variable is uninitialised.

Moreover, for some program P we use the sets Global_P and Local_P in order to refer to the global and local pointer variables of P. Hereafter, we may simply write *Global* and *Local* if clear from the context. Also note that the local variables are identical for each thread. This stems from the fact that they are defined per program and not per function. This is no restriction to our programming language but is convenient as we can uniformly refer to thread-local variables without considering certain scopes.

Another restriction is the fact that we do not have explicit return statements. Instead, we assume that non-void functions implicitly return the value written to the special data variable `_out_`. Clearly, this does not limit the expressibility of our language but simplifies its control flow.

In order to argue about certain runs of programs, it is convenient to define so-called *computations*. Intuitively, a computation is a linearisation of a program reflecting which thread executed which command. Formally, a computation τ is a sequence of actions from Act^*. An action $\mathit{act} \in \mathit{Act}$ is a pair of the form $\mathit{act} = (\mathit{tid}, \mathit{com})$ consisting of a thread identifier *tid* and a command *com*. The command reflects the executed program statement. Since we handle linearised programs, however, conditionals are reflected by assertions with the correspond-

```
Program    ::= VarDecl* Init Function*
VarDecl    ::= (global | local) Class Ptr;
Init       ::= void init() Block
Function   ::= void <function_name>(data_type _in_) Block
             | data_type <function_name>() Block
Block      ::= Stmt | { Stmt* }
Stmt       ::= PtrAssign
             | DataAssign
             | AgeAssign
             | Ptr = malloc();
             | free(Ptr);
             | if (Condition) Block
             | if (Condition) Block else Block
             | while (true) Block
             | break;
             | atomic Block
PtrAssign  ::= Ptr = Ptr;
             | Ptr.next = Ptr;
             | Ptr = Ptr.next;
DataAssign ::= Ptr.data = _in_;
             | _out_ = Ptr.data;
AgeAssign  ::= Ptr.age = Ptr.age;
             | Ptr.age++;
             | Ptr.next.age++;
Condition  ::= Ptr == Ptr
             | Ptr == NULL
             | CAS(Ptr [.next], Ptr, Ptr)
             | dCAS(Ptr [.next], Ptr, Ptr)
             | Ptr.age == Ptr.age
Ptr        ::= <pointer_variable_name>
Class      ::= <RawPtr> | <VPtr>
```

Figure 1.2: Language definition in EBNF style.

ing condition of the conditional. That is, for example, if a thread enters the else-branch of `if (c)` then the corresponding command would be `assert(!c)`. On top of those computations, we have an evaluation function h_τ which maps pointer expressions to addresses from Adr. That is, $h_\tau(x) = a$ if x points to address $a \in Adr$.

1.4 Related Work

Amit et al. [3] verify linearisability by checking whether runs of the concurrent program correspond to some sequential execution of the same program. Therefore, they alternate between exploring the state space of the concurrent program and a sequential reference execution. This technique deems a program buggy when the concurrent program outputs a value that cannot be output by any sequential run.

This approach is extended by Berdine et al. [7] in order to handle an unbounded number of threads using a thread-modular shape analysis. The core idea of this work is to use a Boolean heap abstraction as base domain and lift it to invariants universally quantified over all threads. This procedure results in a similar structure as our approach: one has to account for all possible interleavings of threads by applying sequential and interference steps.

In order to avoid the many interleavings of threads, which makes such analyses suffer from a state-space explosion and poor scalability, Gotsman et al. [17] present a method to automatically infer resource invariants. This approach allows one to neglect interference steps needed for a sound analysis following the concepts of [7]. On the downside, this approach is limited to algorithms using locks.

Another work fighting the state-space explosion problem is [30]. Segalov et al. extend the thread-modular abstraction to relate the local states of different threads. Comparing this work with [2], however, suggests that this approach works well only for loosely coupled threads. That is, the analysis of non-blocking data structures using this technique is not on par with the approach from [2].

A drawback of the previously mentioned approaches is the fact that one needs to supply the proper linearisation points. Since identifying those in complex algorithms can be rather difficult and error-prone, Liu et al. [25] present a methodology for automatically inferring linearisation points.

Another approach to verifying linearisability is taken by Vafeiadis [32] using RGSep. This combination of rely-guarantee and separation logic allows to establish correctness in the presence of an unbounded number of threads. However, it is required that the programmer annotates the code with pre and post conditions describing the atomic effects of functions. Following the separation logic approach, Vafeiadis [33] proposes an approach to automatically infer linearisation points.

The closest related work is from Abdulla et al. [2] which is used as a building block of this thesis. Compared to previous works, the main difference here is a novel abstract domain which merges certain program states. Moreover, a simpler shape analysis is used compared to the one from [7]. As we elaborate on this work in Chapter 2 we skip the details here.

Gotsman et al. [18] study implicit synchronisation patterns in non-blocking data structures for explicit memory management. They observe that such programs implement a certain protocol to prohibit data corruption. These protocols, however, are not reflected in the code but only used to argue about correctness. To make them precise Gotsman et al. introduce the notion of grace periods. A grace period allows a thread to access shared memory safely. That is, it is guaranteed that no interfering thread frees the memory in between. However, Gotsman et al. do not provide means for checking conformance to grace periods.

Besides the above mentioned grace periods, we are not aware of any ownership concept other than [19] which is able to handle fine grained memory operations and thus provides effective means to improve thread-modular reasoning for explicit memory management.

1.5 Contributions

For the present thesis we claim the following contributions. Building on the thread-modular analysis from [2], we are the first to integrate the novel notion of strong pointer races from [19] into such an analysis. This allows to exploit the novel ownership-respecting semantics from [19] and conduct a thread-modular analysis that scales for explicitly managed memory. A formal description of our verification procedure for checking linearisability of concurrent stacks and queues can be found in Chapter 4.

Our second contribution is an evaluation of the above analysis. Therefore, we implemented a prototype and ran several experiments on well-known non-blocking data structures. Our findings substantiate the usefulness of the techniques proposed in [19]. For our novel analysis, we report a speed-up of up to two orders of magnitude compared to traditional analyses relying on the full memory managed semantics. Still, due to the theory provided by [19], our analysis is sound and the results carry over to the memory managed semantics.

Lastly, we provide free access to our prototype implementation[2].

1.6 Outlook

The remainder of this thesis is structured as follows. Chapter 2 presents the building blocks of a thread-modular analysis following the overall structure of [2]. This analysis needs several techniques to overcome the challenges of proving linearisability in the presence of explicit memory management. Therefore, Wolper's data independence argument and observer automata are presented in order to specify unbounded data structures. Afterwards, the analytical aspect of the verification procedure is tackled with the thread-modular framework and shape analysis.

Chapter 3 then combines all those techniques into a verification procedure that allows to establish correctness of concurrent data structures. This chapter is devoted to formally describe a thread-modular analysis integrating the techniques from Chapter 2.

Chapter 4 discusses a semantic optimisation that is based on the novel notion of (strong) pointer races from [19]. Therefore, pointer races are formally introduced. Afterwards, the procedure from Chapter 3 is adapted exploiting the advantages of restricting the verification efforts to strong pointer race free programs.

Building upon the formalism form Chapters 3 and 4, Chapter 5 evaluates the novel analysis. First, a prototype implementation is discussed. Then, several case studies are conducted that show the usefulness of strong pointer race freedom for thread-modular reasoning. We report that the analysis from Chapter 4 provides a speed-up of up to two orders of magnitude compared to the tradi-

[2]The source code is available under: `https://github.com/Wolff09/TMRexp`.

tional thread-modular analysis for explicit memory management from Chapter 3. Moreover, it is shown that the restriction to strong pointer race free programs is reasonable and allows to handle performance-critical code.

Lastly, Chapter 6 presents possible directions of future work and concludes the thesis.

Chapter 2

Thread-Modular Verification

In the following chapter we introduce the building blocks of an analysis checking for linearisability of concurrent heap-manipulating data structures. Such an analysis has to address multiple challenges. Firstly, we have to handle a shared heap. This introduces two dimensions of infinity as the heap is unbounded on the one hand, and may store data with an unbounded domain on the other hand. Secondly, another dimension of infinity is added as we allow an arbitrary number of concurrently executing threads. Lastly, as we intend to verify data structures, the specification has to handle unboundedness, too.

We follow the overall approach of [2]. That is, we tackle the above challenges as follows. In order to handle potentially unbounded data types we employ the data independence argument from Wolper [34]. In combination with observer automata, this allows us to check linearisability of unbounded data structures. To cope with an arbitrary number of threads we employ the thread-modular framework. Furthermore, we integrate a shape analysis to handle the unbounded heap.

2.1 Data Independence

A main challenge when specifying data structures is the fact that data values can occur arbitrarily often. A stack implementation, for example, has to allow for inserting a specific value multiple times. Clearly, we will pop some value from a stack exactly as often as we pushed it. Hence, the specification needs to count the number of occurrences per data value. Counting, however, introduces infinitely many states to the specification making the verification impossible.

Towards a finite state specification, we need to eliminate the counting argument. In order to do so, we restrict our analysis to executions where every data value is input at most once. Thus, we only need to count to one. Still, our analysis remains sound. This is a consequence from Wolper's *data independence* argument [34] which we develop in the following.

Intuitively, a data independent data structure does not react on the input data. That is, the control flow of the data structure is not based on comparing data values. Obviously, this is true for stacks and queues. Formally, we argue about traces of linearisation events.

Definition 3 (Data Independence) *A set Σ of traces of linearisation events is data independent [2] if for all $\tau \in \Sigma$ and for all $f : \mathcal{D} \mapsto \mathcal{D}$ there is some $\tau' \in \Sigma$ with $f(\tau') = \tau$ and every data value in τ' occurs at most once as input.*

With this definition, we say that a data structure implementation is data independent if the set of its traces is data independent. In general, however, proving

data independence of a program is undecidable [34]. Yet, we can establish data independence for a program: it is sufficient to ensure that data values are neither manipulated nor appear in conditions [23]. Hence, we can rely on a syntactic analysis in case of a typed programming language. Moreover, we immediately find that the data structures of our interest are data independent as our restricted programming language ensures the above conditions. With these basic definitions we can now prove the following theorem [2].

Theorem 1 *Let Σ, Σ' be two data independent sets of traces. Then, $\Sigma \subseteq \Sigma'$ iff $\hat{\Sigma} \subseteq \hat{\Sigma}'$ where $\hat{\Sigma} \subseteq \Sigma$ and $\hat{\Sigma}' \subseteq \Sigma'$ contain only those traces from Σ and Σ', respectively, where every data value is input at most once.*

Proof *If $\Sigma \subseteq \Sigma'$ holds, then also $\hat{\Sigma} \subseteq \hat{\Sigma}'$ by definition of $\hat{\Sigma}$ and $\hat{\Sigma}'$. For the reverse direction consider some $\tau \in \Sigma$. Then, by data independence of Σ, there is some $\tau' \in \hat{\Sigma}$ and some $f : \mathcal{D} \to \mathcal{D}$ with $f(\tau') = \tau$. Since $\hat{\Sigma} \subseteq \hat{\Sigma}'$, we have $\tau' \in \hat{\Sigma}'$. By data independence of Σ' this gives $\tau = f(\tau') \in \Sigma'$. Hence $\Sigma \subseteq \Sigma'$.* □

Our goal is to use the above theorem to restrict our verification procedure to program executions where every data value is inserted at most once. Hence, we need to come up with a data independent specification. The following section introduces a class of finite automata, so-called observers, for this task.

2.2 Observer Automata

In order to prove correctness of a given data structure, we need to specify its intended behaviour. Recall from the introduction that we establish correctness in terms of linearisability. That is, we judge over a programs correctness by inspecting the traces of linearisation events emitted by the instrumented program. For the specification itself, we employ *observer automata* introduced by [2]. Intuitively, such automata observe a trace of linearisation events and reach an accepting state if a specification violation is detected. Formally, an observer automaton is a tuple $\mathcal{A} = (L, i, F, \mathcal{V}, T)$ where L is a finite set of observer locations, $i \in L$ is the initial location, $F \subseteq L$ is a set of final locations, $\mathcal{V} = \{z_1, \ldots, z_n\}$ is a finite set of observer variables, and T is a finite set of transitions. A transition is of the form

$$l \xrightarrow{g, evt(\overline{p})} l'$$

where g is a guard and $evt(\overline{p})$ is a linearisation event with parameters $\overline{p}$. Those parameters contain all actual parameters passed to evt and the value to be returned. For our language $\overline{p}$ consists of a single value: either the actual parameter or the return value. The guard is a conjunction of (possibly negated) equalities among observer variables from $\mathcal{V}$ and parameters from $\overline{p}$. We may abuse notation and write

$$l \xrightarrow{evt(\overline{d})} l' \quad \text{instead of} \quad l \xrightarrow{\overline{p}=\overline{d}, evt(\overline{p})} l'$$

if clear from the context.

An observer state is a pair $\langle l, \varphi \rangle$ where l is an observer location and $\varphi : \mathcal{V} \to \mathcal{D}$ is a valuation to the observer variables. An observer step is of the form

$$\langle l, \varphi \rangle \xrightarrow{evt(\overline{d})} \langle l', \varphi \rangle \quad \text{where} \quad l \xrightarrow{g, evt(\overline{p})} l'$$

is an enabled transition, i.e. $g[\overline{p} \mapsto \overline{d}, z_1 \mapsto \varphi(z_1), \ldots, z_n \mapsto \varphi(z_n)]$ evaluates to *true*. A state is initial if its location is initial and similarly final if its location is final. Note here that the valuation to the observer variables is not altered by an observer step.

In the following we want to develop observers to specify stacks and queues. In order to do so, we need to specify the intended behaviour of those data structures. Since their behaviour is commonly known we simply recall it briefly. There are three general rules that apply to both the stack and the queue specification. Additionally, we need one rule to specify the order in which elements are retrieved. In the following we use *in* and *out* to uniformly refer to the input and output functions, respectively. For a stack we have $in = push, out = pop$ and for a queue we have $in = enq, out = deq$. The specification rules are as follows [10].

(air) The data structure must not contain values out of thin air. That is, if *out* yields $d \neq \varnothing$, then there must have been a call to $in(d)$.

(loss) The data structure must not lose content. That is, if $in(d)$ has been issued, then subsequent calls to *out* must not yield $\varnothing$ until d has been retrieved.

(dupl) The data structure must not duplicate content. That is, function *out* must not return d, $d \neq \varnothing$, more often then $in(d)$ has been issued.

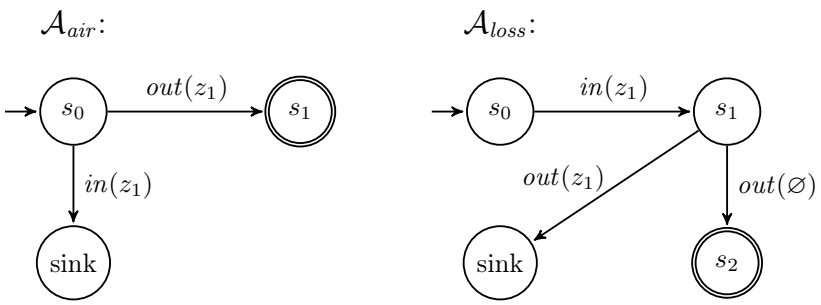
(fifo) A queue must stick to the *first-in-first-out* policy.

(lifo) A stack must stick to the *last-in-first-out* policy.

One can now easily turn the above informal description into observer automata. For every rule from above we create one observer. Hence, we come up with $\mathcal{A}_{air}, \mathcal{A}_{loss}, \mathcal{A}_{dupl}$, $\mathcal{A}_{fifo}$ and $\mathcal{A}_{lifo}$. Consider Figure 2.1 for a definition of those automata. In order to construct observer automata $\mathcal{A}_{stack}$ and $\mathcal{A}_{queue}$ specifying stacks and queues, respectively, one can combine the corresponding automata from above by computing the usual cross-product for automata.

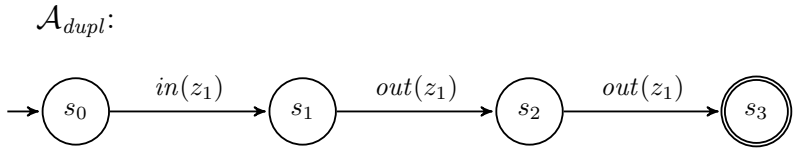
It is important to note that observer automata like the above can only observe a certain number of different data values. Namely those which coincide with the valuation of the observer variables. That is, given an observer $\mathcal{A}$ with variables $\mathcal{V} = \{z_1, \ldots, z_n\}$ and a valuation $\varphi : \mathcal{V} \to \mathcal{D}$, all values in the set $\mathcal{D}_{\overline{\mathcal{V}}} := \{\, d \mid \forall i.\ \varphi(z_i) \neq d \,\}$ trigger identical observer steps. This property is desired since we need to cope with a potentially infinite data domain.

However, we need to specify stacks and queues for all possible data values and not only for a fixed subset. To do so, we non-deterministically choose the valuation to the observer variables before observing a trace. Hence, we say that a trace τ violates the specification if there is some valuation $\varphi : \mathcal{V} \to \mathcal{D}$ such that the corresponding observer automata $\mathcal{A} \in \{\mathcal{A}_{stack}, \mathcal{A}_{queue}\}$ reaches a final state.

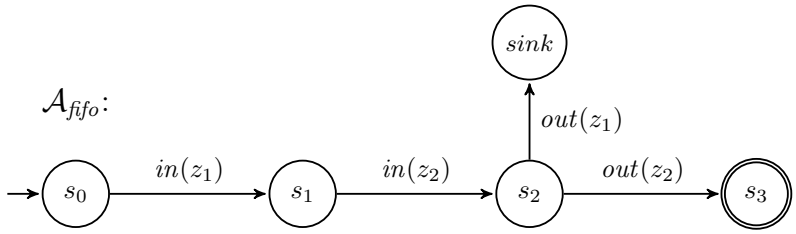


(a) Observer enforcing rule (air)

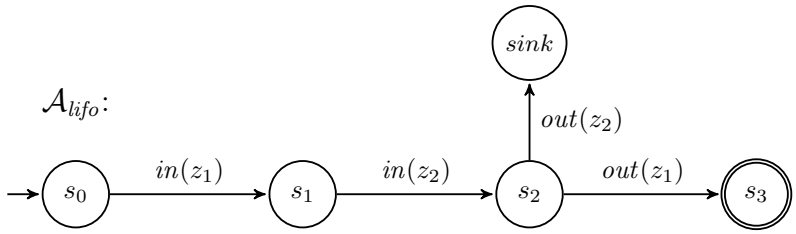
(b) Observer enforcing rule (loss)



(c) Observer enforcing rule (dupl)



(d) Observer enforcing rule (fifo)



(e) Observer enforcing rule (lifo)

Figure 2.1: Observer automata implementing the specification rules for stacks and queues, adapted from [2].

Formally, we define a violation by

$$\mathit{violation}_{\mathcal{A}}(\tau) := \exists \varphi : \mathcal{V} \to \mathcal{D}\ \exists f \in \mathit{final}(\mathcal{A}).\ \langle \mathit{init}(\mathcal{A}), \varphi \rangle \to^* \langle f, \varphi \rangle$$

where $\mathit{init}(\mathcal{A})$ is the dedicated initial state and $\mathit{final}(\mathcal{A})$ is the set of final states of $\mathcal{A}$. The specification induced by observer $\mathcal{A}$ is then the set of violation-free traces $\Sigma_{\mathcal{A}} := \{\, \tau \mid \neg \mathit{violation}_{\mathcal{A}}(\tau) \,\}$. The following theorem now states that it is sufficient to rely on observer automata for the verification of data independent programs.

Theorem 2 *For the verification of stack and queue implementations it is sufficient to*

1. *rely on observer automata for the specification, and*
2. *restrict the analysed program executions to those where every data value is input at most once.*

Proof (Sketch) *Let Σ_P be the set of all traces of linearisation events for a stack or queue implementation P. Furthermore, let Σ_S be a set of traces containing all those that coincide with correct executions of P w.r.t. the specification. Both those sets Σ_P and Σ_S are data independent. The former one is due to the definition of our programming language. The latter one is up to the specification rules of stacks and queues from above [2]. Hence, by invoking Theorem 1, we can check whether or not P is correct, i.e $\Sigma_P \subseteq \Sigma_S$ holds, by checking whether or not $\hat{\Sigma}_P \subseteq \hat{\Sigma}_S$ holds. Moreover, we have $\hat{\Sigma}_S = \Sigma_{\mathcal{A}}$ with $\mathcal{A} \in \{\mathcal{A}_{stack}, \mathcal{A}_{queue}\}$ depending on P.* □

With the above theorem and the elegant specification by observers we can continue with the analytical part of the verification procedure. That is, for a given program we need to produce the set of all necessary traces of linearisation events. As a first step towards this goal, the following section discusses thread-modular reasoning in order to explore all possible program executions.

2.3 Thread-Modular Reasoning

In the following we address the challenge of verifying programs with an arbitrary number of concurrently operating threads. The problem here is that the unboundedness of threads results in an infinite state space. To cope with this challenge, we have to move to a finite, abstract domain. Hence, we employ an abstraction technique called *thread-modular reasoning* [7, 16, 22, 26]. The core idea is to verify single threads of a program. A program state is therefore abstracted into a set of *views*. For every thread we create a view which captures the part of the program state relevant for the corresponding thread. That is, a view contains information about those parts of the heap which are reachable through the global and thread-local variables. It does not hold any information about thread identities and which views can occur together.

The thread-modular approach explores the abstract state space as follows [22]. It extends the set of thread views by *sequential* and *interference* steps until saturated. Intuitively, a sequential step modifies the view by an action of the

corresponding thread as if the program was single threaded. Interference steps account for the possible interleavings of threads. Therefore, the view of a *victim* thread is updated by an action from another *interfering* thread. This update is computed by creating a two-thread view for both victim and interferer, executing a sequential step for the interferer, and then projecting away the interfering thread. Two views may only be combined if they coincide on the shared heap. Formally, thread-modular reasoning solves the following fixed point equation

$$X_{i+1} = X_i \cup \bigcup_{v \in X_i} \mathit{post}(v) \cup \bigcup_{v,w \in X_i} \mathit{interference}(v, w)$$

where *post* and *interference* create new views from the existing ones by sequential and interference steps, respectively. We skip a formal definition of those functions here as they depend on the actual techniques integrated into the thread-modular framework. For one possible instantiation we refer to Chapter 3 below.

In the following, we want to stress a severe drawback of the thread-modular framework [19,30]. Due to the abstraction into views, the approach suffers from imprecision. For a sound analysis the interference steps have to account for all possible relations among the local heap parts of victim an interferer. This easily yields false positives deeming the verification of reasonable programs impossible. To establish correctness for Treiber's stack, for example, we need to correlate the local heaps properly. To see this, consider two threads t, t' which are performing a *push*. Assume that both threads already allocated a new node and are about to direct the nodes next field to the top of stack, i.e. the next command to be exectued is `node.next = ToS`. We hereafter write node_t and $\mathit{node}_{t'}$ to refer to the nodes of t and t', respectively. Now, an interference step has to account for the possibility that node_t and $\mathit{node}_{t'}$ coincide. Hence, t is influenced by the actions of t'. More precisely, if t' conducts the *push*, $\mathit{node}_{t'}$ becomes the new top of stack. But this also means that node_t now coincides with the top of stack. So the interference just produced a view for t where the top of stack points to the newly allocated node_t. In subsequent sequential steps, t sets $\mathit{node}_t.\mathit{next}$ to the top of stack. In fact, this redirects the top of stack to itself. Clearly, this is a false positive and does not occur in Treiber's stack [27].

To make thread-modular reasoning practical one has to prevent false positives like the above. For the garbage collected semantics, an elegant approach is to extend views with ownership information [17,31]. This information is then used to prevent erroneous relations among local parts of the heap. More precisely, if a thread owns some part of the heap, then no other thread can access it. Hence, in the above example, the interference step would not correlate node_t and $\mathit{node}_{t'}$ to coincide, effectively preventing the false positive.

For explicit memory management, however, ownership information cannot be used. Since the allocation and deallocation guarantees are not as strong as for garbage collection, one can experience dangling pointers to newly allocated memory. Hence, there is no exclusive access for the allocating thread and thread-modular reasoning has to consider the above false positive. In order to overcome this problem, works like [2, 30] fight false positives by increasing the number of threads kept in a view to two. This gives the required precision but severely limits scalability of the approach due to a quadratic blow-up of the state space [19].

With the thread-modular framework set up, we have effective means for analysing concurrent programs. In order to instantiate this framework, we need a technique for capturing the dynamic heap. The following section introduces shape analysis for this task.

2.4 Shape Analysis

With the thread-modular framework set up, we now need practical means to track the evolution of the heap. There are two challenges here. On the one hand, the heap contains data values with a potentially infinite domain. On the other hand, the heap is unbounded in its size. Hence, we have to fight two dimensions of infinity when it comes to handling a heap.

Picking up on data independence, we rigorously tackle the challenge of data values. We simply do not track data [2]. Instead, we track single cells which contain certain, fixed data values. For this set we choose the observed data values. Hence, the set of cells to track is finite since the set of observer variables is finite and every data value occurs at most once. Altogether, it is sufficient to track the structure – the shape – of the heap.

In order to capture the structure of the heap, we employ *shape analysis* [29]. We follow the novel representation from [2] abstracting the heap from cells to reachability information among pointers. That is, we track the number of steps that are needed for two pointers to reach each other following their next fields. To overcome the unboundedness of the heap, we abstract from precise reachability information. Therefore, we use four classes of reachability: *zero step*, *one step* and *multiple step* reachability, as well as *unreachable*. Formally, we use the relations $\mathcal{R} := \{=, \mapsto, \dashrightarrow, \mapsfrom, \dashleftarrow, \bowtie\}$ to relate two pointers x and y as follows [2]:

$x = y$: both pointers are identical, i.e. they refer to the very same memory cell,

$x \mapsto y$: x can reach y by following its next field, i.e. $x.next = y$,

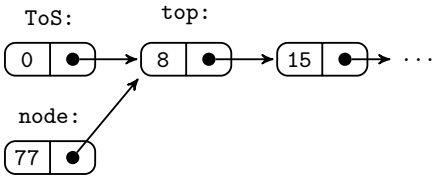
$x \dashrightarrow y$: x can reach y by following two or more next fields, i.e. $x.next \mapsto y$ or $x.next \dashrightarrow y$,

$x \mapsfrom y$: symmetric of $\mapsto$, i.e. $y \mapsto x$,

$x \dashleftarrow y$: symmetric of $\dashrightarrow$, i.e. $y \dashrightarrow x$,

$x \bowtie y$: neither of the previous, i.e. x cannot reach y and vice versa.

Lastly, we do only track a finite set of pointers $\mathbb{T}$ to overcome the unboundedness of the heap. The exact contents of this set is up to the actual incarnation of the method, like the one in Chapter 3. The relations among the pointers of $\mathbb{T}$ are tracked by a *shape matrix*.



(a) A heap setup stemming from a run of Treiber' stack.

	ToS	top	node	zero
ToS	$=$	$\mapsto$	$\bowtie$	$=$
top	$\mapsfrom$	$=$	$\mapsfrom$	$\mapsfrom$
node	$\bowtie$	$\mapsto$	$=$	$\bowtie$
zero	$=$	$\mapsto$	$\bowtie$	$=$

(b) The corresponding shape representation.

Figure 2.2: A concrete heap and its shape representation.

Definition 4 (Shape Matrix) *A shape matrix M w.r.t. $\mathbb{T}$ is a $n \times n$ matrix with $n = |\mathbb{T}|$. The rows and columns are labelled with the elements of $\mathbb{T}$. We write $M[x, y]$ to refer to the cell in the row labelled by $x \in \mathbb{T}$ and the column labelled by $y \in \mathbb{T}$. To allow for an efficient analysis, each cell contains a subset of relations from $\mathcal{R}$. This set is interpreted as a disjunction, i.e.*

$$M[x, y] = X \subseteq \mathcal{R} \iff \bigvee_{\sim \in X \subseteq \mathcal{R}} x \sim y.$$

Moreover, a shape matrix is symmetric. That is $M[y, x]$ contains the symmetric relations from $M[x, y]$.

An example is in order. Therefore consider Figure 2.2. On the left of the figure a heap is depicted which stems from a run of Treiber's stack. The topmost element, pointed to by `ToS`, contains value `0`. The second topmost element contains `8`. Additionally, a thread is executing *push* and holds an outdated pointer `top` to the cell containing `8`. The thread is pushing value `77` which is contained in the cell pointed to by `node`. On the right hand side the corresponding shape is given for $\mathbb{T} := \{\texttt{ToS}, \texttt{top}, \texttt{node}, \texttt{zero}\}$. We use `zero` to showcase how to track certain data values. Clearly, the shape does not contain any data value. But still, by the presense of `zero`, we can infer that the topmost value is `0`. We can do so since we know $\texttt{ToS} = \texttt{zero}$ from the shape and use `zero` to always coincide with the cell containing `0` (by data independence there is only one such cell). Hence, when conducting a *pop* we know that `0` is poped. For any subsequent *pop* we would not have any data binding associated with `ToS`. Thus, we say that the returned value is $\top$ which is an arbitrary member of the undistinguishable data values $\mathcal{D}_{\overline{\mathcal{V}}}$.

With shape analysis as the last building block, we are now ready to instantiate and integrate the techniques discussed so far into a verification procedure capable of automatically proving linearisability of concurrent data structures. The following chapter presents a formal description.

Chapter 3

Instantiating a Thread-Modular Analysis

The following chapter is dedicated to a formal description of a thread-modular analysis checking for linearisability of concurrent stacks and queues. We build upon the techniques and frameworks discusses in Chapter 2.

In order to track the evolution of the dynamic heap we instantiate a shape analysis with the set of tracked pointers

$$\mathbb{T}_n := \mathcal{V} \cup \mathit{Global} \cup \{\, p_i \mid p \in \mathit{Local} \ \wedge \ 1 \leq i \leq n \,\} \cup \{\mathit{NULL}\}$$

where $\mathcal{V}$ is the set of observer variables of the given specification and *NULL* represents a null pointer [2].

Our verification procedure follows the overall structure of thread-modular reasoning. Therefore, we define an n-thread *view* to be a tuple of the form

$$\mathit{view} = (\overline{pc},\ \mathit{os},\ \overline{own},\ \mathit{ages},\ \mathit{shape},\ \mathit{freed},\ \overline{in},\ \overline{out})$$

with

- control locations $\overline{pc} := pc_1, \ldots, pc_n$ for threads $1, \ldots, n$,
- an observer state os,
- disjoint sets of pointers $\overline{own} := own_1, \ldots, own_n$ for threads $1, \ldots, n$ containing pointers from $\mathbb{T}_n$ to addresses owned by the corresponding thread,
- a formula *ages* which is a conjunction of expressions of the from $x \sim y$ relating the version counters of pointers $x, y \in \mathbb{T}_n$ with $\sim \in \{<, =, >\}$,
- a shape matrix for the tracked pointers $\mathbb{T}_n$,
- a set $\mathit{freed} \subseteq \bigcup_{x \in \mathbb{T}_n} \{x, x.\mathit{next}\}$ tracking freed addresses,
- data values $\overline{in} := in_1, \ldots, in_n$ containting the actual parameter (if any) to the function currently executed by threads $1, \ldots, n$, and
- data values $\overline{out} := out_1, \ldots, out_n$ containing the return value for non-void functions exectued by threads $1, \ldots, n$.

Elements form the set *freed* are interpreted as follows. With $x \in \mathit{freed}$ we denote that the cell pointed to by x has been freed. With $x.\mathit{next} \in \mathit{freed}$ we denote that following the next field of x may eventually yield a freed cell. We capture pointers in this set since the shape analysis abstracts from cells of the heap to pointers. Moreover, we abstract from precise reachability information along the lines of shape analysis.

As discussed before, $n = 1$ is sufficient for analysing programs under garbage collection and $n = 2$ is required for explicitly managed memory.

Formally, we saturate the set of all views and solve the following fixed point equation

$$X_{i+1} := X_i \oplus \left(\bigcup_{v \in X_i} \mathit{post}(v) \cup \bigcup_{v,w \in X_i} \mathit{interference}(v,w) \right)$$

where *post* and *interference* compute a sequential and an interference step, respectively, and $\oplus$ merges two sets of configurations. The initial set X_0 contains a single view which stems from executing the initialisation function of the program.

In the following, we formally define the merge operator $\oplus$ and the functions *post* and *interference* from the above fixed point equation.

3.1 Merging Views

Previous works like [7] have show that enumerating all views is hardly possible, even when exploiting the data independence argument and shape analysis. Therefore, we rely on the idea of merging views which was introduced in [2]. Although this decreases the overall precision of the analysis, the observation is that it is still amenable to verifying lock-free stacks and queues. Moreover, [2] reports that this reduces the memory footprint of the analysis such that proving more complex algorithms like Micheal&Scott's lock free queue becomes first possible.

For the above reasons, our analysis integrates the merging of views, too. We did already give the fixed point equation above using the operator $\oplus$ to denote the merging of two sets of views. Intuitively, we merge two views if they coincide except for the shape and the ownership information. The merged view then contains a merged shape where each cell is simply a disjunction of the corresponding cells of the original shapes and an intersection of owned addresses. Formally, for two views v and w, with

$$\begin{aligned} v =& (\overline{pc}, os, \overline{own}^v, \mathit{ages}, \mathit{shape}^v, \mathit{freed}, \overline{in}, \overline{out}) \text{ and} \\ w =& (\overline{pc}, os, \overline{own}^w, \mathit{ages}, \mathit{shape}^w, \mathit{freed}, \overline{in}, \overline{out}), \end{aligned}$$

we define the merge $v \oplus w$ as

$$v \oplus w := \left\{ (\overline{pc}, os, \overline{own}^v \oplus \overline{own}^w, \mathit{ages}, \mathit{shape}^v \oplus \mathit{shape}^w, \mathit{freed}, \overline{in}, \overline{out}) \right\}$$

where the merge $\mathit{shape}^v \oplus \mathit{shape}^w$ of two shapes [2] is defined by

$$(\mathit{shape}^v \oplus \mathit{shape}^w)[x,y] := \mathit{shape}^v[x,y] \cup \mathit{shape}^w[x,y]$$

for all $x, y \in \mathbb{T}_n$, and the merged ownership information $\overline{own}^v \oplus \overline{own}^w$ is defined by

$$own_i^v \oplus own_i^w := own_i^v \cap own_i^w$$

for all $1 \leq i \leq n$. For the remaining cases where v and w differ in some of their elements besides the shape and the ownership information we apply no merging and thus have $v \oplus w := \{v, w\}$. The above operator extends naturally to sets of views and we hereafter use it in both infix and prefix notation.

In order to be able to compute *post*, one eventually needs to split shapes, i.e. undo the merging. Naturally, this needs to account for all possible splits. To prevent the generation of unnecessary false positives we prune from the set of all possible shapes those that are *inconsistent*. So given a consistency predicate *cons* we define the *split* of a shape *shp* as follows

$$\mathit{split}(\mathit{shp}) := \{\, s \mid \mathit{cons}(s) \wedge \forall x \forall y.\ s[x,y] \subseteq \mathit{shp}[x,y] \wedge |s[x,y]| = 1 \,\}$$

where $|\cdot|$ is the usual set cardinality. For a more efficient analysis, we also introduce two partial splits which do not split every cell in the shape but only a single row or a single cell. Formally, the partial splits for a shape *shp* are defined by

$$\begin{aligned}\mathit{split}(\mathit{shp}, x) := \{\, s \mid \mathit{cons}(s) \wedge \forall y.\ s[x,y] \subseteq \mathit{shp}[x,y] \wedge |s[x,y]| = 1 \\ \wedge \forall z \forall y.\ x \neq z \rightarrow s[z,y] \subseteq \mathit{shp}[z,x] \,\} \\ \mathit{split}(\mathit{shp}, x, y) := \{\, s \mid \mathit{cons}(s) \wedge s[x,y] \subseteq \mathit{shp}[x,y] \wedge |s[x,y]| = 1 \\ \wedge \forall z \forall y.\ x \neq z \rightarrow s[z,y] \subseteq \mathit{shp}[z,x] \,\}\end{aligned}$$

where the former split is restricted to the row labelled by x and the latter splits only the cell relating x and y. Note here that we cannot enforce equality among the shape cells that are not splitted since we might need to prune some relations in order to establish the overall consistency of the shape.

Lastly, we need to define the *cons* predicate. Intuitively, it should deem those shapes inconsistent which cannot occur. For example, if we have a shape *shp* with

$$\mathit{shp}[x,y] = \{\mapsto, \dashrightarrow\} \quad \text{and} \quad \mathit{shp}[x,z] = \{\mapsto, \dashrightarrow\}\,,$$

then we have to consider the splitted shape shp' with

$$\mathit{shp}'[x,y] = \{\mapsto\} \quad \text{and} \quad \mathit{shp}'[x,z] = \{\mapsto\}\,.$$

In case y and z do not coincide, for example by $\{=\} \cap \mathit{shp}[y,z] = \varnothing$, we have created an inconsistent shape. It is inconsistent as x has two distinct, direct successors. Clearly, this is not possible in our setup where pointers have a single next selector only. Formally, we define the consistency for a shape *shp* by

$$\begin{aligned}\mathit{cons}(\mathit{shp}) := \forall x, y, z\ \forall \sim_{x,z} \in \mathit{shp}[x,z] \\ \exists \sim_{x,y} \in \mathit{shp}[x,y]\ \exists \sim_{y,z} \in \mathit{shp}[y,z]. \\ \mathit{cons}(\sim_{x,z}, \sim_{x,y}, \sim_{y,z})\end{aligned}$$

where $\mathit{cons}(\cdot,\cdot,\cdot)$ decides about the consistency of a triple of relations among the corresponding pointers. For all consistent combinations consider Table 3.1.

With the possibility of merging and splitting shapes, we are now ready for the discussion of sequential steps in the following section.

3.2 Sequential Steps

Given a view v, a sequential step generates new views capturing the impact of a thread from v executing its next command. In the following we develop the

$\sim_{x,z}$	$\sim_{x,y}$	$\sim_{y,z}$
$=$	$=$	$=$
	$\mapsto$	$\mapsto$
	$\dashrightarrow$	$\dashrightarrow$
	$\mapsfrom$	$\mapsfrom$
	$\dashleftarrow$	$\dashleftarrow$
	$\bowtie$	$\bowtie$
$\mapsto$	$=$	$\mapsto$
	$\mapsto$	$=$
	$\dashrightarrow$	$\mapsfrom, \dashleftarrow$
	$\mapsfrom$	$\dashrightarrow$
	$\dashleftarrow$	$\dashrightarrow$
	$\bowtie$	$\mapsto, \dashrightarrow, \bowtie$
$\dashrightarrow$	$=$	$\dashrightarrow$
	$\mapsto$	$\mapsto, \dashrightarrow$
	$\dashrightarrow$	$=, \mapsto, \dashrightarrow, \mapsfrom, \dashleftarrow$
	$\mapsfrom$	$\dashrightarrow$
	$\dashleftarrow$	$\dashrightarrow$
	$\bowtie$	$\mapsto, \dashrightarrow, \bowtie$

$\sim_{x,z}$	$\sim_{x,y}$	$\sim_{y,z}$
$\bowtie$	$=$	$\bowtie$
	$\mapsto$	$\mapsfrom, \dashleftarrow, \bowtie$
	$\dashrightarrow$	$\mapsfrom, \dashleftarrow, \bowtie$
	$\mapsfrom$	$\bowtie$
	$\dashleftarrow$	$\bowtie$
	$\bowtie$	$=, \mapsto, \dashrightarrow, \mapsfrom, \dashleftarrow, \bowtie$
$\mapsfrom$	$=$	$\mapsfrom$
	$\mapsto$	$\dashleftarrow$
	$\dashrightarrow$	$\dashleftarrow$
	$\mapsfrom$	$=, \bowtie$
	$\dashleftarrow$	$\mapsto, \dashrightarrow, \bowtie$
	$\bowtie$	$\bowtie$
$\dashleftarrow$	$=$	$\dashleftarrow$
	$\mapsto$	$\dashleftarrow$
	$\dashrightarrow$	$\dashrightarrow, \dashleftarrow$
	$\mapsfrom$	$\mapsfrom, \dashleftarrow, \bowtie$
	$\dashleftarrow$	$=, \mapsto, \dashrightarrow, \mapsfrom, \dashleftarrow, \bowtie$
	$\bowtie$	$\bowtie$

Table 3.1: List of all consistent triples of relations.

function *post* which creates such new views. Since a view can contain multiple threads, we write $post_i$ to denote a sequential step for thread i. The overall sequential step *post* is then defined in terms of $post_i$, i.e.

$$post(v) := \bigcup_{1 \leq i \leq n} post_i(v)$$

where v is an n-thread view. In order to define $post_i$ we do a case distinction on the next command pc_i of thread i. For the sake of a lightweight presentation, we say that a result view $v' \in post_i(v)$ is given by

$$\begin{aligned} v' &= (\overline{pc}', os', \overline{own}', ages', shape', freed', \overline{in}', \overline{out}') \text{ with} \\ \overline{pc}' &= pc_1, \ldots, pc_i', \ldots, pc_n \\ \overline{own}' &= own_1, \ldots, own_i', \ldots, own_n \\ \overline{in}' &= in_1, \ldots, in_i', \ldots, in_n \\ \overline{out}' &= out_1, \ldots, out_i', \ldots, out_n \end{aligned}$$

and hereafter define pc_i', own_i', $ages'$, $shape'$, $freed'$, in_i' and out_i' depending on the corresponding command pc_i. Moreover, we may abuse notation and simply write x instead of $x_i \in \mathbb{T}_n$ for local pointers x of thread i in order to uniformly handle global and thread-i local variables. The definitions that follow are based on the concepts from [2].

Pointer Assignments Recall that there are three types of pointer assignments, namely (a) `x = y`, (b) `x = y.next`, and (c) `x.next = y`,

For type (a) of assignments we compute the sequential step by copying the information about y into x, effectively making them equal. Formally, this is done as follows.

$$\begin{aligned}
\mathit{shape}'[x,y] &:= \mathit{shape}'[x,x] := \{=\} && \\
\mathit{shape}'[x,t] &:= \mathit{shape}[y,t] && \text{for all } t \in \mathbb{T}_n \setminus \{x,y\} \\
\mathit{shape}'[t_1,t_2] &:= \mathit{shape}[t_1,t_2] && \text{for all } t_1,t_2 \in \mathbb{T}_n \setminus \{x\} \\
\mathit{ages}' &:= \mathit{ages} \wedge x = y && \\
\mathit{freed}' &:= (\,\mathit{freed} \setminus \{x, x.\mathit{next}\}\,) && \\
&\quad \cup \{x\} && \text{if } y \in \mathit{freed} \\
&\quad \cup \{x.\mathit{next}\} && \text{if } y.\mathit{next} \in \mathit{freed} \\
\mathit{in}_i' &:= \mathit{in}_i && \\
\mathit{out}_i' &:= \mathit{out}_i &&
\end{aligned}$$

Additionally, we add all those relations to ages' which follow from transitivity. We do so since otherwise changing a relation involving y later on could implicitly change the relations for x. For the ownership information, however, we have to be more careful since writing an address to a global pointer publishes that address. We account for potential ownership loss as follows

$$\begin{aligned}
\mathit{own}_i' &:= \mathit{own}_i \cup \{x\} && \text{if } y \in \mathit{own}_i \wedge x \in \mathit{Local} \\
\mathit{own}_i' &:= \mathit{own}_i \setminus \{x\} && \text{if } y \notin \mathit{own}_i \wedge x \in \mathit{Local} \\
\mathit{own}_i' &:= \mathit{own}_i \setminus (\{x\} \cup \mathit{ptrto}(y)) && \text{if } x \in \mathit{Global}
\end{aligned}$$

where $\mathit{ptrto}(y)$ yields all those pointers that reference the same cell as y, i.e. $\mathit{ptrto}(y) := \{\, t \mid \{=\} \cap \mathit{shape}[t,y] \neq \varnothing \,\}$. Lastly, we need to update the program counter and set pc_i to the next command in order.

For type (b) of assignments we need to identify a proper next field of y. A shape, however, might contain multiple choices due to merged views. Hence, we have to cover all possible choices for x that are consistent. To that end, we set

$$\begin{aligned}
\mathit{shape}' := \oplus(\{\, \mathit{shp} \mid \mathit{cons}(\mathit{shp}) \;\wedge\; & \mathit{shp}[y,x] = \{\mapsto\} \;\wedge \\
& \forall u,v \in \mathbb{T}_n \setminus \{x\}.\ \mathit{shp}[u,v] \subseteq \mathit{shape}[u,v] \,\}).
\end{aligned}$$

In addition, we have to account for the case that the cell pointed to by $y.\mathit{next}$ has been freed. To do so we defensively assume that all cells reachable from y have been freed when $y.\mathit{next} \in \mathit{freed}$ holds. That is, we set

$$\begin{aligned}
\mathit{freed}' &:= \mathit{freed} \setminus \{x, x.\mathit{next}\} && \text{if } y.\mathit{next} \notin \mathit{freed} \text{ and} \\
\mathit{freed}' &:= \mathit{freed} \cup \{x, x.\mathit{next}\} && \text{otherwise.}
\end{aligned}$$

The remaining parts of the resulting view are analogous to (a).

The last kind of assignments, type (c), is a bit more involved. In order to establish y as new successor of x we need to remove the current successors of x first. Therefore, we split the shape such that we have precise information about the current successors of x. Then, we can easily remove them. Note that removing the successors of x means that we have to remove all the reachability

information relying on them. That is, the predecessors of x cannot reach the removed successors any longer. Formally, we conduct this removal as follows:

$$\begin{aligned} \mathit{shape}' := \oplus(\{\, \mathit{shp} \mid \mathit{cons}(\mathit{shp}) \;\wedge\; \forall\, t \in \mathbb{T}_n \setminus \{x\}.\ \mathit{shp}[x,t] \cap \{\mapsto, \dashrightarrow\} = \varnothing \;\wedge \\ \exists\, s \in \mathit{split}(\mathit{shape}, x)\ \forall\, u, v \in \mathbb{T}_n \setminus \{x\}\ .\ \mathit{shp}[u,v] \subseteq s[u,v] \,\}) \end{aligned}$$

Then, we can easily connect x and y by setting $\mathit{shape}'[x,y] := \{\mapsto\}$ and adding all relations to shape' which follow from transitivity. For the set of freed pointers we have to account for the fact that x may be able to reach a freed cell if y has seen a free or can reach some freed cell. Formally, we capture this with the update

$$\begin{aligned} \mathit{freed}' :=& \mathit{freed} \cup \{x.\mathit{next}\} && \text{if } \{y, y.\mathit{next}\} \cap \mathit{freed} \neq \varnothing \text{ and} \\ \mathit{freed}' :=& \mathit{freed} \setminus \{x.\mathit{next}\} && \text{otherwise.} \end{aligned}$$

The remaining updates are again analogous to (a).

Data Assignments Recall that our restricted programming language only allows to write data values that are passed to functions or to read data values in order to return them. We have the two data assignments (a) `x.data = _in_` and (b) `_out_ = x.data`.

For the former assignment, type (a), we need to connect the input data value with pointer x. Since the shape analysis does not capture the data stored in the heap, we need other means to track it. Therefore, recall that observers can only distinguish finitely many data value classes, namely one class per observer variable and one for the remaining values. In order to track those values that can be observed, we link the pointer x with those observer variables the valuation of which coincides with the input data value. To do so, we simply reuse the sequential step for pointer assignments and execute `z = x` for all $z \in \mathcal{V}$ with $\varphi(z) = \mathit{in}_i$. If the input data value is not observed by any observer variable, we do nothing instead.

We account for the above policy when retrieving data values from pointers, i.e. when executing assignments of type (b), as follows. Either there is a data binding, i.e. there is an equality among some observer variable and the pointer variable x, or there is none. In the former case, we can get the data value for x from the valuation of the observer variable. In the latter case, we recognize that the data stems from the set of unobserved values. Formally, we set $\mathit{out}_i' = \varphi(z)$ if there is some $z \in \mathcal{V}$ with $\{=\} \subseteq \mathit{shape}[x,z]$. If the shape allows for multiple distinct data bindings, then we have to create a result view for every such binding. If no data binding is present, then we set $\mathit{out}_i' = \top$. Besides updating the program counter for the corresponding thread, nothing remains to do.

Age Assignments The age assignment `x.age = y.age` was already discussed above as part of `x = y`. We simply set $\mathit{ages}' = \mathit{ages} \wedge x = y$ and add all those relations that follow from transitivity.

In addition to the above assignment, we also support `x.age++` and `x.next.age++`. Since both updates are very similar, we discuss only the former statement.

Intuitively, to account for such an assignment, we need to update some relations in *ages* that contain x, namely those where we have $x = z$ or $x < z$ for some $z \in \mathbb{T}_n$. In the former case, we can simply replace $x = z$ with $x > z$. For the latter case, we have to account for the possibilities $x < z$ and $x = z$. The former case, $x < z$ can definitely occur. The latter case, however, might be inconsistent with the remaining formula. To see this, consider $x < z \wedge x < y \wedge y < z$ where increasing x by one cannot lead to $x = z$. In order to choose a consistent one, we can simply pick z_{min} such that

$$x < z_{min} \quad \text{and} \quad \forall y.\ x < y \rightarrow (y = z_{min} \vee z_{min} < y)$$

holds. Now, there are two resulting views for the sequential step of this assignment. For the first view we replace every $x = z$ with $x > z$ and leave all other relations untouched. For the second resulting view we additionally replace $x < z$ with $x = z$ for all z that have the same age as z_{min}, i.e. $ages \rightarrow z = z_{min}$ holds. Lastly, we extend $ages'$ with all those relations that follow form transitivity in both resulting views.

Conditionals For a conditional one has to decide with which branch to continue. Hereafter, we use pc_{true} and pc_{false} to refer to the program counters that model the branches of the conditional when its condition evaluates to *true* and *false*, respectively. There are two kinds of comparisons for conditions, namely (a) age comparisons `x.age == y.age` and (b) pointer comparisons `x == y`. For the former one we check whether the formula *ages* implies the equality in question. That is, we set $pc'_i := pc_{true}$ if $ages \rightarrow x = y$ holds and $pc'_i := pc_{false}$ otherwise.

For pointer comparisons, type (b), we consider the shape, i.e. we can check whether or not $\{=\} \subseteq shape[x, y]$. However, we have to account for merged shapes. That is, the shape could reflect two heap setups where `x == y` holds in one but not in the other. Hence, we have to split the shape and evaluate the condition using precise reachability information. Formally, we define the two sets of shapes $shps_{true}$ and $shps_{false}$ by

$$shps_{true} := \{\, shp \mid shp \in split(shape, x, y) \wedge shp[x, y] = \{=\} \,\} \text{ and}$$
$$shps_{false} := \{\, shp \mid shp \in split(shape, x, y) \wedge shp[x, y] \neq \{=\} \,\}.$$

For every non-empty of the above sets $S \in \{shps_{true}, shps_{false}\}$ we generate a new view with $shape' := \oplus(S)$ as the result of the sequential step. Besides updating the program counter to the corresponding pc_{true} and pc_{false} for $shps_{true}$ and $shps_{false}$, respectively, no further actions are required.

Free The behaviour of a `free(x)` depends on the memory semantics that is used for the analysis. For garbage collection, a free does not have an effect. The cell pointed to by x will be automatically garbage collected when there are no more pointers to it and thus the shape is not influenced and a reallocation of that cell cannot yield dangling pointers. Hence, we only need to update the program counter.

For the memory managed semantics a free marks a cell available for reallocation but does not alter its contents. Since our representation of the heap abstracts

from cells, we have to acknowledge the free for every pointer pointing to a freed cell. That is, we have to (a) mark all pointers equivalent to x as freed and (b) inform all those pointers that can reach x that they can now reach a freed cell. Again, due to merged shapes, we have to split the shape first. To avoid an unnecessary large number of result views, we merge those splitted shapes that correspond to the same reallocation scenario. Hence, we come up with the following set

$$\mathit{shps} := \bigcup_{\mathit{frd} \subseteq \mathbb{T}_n} \oplus (\{\, s \mid s \in \mathit{split}(\mathit{shape}, x) \wedge \forall\, y \in \mathbb{T}_n \setminus \{x\} \,.\, y \in \mathit{frd} \leftrightarrow s[x, y] = \{=\} \,\})$$

which contains a merged shape for every possible combination of pointer equalities among the freed cells involving x. As a result of the sequential step we generate a view for every shape $s \in \mathit{shps}$ with $\mathit{shape}' := s$ and

$$\mathit{freed}' := \mathit{freed} \cup \underbrace{\{\, y \mid s[x, y] = \{=\} \,\}}_{\text{accounts for (a)}} \cup \underbrace{\{\, y.\mathit{next} \mid s[x, y] \cap \{\mapsto, \dashrightarrow\} \neq \varnothing \,\}}_{\text{accounts for (b)}}.$$

Lastly, note that we do not need to update ownership information for the memory managed semantics since it lacks this concept.

Malloc An allocation request `x = malloc()` gives a portion of dynamically allocated memory. For garbage collection, this portion is always fresh. Hence, no pointer can reach the newly allocated memory. We account for this by setting $\mathit{shape}'[x, x] := \{=\}$ and $\mathit{shape}'[x, t] := \{\bowtie\}$ for all $t \in \mathbb{T}_n \setminus \{x\}$ and by removing x from the freed pointers, i.e. $\mathit{freed}' := \mathit{freed} \setminus \{x\}$. Additionally, ownership is claimed over the newly allocated memory and thus we set $\mathit{own}_i' := \mathit{own}_i \cup \{x\}$.

For an allocation request under explicit memory management, the above scenario applies, too. However, we do not update the ownership information since the semantics lacks this notion. Moreover, we have to account for reallocation. Again we suffer from the fact that we track pointers and not cells. Hence, we have to split up the shape in order to precisely identify all pointers that are pointing to the same cell and are reallocated for x. Similar to `free`, we define the set shps to contain a merged shape for every possible reallocation scenario as follows

$$\mathit{shps} := \bigcup_{\mathit{frd} \subseteq \mathit{freed}} \oplus (\{\, s \mid s \in \mathit{split}(\mathit{shape}, x) \wedge \forall\, y \in \mathbb{T}_n \cap \mathit{freed}.\ y \in \mathit{frd} \leftrightarrow s[x, y] = \{=\} \,\})$$

where the last condition ensures that not only a subset but all dangling pointers to the newly allocated cell are reallocated. In addition to the view representing the allocation of a fresh cell, we add a resulting view for every shape $s \in \mathit{shps}$ with

$$\mathit{shape}' := s \quad \text{and} \quad \mathit{freed}' := \mathit{freed} \setminus \{\, y \mid s[x, y] = \{=\} \,\}.$$

Regarding the definition of freed' note that a reallocation does not initialise the reallocated cells. Hence, following the next field of the new cell may yield a freed cell as before.

Linearisation Points Since our programming language does not feature an explicit construct for linearisation points, we assume that certain statements are annotated, e.g. by comments, to emit linearisation events. Whenever such a statement is handled, the resulting views of the sequential step are modified. This modification updates the observer state according to the emitted linearisation event in a straight forward way. If the observer reaches a final state, then the method aborts and reports the bug.

Returning from Functions According to our language specification, there are no explicit return statements. Instead, we implicitly return the value written to `_out_` when reaching the end of a non-void function. The return value is stored in out_i by construction. Moreover, when returning from a function, all local variables $x \in \mathit{Local}$ leave scope. We account for this as follows. Firstly, we set $\mathit{shape}'[x,t] := \{\bowtie\}$ for all $t \in \mathbb{T}_n \setminus \{x\}$. Secondly, we obtain ages' by removing from ages every conjunct containing x. Thirdly, we remove x from own_i and both $x, x.\mathit{next}$ from freed. Lastly, we set the program counter to a dedicated value indicating that there are no further commands, i.e. we set $pc_i := \bot$.

Calls For function calls we assume the most general client of the program. Hence, whenever a thread finishes the execution of a function, we have to account for all possible subsequent function calls. By construction, we recognise termination of a function if we have $pc_i = \bot$. We then create a range of result views – one per thread and actual argument. That is, for every function f that takes an argument we come up with a result view where we call $f(\mathit{val})$ for all possible values val. By data independence and the use of observers it is sufficient to consider $\mathit{val} \in \{\, d \mid d = \varphi(z) \wedge z \in \mathcal{V} \,\} \cup \{\top\}$. For functions that do not take parameters we have a single view. Each of the above views simply sets pc_i' to the first statement of the called functions. Additionally, we have to set in_i' to the corresponding value val for functions taking arguments.

Break A break statement unconditionally redirects the control flow to the next statement after the innermost loop. Hence, we only need to update the program counter and keep the remaining parts of the view.

Compare-And-Swap As discussed in Chapter 1, one can interpret a Compare-And-Swap statement as syntactic sugar. We already introduced all necessary concepts to handle such a command. Hence, we skip a formal definition and continue with the interference steps in the next section.

3.3 Interference Steps

An interference step for view v w.r.t. another view w computes how an action from a thread of w affects the threads of v. This step is called interference since the thread from w is considered to be not part of v. Intuitively, this

accounts for the scenario where the thread from v are interrupted by a thread from w. In order to capture the impact of those interferences on v we proceed as follows. We extend v with a thread from w, execute a sequential step for the new thread, and finally project away that thread. This gives an n-thread view again. Formally, we define an interference step by

$$\begin{aligned} &\mathit{interference}(v,w) := \mathit{project} \circ \mathit{post}_{n+1} \circ \mathit{extend}(v,w) && \text{if } \mathit{match}(v,w) \\ &\mathit{interference}(v,w) := \varnothing && \text{otherwise} \end{aligned}$$

where *match* guards the interference step to be only executed if w can actually interfere v, *extend* augments v with a thread from w, and *project* removes the newly added thread form all views produced by the sequential step.

In the following, we formally define the predicate *match* as well as the functions *extend* and *project* from above.

Check for Interference In order to avoid unnecessary false positives to be generated during interference steps, we guard the interference with the *match* predicate. For two views v and w, $\mathit{match}(v,w)$ evaluates to *true* only if w can interfere v. Intuitively, a thread from w can interfere v if the global states in v and w coincide. Hence, we check whether or not the interfering thread can exist in v. If so, we compute an interference step. Formally, for two n-thread views v, w with

$$\begin{aligned} v &= (\overline{pc}^v, os^v, \overline{own}^v, ages^v, shape^v, freed^v, \overline{in}^v, \overline{out}^v) \text{ and} \\ w &= (\overline{pc}^w, os^w, \overline{own}^w, ages^w, shape^w, freed^w, \overline{in}^w, \overline{out}^w), \end{aligned}$$

we define the *match* predicate by

$$\begin{aligned} \mathit{match}(v,w) := \;& pc_1^v = pc_1^w \;\wedge \cdots \wedge\; pc_{n-1}^v = pc_{n-1}^w \;\wedge \\ & os^v = os^w \;\wedge \\ & ages^v|_{\mathbb{T}_{n-1}} \leftrightarrow ages^w|_{\mathbb{T}_{n-1}} \;\wedge \\ & freed^v|_{\mathbb{T}_{n-1}} = freed^w|_{\mathbb{T}_{n-1}} \;\wedge \\ & \mathit{match}(shape^v, shape^w) \;\wedge \\ & in_1^v = in_1^w \;\wedge \cdots \wedge\; in_{n-1}^v = in_{n-1}^w \;\wedge \\ & out_1^v = out_1^w \;\wedge \cdots \wedge\; out_{n-1}^v = out_{n-1}^w \;\wedge \\ & in_n^v = in_n^w \rightarrow in_n^v \notin \top \;\wedge \\ & out_n^v = out_n^w \rightarrow out_n^v \notin \{\top, \bot\} \end{aligned}$$

where $\cdot|_{\mathbb{T}_{n-1}}$ is an restriction to $\mathbb{T}_{n-1} \subseteq \mathbb{T}_n$. More precisely, for $u \in \{v, w\}$, $ages^u|_{\mathbb{T}_{n-1}}$ is generated from $ages^u$ by removing all conjuncts that contain an expression from $\mathbb{T}_n \setminus \mathbb{T}_{n-1}$ and $freed^u|_{\mathbb{T}_{n-1}}$ is generated from $freed^u$ by removing all $x, x.next$ with $x \in \mathbb{T}_n \setminus \mathbb{T}_{n-1}$. Regarding the last two lines of the above definition, note that if the data handled by the nth thread coincides in both v and w then it is the very same thread by data independence and we prevent the

interference. For two shapes we define

$$\begin{aligned} \mathit{match}(\mathit{shape}^v, \mathit{shape}^w) := \exists\, \mathit{shp}, s, s'.\ \mathit{cons}(\mathit{shp}) \wedge \\ s \in \mathit{split}(\mathit{shape}^v) \ \wedge\ s' \in \mathit{split}(\mathit{shape}^w) \wedge \\ \forall\, x \in \mathbb{T}_{n-1}\ \forall\, y \in \mathcal{V}.\ \mathit{shp}[x,y] \subseteq s[x,y] \cup s'[x,y] \\ \forall\, x, y \in \mathbb{T}_{n-1} \backslash \mathcal{V}.\ \mathit{shp}[x,y] \subseteq s[x,y] \cap s'[x,y] \end{aligned}$$

in order to check whether or not both shapes represent a common heap structure. The last line of the above definition allows the two shapes to differ for the local variables of the nth thread. Moreover, the last line but one accounts for the fact that we cannot require the intersection of the shape cells relating observer variables to be non-empty. This is because those entries capture which pointers contain observed data values. Since they have to reflect the data of thread-local variables, too, we have to consider them to be part of the local state. For an example, consider Treiber's stack. Assume that some thread wants to push a data value observed by the observer variable $z \in \mathcal{V}$. Moreover, assume that this thread already allocated new memory and directed the next field of the local `node` pointer to the global top of stack `ToS`. This is reflected by $\texttt{node} = z \mapsto \texttt{ToS}$ in the shape. Now, we have to consider the interference with another thread that has not seen this data value before. The shape of the latter thread, however, contains only $z \bowtie \texttt{ToS}$. Hence, the intersection of the corresponding shape cells is empty. This scenario shows that we indeed require to exclude the observed values from the above condition.

It is worth noting that we always extend v with the nth thread of w. This is sufficient since the overall fixed point explores all permutations of w. Hence, we account for all possible interfering threads. Lastly, note that this is the reason why we enforce the two configurations to coincide on the threads up to thread $n-1$.

Extension For the interference step of two n-thread views v and w we need to extend v with an interfering thread from w. Along the definition of *match* we choose the nth thread from w as the interferer. For two such matching views v and w with

$$\begin{aligned} v &= (\overline{pc}^v, \mathit{os}, \overline{own}^v, \mathit{ages}^v, \mathit{shape}^v, \mathit{freed}^v, \overline{in}^v, \overline{out}^v) \text{ and} \\ w &= (\overline{pc}^w, \mathit{os}, \overline{own}^w, \mathit{ages}^w, \mathit{shape}^w, \mathit{freed}^w, \overline{in}^w, \overline{out}^w), \end{aligned}$$

we define the extended $(n+1)$-thread view as follows:

$$\begin{aligned} \mathit{extend}(v, w) := (&\overline{pc}^v,\ pc_n^w,\ \mathit{os}, \\ &\mathit{extend}(\overline{own}^v,\ \overline{own}^w), \\ &\mathit{extend}(\mathit{ages}^v,\ \mathit{ages}^w), \\ &\mathit{extend}(\mathit{shape}^v,\ \mathit{shape}^w), \\ &\mathit{extend}(\mathit{freed}^v,\ \mathit{freed}^w), \\ &\overline{in}^v,\ in^w,\ \overline{out}^v,\ out^w). \end{aligned}$$

For the extension of the age relations and the set of freed pointers, we can easily copy the relevant information from w to v. However, we additionally need to

rename the pointers of thread n from w. They still belong to the same thread, but this thread is now the $(n+1)$st thread of v. Hence we come up with

$$\begin{aligned} \mathit{extend}(\mathit{ages}^v, \mathit{ages}^w) &:= \mathit{ages}^v \wedge \mathit{rename}(\mathit{ages}^w) \text{ and} \\ \mathit{extend}(\mathit{freed}^v, \mathit{freed}^w) &:= \mathit{freed}^v \cup \mathit{rename}(\mathit{freed}^w) \end{aligned}$$

where *rename* replaces x_n with x_{n+1} for every $x \in \mathit{Local}$. Extending ownership information requires to account for merged ownership information in v and w. That is, even if both views would capture the very same state of some thread, the ownership information might differ. Hence, we proceed similar to the merging of views and define

$$\begin{aligned} \mathit{extend}(\mathit{own}^v, \mathit{own}^w) := \mathit{own}_1^v \cap \mathit{own}_1^w, \ldots, \mathit{own}_{n-1}^v \cap \mathit{own}_{n-1}^w, \\ \mathit{own}_n^v, \mathit{rename}(\mathit{own}_n^w) \end{aligned}$$

where we merge the ownership information of the first $n-1$ threads and copy the information from the nth thread of v and w. For the extended shape we have to be more careful. The problem here is due to thread-modularity: the views do not contain any information about the relations between the local variables of the nth thread from v and the nth thread from w. Hence, we have to consider all possible relations and prune those that are inconsistent. Formally, we define the extended shape w.r.t. $\mathbb{T}_{n+1}$ by

$$\begin{aligned} \mathit{extend}(\mathit{shape}^v, \mathit{shape}^w) := \oplus (\{\, \mathit{shp} \mid \mathit{cons}(\mathit{shp}) \wedge & \quad \text{(C1)} \\ \forall x, y \in \mathbb{T}_n \setminus \mathcal{V}.\ \mathit{shp}[x,y] \subseteq \mathit{shape}^v[x,y] \wedge & \quad \text{(C2)} \\ \forall x \in \mathbb{T}_n \setminus \mathcal{V}\ \forall y \in \mathcal{V}.\ \mathit{shp}[x,y] \subseteq \mathit{shape}^v[x,y] \cup \mathit{shape}^w[x,y] \wedge & \quad \text{(C3)} \\ \forall x \in \mathbb{T}_{n-1} \forall y \in \mathbb{T}_n \setminus \mathbb{T}_{n-1}.\ \mathit{shp}[x, \mathit{rename}(y)] \subseteq \mathit{shape}^w[x,y] \wedge & \quad \text{(C4)} \\ \forall x, y \in \mathbb{T}_n \setminus \mathbb{T}_{n-1}.\ \mathit{shp}[\mathit{rename}(x), \mathit{rename}(y)] \subseteq \mathit{shape}^w[x,y] \wedge & \quad \text{(C5)} \\ \forall x, y \in \mathbb{T}_n \setminus \mathbb{T}_{n-1}.\ \mathit{shp}[x, \mathit{rename}(y)] \subseteq \mathcal{R}\,\}) & \quad \text{(C6)} \end{aligned}$$

where we have to apply a renaming for the same reasons as before.

In the following, we elaborate on the above definition. Condition (C2) ensures that the shape from v is extended. Note, however, that we cannot replace the subset relation in this condition with an equality since some relations might be pruned by condition (C1). Condition (C3) is similar to the previous one, but uses a union of both shapes. This accounts for the fact that the observer variables possibly capture thread-local information as discussed before. Next, condition (C4) states that the relations involving the local pointers of the interfering thread can be copied from the shape of w. We can do so since v and w coincide for threads 1 to $n-1$. Condition (C5) simply ensures that relations among the local variables of the interferer are preserved. Lastly, condition (C6) allows all possible relations among pointers local to thread n and the interfering thread $n+1$. In combination with condition (C1) those relations are pruned such that the overall shape is consistent.

Regarding condition (C6), recall from Section 2.3 that we may want to restrict the relations based on ownership information in order to restore precision of an analysis for garbage collection with single-thread views. For a restriction we can safely remove those relations that would harm ownership. That is, if thread n

from v owns the cell pointed to by some pointer x, then no pointer from the interfering thread can reach this cell, i.e. there is no local pointer y from the interferer with $y = x \vee y \mapsto x \vee y \dashrightarrow x$. Similarly, the reverse holds for cells owned by the interferer.

Projection For an extended $(n+1)$-thread view v the projection has to undo the extension discussed above. That is, we have to remove the interfering thread from v. We do so by simply removing all information about the interferer, i.e. the $(n+1)$st thread. Formally, we define the projection of v with

$$v = (\overline{pc}, pc_{n+1}, os, \overline{own}, own_{n+1}, ages, shape, freed, \overline{in}, in_{n+1}, \overline{out}, \overline{out}_{n+1})$$

to an n-thread view by

$$project(v) = (\overline{pc}, os, \overline{own}, ages|_{\mathbb{T}_n}, shape|_{\mathbb{T}_n}, freed|_{\mathbb{T}_n}, \overline{in}, \overline{out})$$

with the restriction operator as defined above. This definition extends naturally to sets of views as required for our definition of the *interference* function.

3.4 Summary

In this chapter we have show how integrate the various techniques discussed in Chapter 2 into an analysis checking for linearisability. As discussed before, this verification approach suffers from imprecision for explicit memory management. One possible solution is to increase the number of threads kept in a view. Since this severely limits scalability, we opt for a more elaborate solution to this problem. Towards this, the following chapter discusses the novel notion of strong pointer races [19]. Based on this we are ultimately able to exploit ownership information for explicit memory management. Although this ownership is not as strong as the ownership guarantees under garbage collection, it is amenable to thread-modular reasoning with one thread per view. We will see that this approach scales well and outperforms the traditional analysis for explicit memory management by up to two orders of magnitude.

Chapter 4

Improving Thread-Modular Verification

This chapter is dedicated to improving the thread-modular analysis from above for explicit memory management. We already discussed that the method suffers from imprecision as the memory managed semantics lacks the concept of ownership. Hence, one has to increase the number of threads kept in a view [2, 30]. This gives precision but severely limits scalability.

In the following we want to discuss the notion of pointer races introduced in [19]. The core idea is to put ownership back into the memory managed semantics for pointer race free programs, i.e. for programs that avoid accessing freed memory. Although this ownership differs from the ownership guarantees given by garbage collection, it is amenable to thread-modular reasoning. With the new ownership concept we can adapt the thread-modular verification procedure from the previous chapter to use one-thread views for explicit memory management.

In the following we give an introduction to pointer races first. Then, we discuss how to integrate this notion into the verification procedure from Chapter 3. Lastly, exploiting ownership allows us to improve the interference by pruning certain steps.

4.1 Pointer Races

In order to develop the notion of pointer races introduced in [19], consider a motivating example first. Therefore, recall Treiber's stack from Chapter 1. We already discussed that this program suffers from the ABA problem under explicit memory management but not under gargabe collection. The observation form [19] is the following. The difference between those two memory semantics becomes apparent only if there are certain kinds of racy accesses: so-called pointer races. Intuitively, a pointer race is an access to a freed cell through a dangling pointer.

Formally, we define pointer races as an access to pointers that are not valid. This requires a notion of validity of pointers. Intuitively, a pointer is valid if it points to an address that is allocated. Additionally, we forbid valid pointers to point to allocated addresses by accident. That is, in order to be valid a pointer must have learned the address from the last allocation. Hence, dangling pointers are excluded. Formally, validity is defined as follows.

Definition 5 (Valid Pointer Expressions) *The valid pointer expressions in a computation τ are inductively defined by $\tau_\varepsilon := \mathbb{T} \cup \{\, a.next \mid a \in Adr \,\}$ and*

$$
\begin{aligned}
valid_{\tau.(\cdot,\texttt{x=y})} &:= valid_\tau \cup \{x\} && \text{if } y \in valid_\tau \\
valid_{\tau.(\cdot,\texttt{x=y})} &:= valid_\tau \setminus \{x\} && \text{if } y \notin valid_\tau \\
valid_{\tau.(\cdot,\texttt{x.next=y})} &:= valid_\tau \cup \{h_\tau(x).next\} && \text{if } y \in valid_\tau \\
valid_{\tau.(\cdot,\texttt{x.next=y})} &:= valid_\tau \setminus \{h_\tau(x).next\} && \text{if } y \notin valid_\tau \\
valid_{\tau.(\cdot,\texttt{x=y.next})} &:= valid_\tau \cup \{x\} && \text{if } y \in valid_\tau \wedge h_\tau(y).next \in valid_\tau \\
valid_{\tau.(\cdot,\texttt{x=y.next})} &:= valid_\tau \setminus \{x\} && \text{if } y \notin valid_\tau \vee h_\tau(y).next \notin valid_\tau \\
valid_{\tau.(\cdot,\texttt{free(x)})} &:= valid_\tau \setminus ptreq(x) \\
valid_{\tau.(\cdot,\texttt{malloc(x)})} &:= valid_\tau \cup \{x\} \\
valid_{\tau.act} &:= valid_\tau && \text{otherwise}
\end{aligned}
$$

with the set $ptreq(x)$ containing those pointer expressions that are affected by the free, i.e. $ptreq(x) := \{\, y \mid h_\tau(y) = h_\tau(x) \,\} \cup \{h_\tau(x).next\}$.

For convenience, we also define the symmetric set $invalid_\tau$ containing all those pointer expressions that are invalid in τ, i.e. $e \in invalid_\tau$ iff $e \notin valid_\tau$.

With the above definition, pointers remain valid only during the lifetime of a memory cell. That is, an allocation validates the receiving pointer but not the dangling pointers that may see the reallocation. A free then renders all pointers invalid that reference the freed cell. This ensures that dangling pointers are always invalid. With the set of invalid pointers at hand we can now formally define pointer races.

Definition 6 (Pointer Race) *A computation $\tau.act$ is called pointer race if* `x` $\in invalid_\tau$ *and the command of act contains one of the following: (a)* `x.data`*, (b)* `x.next`*, (c)* `free(x)`*, or (d) a comparison involving* `x`*. We also say that act raises a pointer race. Moreover, a set of computations is pointer race free if it does not contain a pointer race.*

The above definition basically ensures that a pointer race free program neither accesses nor compares pointers to cells that have been freed. Based on this, [19] studies the difference between the garbage collected and memory managed semantics. The main finding here is that the two semantics differ only in the presence of pointer races.[1]

Theorem 3 *If a program is free from pointer races under the garbage collected semantics, then the memory managed semantics and the garbage collected semantics coincide modulo reallocation of memory cells [19]. That is, for every computation in the memory managed semantics there is a heap-equivalent one in the garbage collectied semantics. However, for the sake of simplicity, we skip the formal details regarding the equivalence.*

[1] Hereafter, we present the overall results in form of theorems. Formal proofs can be found in the full version of [19]. We skip them here as they are not part of our contribution and would unnecessarily complicate this introduction to pointer races.

The above theorem reveals to be a very powerful tool for common programming tasks. Since the absence of pointer races is usually a desired property, one can follow a novel verification policy [19]: first, check a given program for pointer races under garbage collection. If there are such races tell the programmer about those potential bugs. Otherwise, verify the program under garbage collection. The results of the verification then carry over to the memory managed semantics. Altogether, this makes reasoning under the more complicated memory managed semantics superfluous for common programming tasks.

Unfortunately, we cannot apply the above theorem and the associated verification strategy for our purpose. The reason is that performance-critical applications intentionally make use of racy accesses – those programs are not pointer race free. For an example consider Treiber's stack. Among others, there is a pointer race in the *pop* function. When a thread executes `top = ToS` it creates a local copy of the global pointer referencing the topmost stack element. In a subsequent command, the thread executes `node = top.next` – and raises a pointer race. The problem here is that an interfering thread could have executed a pop in between resulting in `top` pointing to a freed cell. Hence, `top` is potentially invalid and accessing its next field raises a pointer race according to the definition.

In order to cope with such programs, [19] proposes the more relaxed notion of strong pointer races. The core idea is to tolerate dereferences and comparisons of invalid pointers. The underlying observation is that performance-critical data structures tend to read from invalid pointers. To shield themselves against data corruption, such accesses are followed by an integrity check. If this check fails then the value of the racy access is not used. That is, if the content of a variable stems from a racy access, then the variable is effectively dead. In Treiber's stack, this is done by the Compare-And-Swap operation following the command `node = top.next` from above. If `top` reveals to be outdated, i.e. freed in between, the value of `node` is not used. Also note that the integrity check is a comparison involving the potentially invalid pointer `top`. Hence, we have to allow both accessing and comparing invalid pointers.

Formally, the notion of strong pointer races employs a the set of strongly invalid expressions. This set captures all those expressions that result from accessing invalid expressions. In the above example, `node` becomes strongly invalid when `top` is invalid.

Definition 7 (Strongly Invalid Pointer Expressions) *The strongly invalid expressions in a computation τ are inductively defined by $\tau_\varepsilon := \varnothing$ and*

$$\begin{aligned}
sinvalid_{\tau.(\cdot,\texttt{x=y})} &:= sinvalid_\tau \cup \{x\} && \text{if } y \in sinvalid_\tau \\
sinvalid_{\tau.(\cdot,\texttt{x.next=y})} &:= sinvalid_\tau \cup \{x.next\} && \text{if } y \in sinvalid_\tau \\
sinvalid_{\tau.(\cdot,\texttt{x=y.next})} &:= sinvalid_\tau \cup \{x\} && \text{if } y \in invalid_\tau \\
sinvalid_{\tau.(\cdot,\texttt{_out_=x.data})} &:= sinvalid_\tau \cup \{_out_\} && \text{if } x \in invalid_\tau \\
sinvalid_{\tau.act} &:= sinvalid_\tau \setminus valid_\tau && \text{otherwise}
\end{aligned}$$

Note here that we may also deem data expressions strongly invalid if the value stems from accessing an invalid pointer. By the language definition, however, the only data expression that may ever be rendered strongly invalid is the special

variable `_out_` used to store the return value. It is also worth noting that the above definition relies on the assumption that memory once allocated remains accessible, i.e. it is not reclaimed by the underlying operating system (cf. Section 1.2.1). Hence, we do not need to fear dereferences of dangling pointers to result in segmentation faults. The core idea of the strongly invalid pointers is to capture those expressions a well-behaved computation should not rely on. The following definition of strong pointer races makes this notion precise.

Definition 8 (Strong Pointer Race) *A computation $\tau.act$ is a strong pointer race if the command of act corresponds to one of the following:*

(a) `x.next=y`, `x.data=_in_` *or* `free(x)` *with* `x` $\in invalid_\tau$,

(b) a comparison involving `x` *with* `x` $\in sinvalid_\tau$, *or*

(c) a statement containing `x.next` *or* `x.data` *with* `x` $\in sinvalid_\tau$.

Similarly to regular pointer races, we say that act raises a strong pointer race and that a set of computations is strong pointer race free if it does not contain a strong pointer race.

The definition of strong pointer races is, as one might expect, no drop-in replacement for regular pointer races in Theorem 3. In order to provide an analogue of this theorem, [19] defines a new memory semantics. This new semantics is based on the observation that strong pointer race free computations respect ownership. The ownership, however, does not grant the same exclusivity as ownership under garbage collection. A thread might, by accident, access memory owned by another thread through a dangling pointer. However, if they respect ownership they do not influence the owner of the memory. Phrased differently: programs that respect ownership are invariant to the fact whether memory cells are reallocated or fresh.

Definition 9 (Owned Addresses) *The set of addresses owned by a thread t in a computation τ is inductively defined by $own_\epsilon(t) := \varnothing$ and*

$$
\begin{aligned}
own_{\tau.(t,\texttt{x=malloc()})}(t) &:= own_\tau(t) \cup \{a\} && \text{if } x \in Local \text{ and } \texttt{malloc} \text{ gives } a\\
own_{\tau.(t,\texttt{free(x)})}(t) &:= own_\tau(t) \setminus \{a_x\} && \text{if } x \in valid_\tau\\
own_{\tau.(t,\texttt{x=y})}(t) &:= own_\tau(t) \setminus \{a_y\} && \text{if } x \in Global \wedge y \in valid_\tau\\
own_{\tau.(t,\texttt{x=y.next})}(t) &:= own_\tau(t) \setminus \{a_{y.next}\} && \text{if } x \in Global \wedge y, a_y.next \in valid_\tau\\
own_{\tau.(t,\texttt{x=y.next})}(t) &:= own_\tau(t) \setminus \{a_{y.next}\} && \text{if } a_y \notin own_\tau(t) \wedge y, a_y.next \in valid_\tau\\
own_{\tau.(t',\texttt{x=y.next})}(t) &:= own_\tau(t) \setminus \{a_{y.next}\} && \text{if } a_y \notin own_\tau(t) \wedge y, a_y.next \in valid_\tau\\
own_{\tau.(t,\texttt{assert x=y})}(t) &:= own_\tau(t) \setminus \{a_x, a_y\} && \text{if } x \in invalid_\tau \vee y \in invalid_\tau\\
own_{\tau.(t,\texttt{assert x}\neq\texttt{y})}(t) &:= own_\tau(t) \setminus \{a_x, a_y\} && \text{if } x \in invalid_\tau \vee y \in invalid_\tau\\
own_{\tau.act}(t) &:= own_\tau(t) && \text{otherwise}
\end{aligned}
$$

with $a_x := h_\tau(x)$, $a_y := h_\tau(y)$, $a_{y.next} := h_\tau(y.next)$ and $t' \neq t$.

Since the new memory semantics will be based on ownership and is of major interest for the verification procedure, let us elaborate a bit on the above definition in order to develop a deep understanding of ownership. Naturally, ownership is granted to freshly allocated memory and removed upon freeing it. There are,

however, more subtle cases in which ownership is removed. The first cases of losing ownership in the above definition are due to publishing. Intuitively, if a thread writes the address of an owned cell into a public pointer, then the cell is no longer owned as this corresponds to granting other threads access. In case an owned cell is published via a next field of a non-owned pointer, ownership is not removed immediately. Instead, ownership is lost when some other thread (t' in the above definition) accesses that next field[2].

Additionally, note that those cases are guarded such that ownership is only lost when the cell was published via a valid pointer. This guard prevents losing ownership by accident. Moreover, it is reasonable since passing an invalid pointer renders the receiving pointer invalid, too. Hence, no other thread can dereference and thus depend on this pointer without raising a strong pointer race.

Lastly, we also remove ownership of a cell when comparing its address with the address of an invalid pointer. The reason lies in the fact that such comparisons can break our desired invariant: a thread might find out that a certain cell was reallocated and rely on this finding in its subsequent execution.

With the above notion of ownership we can now formally define the novel ownership-respecting memory semantics introduced by [19].

Definition 10 (Ownership-Respecting Memory Semantics) *A computation $\tau.act$ is an ownership violation if the command of act is (a)* `x.next=y`*, (b)* `x.data=_in_`*, or (c)* `free(x)`*, where $h_\tau($`x`$) \in owned_\tau(t')$ and ($t = t'$ or* `x` $\in$ *Global).*

The ownership-respecting memory semantics is the set of all those computations from the memory managed semantics that are no ownership violations.

The above definition suffers from a drawback for thread-modular reasoning. As discussed in Chapter 4, we need ownership information to make the method precise enough to be practical. Recall that the described procedure did not track ownership information precisely but under-approximated the set of actually owned addresses per thread. In order to conduct a sound analysis for the new ownership-respecting semantics we additionally need to track an over-approximation of owned cells to identify all possible ownership violations. This increases the overhead of the new memory semantics compared to garbage collection. Luckily, [19] establishes the fact that ownership violations are always strong pointer races. Indeed, if a thread modifies an owned cell of another thread a strong pointer race is raised. This is the case since the modification raising an ownership violation is conducted via an invalid pointer. To see this, assume that the pointer was valid. Then the thread learned the address from the owning thread. Hence, the owner published the address and lost ownership – a contradiction to the fact that an ownership violation occurred. Using this fact it is sufficient to ensure strong pointer race freedom as this eliminates ownership violations as well. The following theorem now combines this argument with the analogue of Theorem 3 [19].

[2]The reason for this quite unnatural definition regarding next fields lies in the technical details of the proofs from [19].

Theorem 4 *If a program is strong pointer race free, then the memory managed semantics and the ownership-respecting semantics coincide. Moreover, whether or not a program is strong pointer race free can be checked under the ownership-respecting semantics.*

The above theorem allows, similarly to the one before, to abandon explicit memory management for verification. Therefore, one checks under the ownership-respecting semantics whether or not a given program is strong pointer race free. If so, an actual analysis can be conducted under the easier semantics. Otherwise, the programmer is informed about the potential bug.

In the following we adapt the thread modular-analysis from Chapter 3 to the ownership-respecting semantics.

4.2 Adapting the Verification Procedure

In the following we present an adopted version of the verification procedure from Chapter 3 to perform an analysis for the new ownership-respecting semantics. Additionally, we integrate a check for strong pointer race freedom. Rather than executing two phases, we interleave a strong pointer race check with the actual analysis. It is worth noting that the following section is a main contribution of the present thesis.

The verification procedure needs to be adapted to cope with the following needs: (a) tracking valid and strongly invalid pointers, (b) adapting the merge operator, (c) adapting the sequential step to propagate invalid and strongly invalid pointers properly, (d) adapting the sequential step to handle the new ownership model properly, (e) detect strong pointer races, and (f) adapt the interference with respect to ownership, invalid and strongly invalid pointers.

The first and most basic modification necessary is due to the need to track the invalid and strongly invalid pointer expressions. For the invalid pointer expressions we augment n-thread views with a set $inv \subseteq \mathbb{T}_n \times \mathcal{R}$. An entry $(x, \sim) \in inv$ is interpreted as follows: $(x, =)$ means x is invalid, $(x, \mapsto)$ encodes the fact that the next field of x is invalid, and $(x, \dashrightarrow)$ is used when following a chain of next fields starting at x might eventually yield an invalid pointer. For the strongly invalid pointer expressions we proceed similarly. We extend the views with a set $sin \subseteq \mathbb{T}_n \times \mathcal{R}$ the elements of which are interpreted analogously to elements from inv.

In the following, we discuss the remaining adaptions. We proceed in the same order as in the previous chapter.

4.2.1 Merging Views

Since we extended views by the sets inv and sin, we have to account for this in the merge operator $\oplus$ from Section 3.1. To that end, we say that two views v

and w can be merged only if they coincide on both *inv* and *sin*. Formally, for

$$v = (\overline{pc}, os, \overline{own}^v, ages, shape^v, freed, \overline{in}, \overline{out}, inv, sin) \text{ and}$$
$$w = (\overline{pc}, os, \overline{own}^w, ages, shape^w, freed, \overline{in}, \overline{out}, inv, sin)$$

we have

$$v \oplus w := (\overline{pc}, os, \overline{own}^v \oplus \overline{own}^w, ages, shape^v \oplus shape^w, freed, \overline{in}, \overline{out}, inv, sin)$$

where the merge of ownership information and shapes remains as before. There are no further adaptions needed for the merge operator.

4.2.2 Sequential Steps

As before we describe the working principles of the function $post_i$ applied to some n-thread view v. Therefore, we do a case distinction on the statement executed next by the ith thread. Like in Chapter 3, we assume that v and a resulting view $v' \in post_i(v)$ are of the following form:

$$v = (\overline{pc},\ os,\ \overline{own},\ ages,\ shape,\ freed,\ \overline{in},\ \overline{out},\ inv,\ sin) \text{ and}$$
$$v' = (\overline{pc}',\ os',\ \overline{own}',\ ages',\ shape',\ freed',\ \overline{in}',\ \overline{out}',\ inv',\ sin') \text{ with}$$
$$\overline{pc}' = pc_1, \ldots, pc_i', \ldots, pc_n$$
$$\overline{own}' = own_1, \ldots, own_i', \ldots, own_n$$
$$\overline{in}' = in_1, \ldots, in_i', \ldots, in_n$$
$$\overline{out}' = out_1, \ldots, out_i', \ldots, out_n$$

For brevity, we only remark on those parts of the sequential step that need to be adapted. In particular, we may only describe how to update the newly added components *inv* and *sin*. For the ownership information we proceed as in the case of garbage collection if not stated otherwise.

Pointer Assignments For pointer assignments we have to properly track the sets *inv* and *sin*. In addition, we have to account for the new notion of ownership according to Definition 9. Moreover, we need to check for strong pointer races.

For an assignment of the form `x = y`, we proceed as follows. We copy the information about y over to x, i.e. we set

$$inv' := inv \cup \{\, (x, \sim) \mid (y, \sim) \in inv \,\} \text{ and}$$
$$sin' := sin \cup \{\, (x, \sim) \mid (y, \sim) \in sin \,\}.$$

Updating the ownership information is similar to the procedure as before. We only need to account for the fact that ownership is only lost if the address was published via a valid pointer. Hence, we have to add the side condition $(y, =) \notin inv$ and come up with the following

$$own_i' := own_i \cup \{x\} \qquad \text{if } y \in own_i \wedge x \in Local$$
$$own_i' := own_i \setminus \{x\} \qquad \text{if } y \notin own_i \wedge x \in Local$$
$$own_i' := own_i \setminus (\{x\} \cup ptrto(y)) \qquad \text{if } x \in Global \wedge (y, =) \notin inv$$

where $ptrto(y)$ yields the pointers equivalent y. Lastly, note that this statement does not raise a strong pointer race.

For assignments `x.next = y` we proceed similarly to the above. The first step, however, is to check for strong pointer races. The statement is racy if we have $(x, \mapsto) \in (inv \cup sin)$. If a strong pointer race is detected we abort the procedure and report the bug to the programmer. Otherwise we continue analogously to the previous pointer assignment.

The remaining type of pointer assignments, `x = y.next`, is in order. Again, we check for strong pointer races first. The assignment is deemed racy and the method aborts if $(y, \mapsto) \in sin$. Otherwise, we update the set inv and the ownership information as above. The update of sin, however, needs a bit more care since x might become strongly invalid depending on the validity of y. Hence, we set

$$sin' := (sin \setminus \{ (x, \sim) \mid \sim \in \mathcal{R} \}) \cup \{ (x, \sim) \mid (y, \sim) \in (inv \cup sin) \}.$$

Data Assignments Consider the data assignment `_out_ = x.data`. A strong pointer race is raised if $(x, =) \in sin$ holds. For the updates, we only need to revisit the case when `_out_` becomes strongly invalid. By definition, this occurs if x is invalid. In order to track the invalidity of the return value we set $out_i' := \bot$. The sets inv and sin as well the ownership information remain unchanged.

For a data assignment of the form `x.data = _in_` we only need to add a strong pointer race check. This is done by checking whether or not $(x, =) \in (inv \cup sin)$ holds.

Conditionals Conditionals containing a comparison of the form `x == y` need only little changes. First, we add a strong pointer race check. In order to continue with the analysis we have to ensure that neither x nor y is strongly invalid, i.e. $(x, =) \notin sin \wedge (y, =) \notin sin$ holds. It only remains to update ownership information. We have to account for the case that ownership is lost by comparing a pointer to an owned address with an invalid pointer. That is, if $x \in own_i \wedge (y, =) \in inv$ holds we have to remove the address pointed to by x from the set of owned addresses. Hence, we need to remove all pointers that are equivalent to x from own_i. We already gave a description on how to do this for pointer assignments above. The symmetric case where ownership of y is lost is analogous.

Lastly, it is worth noting that age comparisons are not restricted in the ownership-respecting semantics. That is, they do not raise strong pointer races. This fact is crucial since non-blocking data structures prevent data corruption by comparing age fields of potentially invalid pointers. If those fields would be considered strongly invalid like their data field counterparts, then we would suffer from strong pointer races in performance-critical applications. Hence, the relaxed handling of age field comparisons in the ownership-respecting semantics is a desired property.

Free For a `free(x)` we have to invalidate all pointers that are pointing to the freed cell. But first we ensure that no strong pointer race is raised. Hence, we abort if $(x, =) \in \mathit{inv}$. In order to prevent false positives and make the approach practical we have to restrict the above abort to those cases where x has an established data binding, i.e. its data value is observed. We do so since interference steps potentially establish equality among pointers erroneously. For observed data values we can prevent those scenarios using the data independence argument (cf. definition of the *match* predicate from Section 3.3). For unobserved values, however, we cannot and thus a cell might erroneously be freed multiple times. Since this only occurs for unobserved data values we relax the strong pointer race check in this case. This does not harm soundness since the observed data values are implicitly universally quantified.

For the updates we extend the corresponding sequential step from the memory managed analysis. Recall from Chapter 3 that this step results in set V of views representing the various reallocation scenarios. For every result view $v' \in V$ we update the set *inv* by adding all those pointers that are affected by the free. Therefore, we set

$$\mathit{inv}' := \mathit{inv} \cup \left\{ (x, \sim) \mid x \in (\mathit{freed}' \setminus \mathit{freed}) \cap \mathbb{T}_n \wedge \sim \in \{=, \mapsto\} \right\}$$

where $\mathit{freed}' \setminus \mathit{freed}$ exactly captures those pointers that are affected. Additionally, ownership of the freed cell is lost. To account for this we have to set $\mathit{own}_i' := \mathit{own}_i \setminus (\mathit{freed}' \setminus \mathit{freed})$. Note that the freed cell cannot be owned by another thread since this would have raised a strong pointer race: by construction x can only point to a cell owned by another thread if it is invalid. Lastly, note that the set *sin* is not affected by the free.

Malloc For the allocation of dynamic memory we proceed as for the memory managed semantics before. In addition, we have to track ownership information as well as invalid and strongly invalid pointers. To that end, we extend every resulting view $v' \in \mathit{post}_i(v)$ from the sequential step for the memory managed semantics. Since the allocation `x=malloc()` renders the receiving pointer valid, we have to remove it from both the set of invalid and the set of strongly invalid pointers. Hence, we set $\mathit{inv}' := \mathit{inv} \setminus \{(x, =)\}$ and $\mathit{sin}' := \mathit{sin} \setminus \{(x, =)\}$. Note that an allocation validates neither dangling pointers nor the next field of x. Lastly, a thread owns those addresses that it allocates using local variables. To account for this we set $\mathit{own}_i' := \mathit{own} \cup \{x\}$ if $x \in \mathit{Local}$ and $\mathit{own}_i' := \mathit{own}_i \setminus \{x\}$ otherwise.

Returning from Functions When returning from functions we have to extend the clean up step from before. We additionally need to remove the local variables from the sets *inv* and *sin*. To do so we set

$$\mathit{inv}' := \mathit{inv} \setminus L \qquad \text{and} \qquad \mathit{sin}' := \mathit{sin} \setminus L$$

where $L := \{ (x, \sim) \mid x \in \mathit{Local} \wedge \sim \in \mathcal{R} \}$ is the set of tuples from $\mathbb{T}_n \times \mathcal{R}$ which contain a pointer variable that goes out of scope. For the ownership information there is nothing to adapt as it is already handled properly in Section 3.2.

4.2.3 Interference Steps

For the interference step of the adapted verification procedure we defined the functions $match_{own}$, $extend_{own}$ and $project_{own}$ to replace *match*, *extend* and *project* from Section 3.3, respectively.

Check for Interference During the interference we have to account for the sets *inv* and *sin* that have been added to views. For the check whether a view v can be interfered by another view w we have to extend the procedure from Section 3.3. For two views v, w with

$$v = (\overline{pc}^v, os^v, \overline{own}^v, ages^v, shape^v, freed^v, \overline{in}^v, \overline{out}^v, inv^v, sin^v) \text{ and}$$
$$w = (\overline{pc}^w, os^w, \overline{own}^w, ages^w, shape^w, freed^w, \overline{in}^w, \overline{out}^w, inv^w, sin^w)$$

we define $match_{own}$ by

$$\begin{aligned} match_{own}(v, w) := \ & match(\hat{v}, \hat{w}) \wedge inv^v|_{\mathbb{T}_{n-1}} = inv^w|_{\mathbb{T}_{n-1}} \\ & \wedge sin^v|_{\mathbb{T}_{n-1}} = sin^w|_{\mathbb{T}_{n-1}} \end{aligned}$$

where we reuse the definition of *match* from before and write $\hat{v}, \hat{w}$ for

$$\hat{v} = (\overline{pc}^v, os^v, \overline{own}^v, ages^v, shape^v, freed^v, \overline{in}^v, \overline{out}^v) \text{ and}$$
$$\hat{w} = (\overline{pc}^w, os^w, \overline{own}^w, ages^w, shape^w, freed^w, \overline{in}^w, \overline{out}^w).$$

Extension Extending a view with an interfering thread is similar to the procedure from before. For two matching views v and w with

$$v = (\overline{pc}^v, os, \overline{own}^v, ages^v, shape^v, freed^v, \overline{in}^v, \overline{out}^v, inv^v, sin^v) \text{ and}$$
$$w = (\overline{pc}^w, os, \overline{own}^w, ages^w, shape^w, freed^w, \overline{in}^w, \overline{out}^w, inv^w, sin^w),$$

we define the extended $(n+1)$-thread view as follows:

$$\begin{aligned} extend_{own}(v, w) := (& \overline{pc}^v,\ pc_n^w,\ os, \\ & extend(\overline{own}^v,\ \overline{own}^w), \\ & extend(ages^v,\ ages^w), \\ & extend(shape^v,\ shape^w), \\ & extend(freed^v,\ freed^w), \\ & \overline{in}^v,\ in^w,\ \overline{out}^v,\ out^w \\ & extend(inv^v, inv^w) \\ & extend(sin^v, sin^w)) \end{aligned}$$

where this definition differs from the one from Section 3.3 only in the last two lines. The extension of *inv* and *sin* is conducted analogously to the extension of *freed*. Hence, we come up with

$$extend(inv^v, inv^w) := inv^v \cup rename(inv^w) \text{ and}$$
$$extend(sin^v, sin^w) := sin^v \cup rename(sin^w)$$

where we again rename the thread-local pointers of the interfering thread.

In addition to the above extension, we have to carefully revisit the shape extension since the relations among local variables are restricted by the introduced ownership. Intuitively, when a thread owns the cell pointed to by its local pointer variable x, then any other pointer y not local to that thread is either pointing to another cell or (strongly) invalid. Formally, we define the shape extension by

$$
\begin{aligned}
&\mathit{extend}(\mathit{shape}^v, \mathit{shape}^w) := \oplus\,(\{\, \mathit{shp} \mid \mathit{cons}(\mathit{shp}) \wedge && \text{(C1)}\\
&\qquad \forall x, y \in \mathbb{T}_n \setminus \mathcal{V}.\ \mathit{shp}[x,y] \subseteq \mathit{shape}^v[x,y] \wedge && \text{(C2)}\\
&\qquad \forall x \in \mathbb{T}_n \setminus \mathcal{V}\ \forall y \in \mathcal{V}.\ \mathit{shp}[x,y] \subseteq \mathit{shape}^v[x,y] \cup \mathit{shape}^w[x,y] \wedge && \text{(C3)}\\
&\qquad \forall x \in \mathbb{T}_{n-1} \forall y \in \mathbb{T}_n \setminus \mathbb{T}_{n-1}.\ \mathit{shp}[x, \mathit{rename}(y)] \subseteq \mathit{shape}^w[x,y] \wedge && \text{(C4)}\\
&\qquad \forall x, y \in \mathbb{T}_n \setminus \mathbb{T}_{n-1}.\ \mathit{shp}[\mathit{rename}(x), \mathit{rename}(y)] \subseteq \mathit{shape}^w[x,y] \wedge && \text{(C5)}\\
&\qquad \forall x, y \in \mathbb{T}_n \setminus \mathbb{T}_{n-1}.\ \mathit{shp}[x, \mathit{rename}(y)] \subseteq \mathcal{R}(x,y)\,\}) && \text{(C6n)}
\end{aligned}
$$

where we only modified the last line compared to the definition from Section 3.3. Instead of allowing all possible relations, we restrict the choice in (C6n) according to the above intuition. Hence, we define $\mathcal{R}(x,y) \subseteq \mathcal{R}$ to be the largest set satisfying

$$
\begin{aligned}
\{=\} \subseteq \mathcal{R}(x,y) \quad &\Rightarrow \quad (y,=) \in (\mathit{inv}^w \cup \mathit{sin}^w)\\
\{\mapsto\!\dashv\} \subseteq \mathcal{R}(x,y) \quad &\Rightarrow \quad (y,\mapsto) \in (\mathit{inv}^w \cup \mathit{sin}^w)\\
\{\dashv\!\!\dashrightarrow\} \subseteq \mathcal{R}(x,y) \quad &\Rightarrow \quad (y,\dashrightarrow) \in (\mathit{inv}^w \cup \mathit{sin}^w)
\end{aligned}
$$

if $x \in \mathit{own}_n^v$ and the analogue conditions

$$
\begin{aligned}
\{=\} \subseteq \mathcal{R}(x,y) \quad &\Rightarrow \quad (x,=) \in (\mathit{inv}^v \cup \mathit{sin}^v)\\
\{\mapsto\} \subseteq \mathcal{R}(x,y) \quad &\Rightarrow \quad (x,\mapsto) \in (\mathit{inv}^v \cup \mathit{sin}^v)\\
\{\dashrightarrow\} \subseteq \mathcal{R}(x,y) \quad &\Rightarrow \quad (x,\dashrightarrow) \in (\mathit{inv}^v \cup \mathit{sin}^v)
\end{aligned}
$$

if $y \in \mathit{own}_n^w$ holds.

Projection The projection from before is extended to additionally remove those entries from *inv* and *sin* that belong to the interfering thread. Formally, we define the projection of a $(n+1)$-thread view v with

$$
\begin{aligned}
v = (&\overline{\mathit{pc}}, \mathit{pc}_{n+1}, \mathit{os}, \overline{\mathit{own}}, \mathit{own}_{n+1}, \mathit{ages}, \mathit{shape}, \mathit{freed},\\
&\overline{\mathit{in}}, \mathit{in}_{n+1}, \overline{\mathit{out}}, \overline{\mathit{out}}_{n+1}, \mathit{inv}, \mathit{sin})
\end{aligned}
$$

to an n-thread view by

$$
\begin{aligned}
\mathit{project}_{\mathit{own}}(v) = (&\overline{\mathit{pc}}, \mathit{os}, \overline{\mathit{own}}, \mathit{ages}|_{\mathbb{T}_n}, \mathit{shape}|_{\mathbb{T}_n}, \mathit{freed}|_{\mathbb{T}_n},\\
&\overline{\mathit{in}}, \overline{\mathit{out}}, \mathit{inv}|_{\mathbb{T}_n}, \mathit{sin}|_{\mathbb{T}_n}).
\end{aligned}
$$

Based on the above interference for the ownership-respecting semantics, the following section introduces an optimisation that prunes certain steps. We already exploited the ownership information for relating local variables but observe in the following that we can even omit certain interferences using this information.

4.3 Improving Interference

In the following we develop an optimisation for the interference steps for the ownership-respecting semantics from above. We first provide an observation regarding interference steps and ownership for the garbage collected semantics. Afterwards, we lift the observation to the new semantics.

The underlying observation of our optimisation is the fact that certain interference steps can be skipped for the garbage collected semantics. This is due to ownership. If a thread owns a certain part of the heap, then it is guaranteed that no other thread has access to that part. That is, the owned memory is somewhat isolated from the remaining heap. Hence, reading from or writing to the owned parts cannot affect other threads. This clearly holds due to the strong memory guarantees established by garbage collection. As a result, we may skip interference steps where the interferer would only manipulate owned memory.

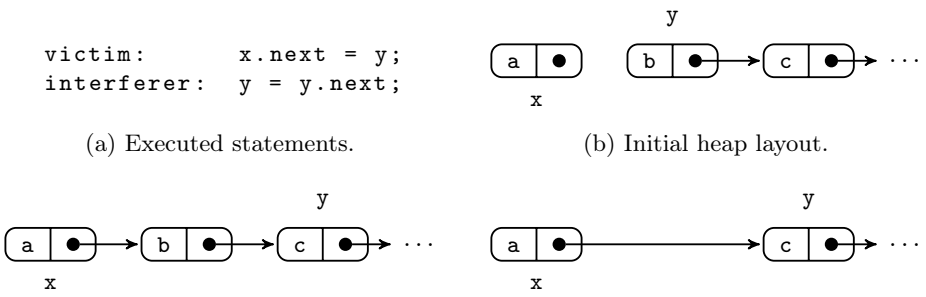
For the ownership-respecting semantics, however, the above observation cannot be established that easily. The memory guarantees given here are not as strong: owned cells can still be accesses via dangling pointers. In order to prove an analogue to the interference skipping rule from above we have to establish the fact that other threads are not influenced by the manipulation of owned addresses although they might be able to observe them through dangling pointers. Intuitively, we argue that no thread can react to changes made to owned addresses of other threads without raising a strong pointer race. Since our entire analysis is tailored around strong pointer race freedom this is no limitation to our procedure. The following theorem makes the argument precise [19].

Theorem 5 *During the analysis of a strong pointer race free program an interference step for a victim thread t_v and an interfering thread t_i can be omitted if t_i manipulates only those parts of the heap it owns.*

Proof (Sketch) *Consider the two threads t_v, t_i and the set $owned(t_i)$ of owned addresses of t_i. We first establish the fact that any pointer to an address from $owned(t_i)$ held by t_v is invalid. Upon this, we can argue that influencing t_v would raise a strong pointer race.*

Consider some pointer x local to t_v. Furthermore, assume that x references some address $a \in owned(t_i)$. Then, x is invalid. Towards a contradiction assume x was valid. By $a \in owned(t_i)$ and the definition of ownership we know that t_i allocated a and that a has not been freed since this very allocation made by t_i. Moreover, by the definition of validity, t_v has learned about a from some communication via valid, global pointers. Hence, a has been published – this contradicts the premise $a \in owned(t_i)$.

Relying on the above fact we can now argue that t_v is not influenced by t_i manipulating owned addresses. To see this, note that an influence can only occur if t_v dereferences some invalid pointer x like the above. This, however, renders the receiving variable strongly invalid. As a consequence of strong pointer race freedom, t_v cannot access or compare such strongly invalid variables. Hence, the variable is effectively dead and no influence occurs. □



(a) Executed statements.

(b) Initial heap layout.

(c) Resulting heap without interference.

(d) Resulting heap with interference.

Figure 4.1: Counter example to pruning an interference step.

Regarding the above theorem note that the symmetric does not hold: if a victim modifies some owned address, then the interference step cannot be omitted in general. To see this consider the example from Figure 4.1. The upper part of this figure describes the initial heap layout prior to the interference step in question. Assume that y is a global pointer and that x is a local pointer of a victim thread t_v. Moreover, assume that the address pointed to by x is owned by t_v. The next command that t_v wants to execute is `x.next = y`. Another thread t_i, however, tries to execute `y = y.next`. If we allow t_i to interfere t_v we explore a view with the heap layout depicted in Figure 4.1d. This view stems from first moving y to the next node and then connecting x to that node, i.e. it arises by executing an interference step followed by a sequential step. Skipping interference in such a scenario does not explore this view but yields the heap from Figure 4.1c. Hence, the analysis pruning those interference steps would not be sound.

Nevertheless, the above theorem provides a lightweight and yet powerful optimisation. In the following chapter we evaluate the overall impact of moving from the memory management semantics to the ownership-respecting semantics for thread-modular verification. We also provide detailed statistics about the impact of the above optimisation: we will see that about two thirds of the interference steps can be pruned for the ownership-respecting semantics.

Chapter 5

Evaluation

This chapter is dedicated to a practical evaluation of the presented methodologies from the previous chapters. Therefore, we first discuss our prototype implementation of the thread-modular analysis from above. This prototype supports analysing programs under garbage collection and explicit memory management. Moreover, it is the first model checker to integrate an analysis for the ownership-respecting semantics. After an introduction to our tool we do case studies on various concurrent data structures. Our experiments substantiate the usefulness of the ownership-respecting semantics: we show that this new semantics provides a speed-up of up to two orders of magnitude compared to an analysis relying on the full memory managed semantics. Still, correctness results of the analysis carry over to explicit memory management.

5.1 Prototype Implementation

In the following we discuss our C++ prototype[1] of the thread-modular analysis from above. We first give an introduction to its architecture. Afterwards, we discuss some notable differences compared to the formalism from Chapters 3 and 4.

5.1.1 Architecture

The architecture of our model checker consists of two layers: a data layer and an algorithmics layer. The data layer is depicted in Figure 5.1. It is responsible for providing the necessary data structures needed throughout a run of our tool. A description of the various classes is in order.

`Cfg` This class implements the thread-modular views described in Chapter 3. It allows to keep information about up to three threads.

`Encoding` This class is a collection of views. It is used to capture all those views that are explored by sequential and interference steps during the fixed point iteration. Adding new views to an encoding automatically merges them into the existing views as discussed in Section 3.1. The internal storage of this class is tailored towards interference steps. Therefore, it contains a set of buckets. Each bucket contains views that can potentially interfere with each other. Views of different buckets, however, cannot. This reduces the search space during interference steps.

`Observer` Class implementing the observer automata from Section 2.2.

[1] The source code is freely available at: `https://github.com/Wolff09/TMRexp`

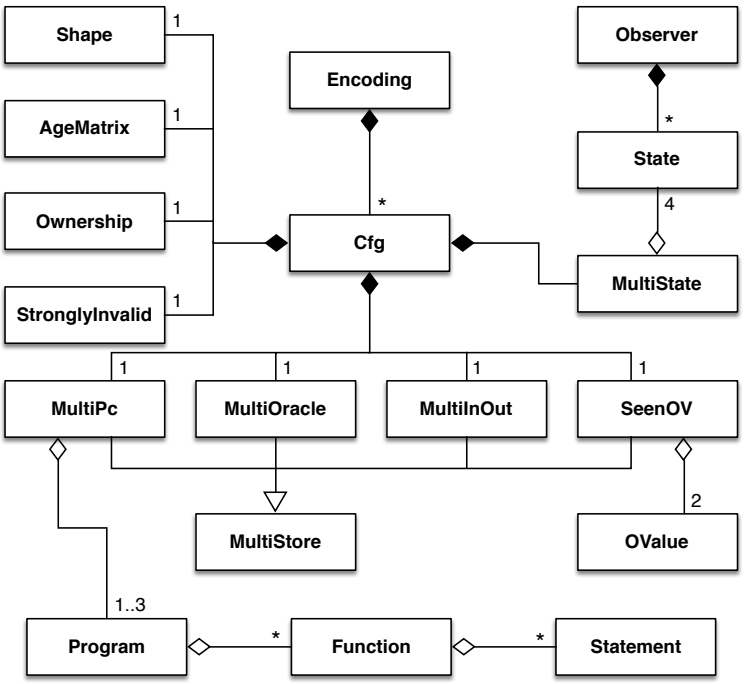


Figure 5.1: Prototype data layer.

`State` This class is used to refer to states of an observer automaton. This allows for an efficient handling of observer steps.

`MultiState` This class implements the automaton cross product. It holds multiple observer states at the same time and computes an on-the-fly cross product. More precisely, it acts as $\mathcal{A}_{stack}$ and $\mathcal{A}_{queue}$ by holding references to observer states from the automata $\mathcal{A}_{air}, \mathcal{A}_{loss}, \mathcal{A}_{dupl}$, $\mathcal{A}_{fifo}$ and $\mathcal{A}_{lifo}$ which were introduced in Section 2.2.

`Shape` A straight forward implementation of the shape matrix from Section 2.4. The underlying data structure used is a vector of vectors (`std::vector`). Although the shape is symmetric it does store all entries. This is needed as shape cells are read so frequently that computing the symmetric relations on demand would decrease performance.

`AgeMatrix` This class realises the *ages* formula contained in views from Chapter 3. However, it does not store those relations as a logic formula but uses a matrix representation.

`MultiStore` An abstract container class for storing multiple values of the same type. This class is used for the sake of uniformly handling certain information that needs to be stored for each thread.

`MultiPc` Container class based on the above `MultiStore` for program counters. The per-thread program counters are realised as pointers to the corresponding program statements.

`MultiOracle` This class provides an oracle value for each thread. Those oracles are required to properly implement conditional linearisation points of certain algorithms, like Michael&Scott's lock-free queue [33].

`MultiInOut` Implements the in_i and out_i fields for threads $1 \leq i \leq 3$ as discussed in Chapter 3. We use only one field for both in_i and out_i as we have either a return value or a formal parameter due to our programming language definition.

`OValue` Class capturing data values. Incarnations of this class represent either an observed data value, an undistinguished data value $\top$ (c.f. Section 2.4), or one of the special data values $\varnothing, \bot$.

`SeenOV` Class used to indicate which observable data values have already been used as input in a certain view. This allows to restrict the explored executions of a program to those where every data value is input at most once, as discussed in Sections 2.1 and 2.2.

`StronglyInvalid` This class provides information about those tracked pointers that are deemed strongly invalid.

`Ownership` This class stores ownership information. Following the description given in Chapter 3, ownership information is stored for pointers, not addresses. This class provides efficient means for relating pointers with the owning threads. Moreover, this class automatically handles publishing of pointers by writing their address to global pointer variables.

`Program` Top-level node of an abstract syntax tree representing the program to check.

`Function` Class representing functions from the program to check. This class is part of the abstract syntax tree and contains a list of `Statement`s.

`Statement` Abstract base class of the syntax tree nodes that represents program statements. Every statement described in Section 1.3 is modelled by a subclass of this class. Since this part is not the focus of the present thesis and follows the usual compiler construction techniques, we skip the details about the numerous subclasses here.

In contrast to the above data layer, the algorithmics layer is a loosely coupled set of namespace functions and comes basically without any class definitions[2]. In the following we describe the top-level functions.

`fixed_point` Given a program, an observer automaton and a memory semantics, this function computes the fixed point from Chapter 3. It collects all views in an `Encoding` by applying sequential and interference steps exhaustively. If a bug is found, i.e. if an observer reaches a final state or a strong pointer race is detected, the function terminates abruptly with an exception.

`post` Given a view and a thread identifier, this function computes a sequential

[2]More precisely: there are class definition which belong to the algorithmics layer. However, they are only used to pass around values more elegantly and are thus omitted hereafter.

step for the corresponding thread. There are several overloads of this function which are specialised towards handling a specific statement. That is, this function follows the case distinction done in Sections 3.2 and 4.2.2.

`mk_all_interference` This function computes all possible interference steps for a given `Encoding`. That is, it enumerates all pairs of views and conducts the procedure from Section 3.3. It order to make the enumeration more efficient this function relies on the bucket store of the `Encoding`: two views are considered for interference only if they stem from the same bucket.

The actual implementation of the algorithmics layer contains more functions than those mentioned above. The remaining functions are, however, implementation specific and not part of the interface[3]. Hence, we do not further elaborate on those functions for the architectural overview given here.

5.1.2 Implementation Details

In the following we want to stress that our tool is a prototype. It underwent major changes during the development. Some are related to the method itself. That is, we needed some initial experiments to see how the tool behaves regarding the precision of the shape analysis. This led to an iterative process which steadily refined the formalisation from Chapter 3. Moreover, the first version of the tool helped to find bugs in the theory of strong pointer races from [19]. Hence, we were required to adapt to the changes made by [19], too. Altogether, this process resulted in certain design decision which might seem unreasonable by now but are hardly to cure in the current stage of development.

The most severe impact of the above is the fact that our tool can handle only a subset of the strongly invalid pointer expressions. The tool is not able to capture information about the next fields of pointers being strongly invalid. Still we conduct a sound analysis. We do so by preventing that next fields of pointers become strongly invalid in the first place. Hence, our tool has to deem certain programs incorrect by the sheer fact that some next fields become strongly invalid. Interestingly, this shortcoming did not limit the applicability of our tool to the concurrent data structures we wanted to analyse.

Secondly, there are two versions of the tool that differ in the way age fields are handled. One version tracks only the relation among the age fields of pointers. The second version additionally considers the age of next fields of pointers. Currently, there is no automatic detection unit which switches to the one or the other version on demand. That is, one has to select the proper version of the tool when ages of next fields are needed for the analysis.

Thirdly, we want to note that we consider the formal description of the analysis given in Chapters 3 and 4 as the specification of our implementation. For the actual implementation we had to move from the set theoretic presentation to a more algorithmic approach in order to implement it in a procedural language like C++. Since those changes are very technical and the source code is freely available we do not elaborate on this fact.

[3] More precisely: the remaining functions are not defined in the C++ header files but in the corresponding implementation units.

Lastly, we want to provide some code metrics. Our prototype consists of about 5300 developer lines of code across 22 header files and 24 implementation units. It features 52 classes 33 of which are responsible for the abstract syntax tree. The algorithmics layer described above consists of a total of 94 functions. Additionally, there are 35 helper functions for generating syntax trees easily. Those functions are required since there is no parser front end.

5.2 Case Studies

The following case studies evaluate our model checker using numerous concurrent data structures. We start with simple single lock implementations that function as a sanity check for our implementation. Afterwards, we focus on the well know lock-free stack and queue implementations from Treiber [12] and from Michael and Scott [27], respectively. Lastly, we stress test our tool with erroneous programs in order to give evidence for a sound analysis.[4]

5.2.1 Single Lock Stack

In the following we apply our tool to a concurrent stack implementation that uses a single lock to establish mutual exclusion among the critical sections of the `push` and `pop` functions. Algorithm 4 gives the implementation of this algorithm. Note here, that we simulate the lock with an atomic block since our programming language does not support locks.

The linearisation points for this algorithm are easy to identify. The first one, marked with `@1`, is in the critical section of the `push` function. It unconditionally emits the linearisation event $push(d)$ where d is the value of the actual parameter `_in_`. For `pop`, there are two linearisation points `@2` and `@3` depending on whether or not the stack is empty. In case of an empty stack $pop(\varnothing)$ is emitted for the statement labelled with `@2`. Naturally, this linearisation point has the side condition `node == NULL`. In the case where the stack is not empty, the conditional inside the critical section is entered and $pop(d)$ is emitted for the statement marked with `@3`, where d is the value from `node.data`. This linearisation point has no side condition as it is guarded by the conditional.

The experimental results for the above algorithm are listed in Table 5.1. The experiments were conducted on an Intel Xeon E5-2650 v3 running at 2.3 GHz. The table includes the following readings:

1. runtime taken to establish correctness,
2. number of explored views (i.e. size of the search space),
3. number of buckets in the `Encoding`,
4. number of sequential steps,
5. number of interference steps, and
6. number of interference steps pruned by the ownership-based optimization.

[4] Our experimental results substantiating the usefulness of (strong) pointer race freedom appear in [19].

Algorithm 4 Coarse stack.

```
global Ptr ToS; /* top of stack */
local  Ptr node;

void init() {
   ToS = NULL;
}

void push(data_type _in_) {
   node = malloc();
   node.data = in;
   atomic {
      node.next = ToS;
      ToS = node;           // @1
   }
}

data_type pop() {
   atomic {
      node = ToS;           // @2
      if (node != NULL)
         ToS = node.next;   // @3
   }
   if (node == NULL) {
      _out_ = ∅;
   } else {
      _out_ = node.data;
      free(node);
   }
}
```

	GC	OWN	GC$^-$	OWN$^-$	MM
time in seconds	0.053	0.21	0.18	0.49	5.04
explored views	328	703	507	950	16117
bucket count	20	42	20	42	62
sequential steps	941	1914	1243	2477	25462
interferences steps	3624	7650	19321	33693	183388
pruned interferences	9332	20446	–	–	–

Table 5.1: Experimental results for coarse stack.

The above metrics were collected for all available memory semantics, i.e. for garbage collection (GC), the ownership-respecting semantics (OWN) and explicit memory management (MM). Additionally, we included results where we turned off the ownership-based optimisation from Section 4.3 which prunes certain interference steps (suffixed with $^-$). A discussion of our experiments is in order.

The most notable finding here is the performance of the analysis for the novel ownership-respecting semantics. On the one hand, it outperforms the analysis for explicitly managed memory. We experience a speed-up of over two orders of magnitude. On the other hand, there is only little difference between the analysis for the ownership-respecting semantics and the garbage collected semantics. This fact is rather surprising since the former analysis establishes correctness for the full memory managed semantics whereas the latter one does not.

Analysing the numbers closer gives rise to another interesting observation. The number of interference steps performed for explicit memory management is exactly those two orders of magnitude larger than the corresponding number of steps for the ownership-respecting semantics. The number of sequential steps, however, grows only by one order of magnitude. Additionally, we see that the explored state space grows by one order of magnitude, too. Altogether, those numbers show the impact of keeping more threads in a view: a quadratic blow-up as described in Section 2.3.

Lastly, note that the experimental results for the memory managed semantics are an under-approximation of the actual values. In order to establish correctness, we had to provide some hints to our tool. In this case, we wrapped the allocation and the data assignment (first two lines of the *push* function in Algorithm 4) into an atomic block to prevent the generation of false positives during interference steps. Since we did not provide hints for the remaining semantics, the significance of our experiments is not diminished.

We now turn to the next case study which addresses a concurrent queue implementation which follows the same principles as the above stack.

5.2.2 Single Lock Queue

Algorithm 5 presents a queue implementation that follows the same principle as the stack implementation from above. We use a single lock to enforce mutual exclusion of the functions `enq` and `deq`.

The linearisation points are similar to those from above. The first one, marked with `@1`, is in the `enq` function and is emitted unconditionally. For `deq` we have two linearisation points `@2` and `@3`. The former accounts for the fact that the queue is empty and the latter handles the case where the value `next.data` is dequeued. Both those events are emitted without any side conditions.

Experiments are conducted in the same environment as before and the same numbers are collected. The results can be found in Table 5.2. The similarity of this algorithm compared to the above stack implementation is reflected in the experimental results. The observations from above carry over. Moreover,

Algorithm 5 Coarse queue.

```
global Ptr Head, Tail;
local  Ptr node, next;

void init() {
   Head = malloc();
   Tail = Head;
}

void enq(data_type _in_) {
   node = malloc();
   node.data = _in_;
   node.next = NULL;
   atomic {
      Tail.next = node;
      Tail = node;            // @1
   }
}

data_type deq() {
   atomic {
      node = Head;
      next = Head.next;
      if (next == NULL) {
         _out_ = ∅;           // @2
      } else {
         /* read data inside the atomic block to ensure
          * that no other thread frees "next" in between
          */
         _out_ = next.data;   // @3
         Head = next;
      }
   }
   if (next != NULL)
      free(node);
}
```

	GC	OWN	GC$^-$	OWN$^-$	MM
time in seconds	0.043	0.57	0.19	2.29	31.18
explored views	199	520	331	790	27499
bucket count	15	30	15	30	93
sequential steps	589	1335	779	1966	60271
interferences steps	1008	8545	9530	65025	442306
pruned interferences	4986	27224	–	–	–

Table 5.2: Experimental results for coarse queue.

our tool needs the same hints in order to establish correctness for the memory managed semantics. Hence, instead of elaborating on the experiments for this algorithm we now turn towards non-blocking data structures.

5.2.3 Treiber's Lock-Free Stack

In the following we present experimental results of running our model checker on Treiber's lock-free stack [12]. The implementation of this algorithm was already given in Chapter 1. However, to apply our model checker we have to revisit this implementation and translate it into our restricted programming language. Algorithm 6 gives an updated version of the implementation. For a description of the linearisation points, refer to Section 1.2.3. The experiments where conducted as above and the overall results can be found in Table 5.3. A discussion of the findings is in order.

Algorithm 6 Treiber's stack translated into our programming language.

```
global VPtr ToS; // top of stack
local  VPtr top, node;

void init() {
   ToS = NULL;
}

void push(data_type _in_) {
   node = malloc();
   node.data = _in_;
   while (true) {
      top = ToS;
      node.next = top;
      if (dCAS(ToS, top, node))        // @1
         break;
   }
}

data_type pop() {
   while (true) {
      top = ToS;                       // @2
      if (top == NULL) {
         _out_ = ∅;
         break;
      } else {
         node = top.next;
         if (dCAS(ToS, top, node)) {   // @3
            _out_ = top.data;
            free(top);
            break;
         }
      }
   }
}
```

	GC	OWN	GC$^-$	OWN$^-$	MM
time in seconds	0.059	2.37	0.146	3.89	612
explored views	269	744	269	746	116776
bucket count	10	17	10	17	26
sequential steps	779	2656	837	2174	322328
interferences steps	4159	45815	11530	73470	7913705
pruned interferences	14196	88897	–	–	–

Table 5.3: Experimental results for Treiber's lock-free stack.

First, we compare the results for the analysis using garbage collection with the one using the ownership-respecting semantics. Here we see that the gab between those two analyses is bigger than for the single lock implementations from above. We experience a slow-down of two orders of magnitude when moving from garbage collection to the ownership-respecting semantics.

Another interesting observation can be made when comparing the two runs of the ownership-respecting semantics. We find that the optimisation prunes certain steps such that the unoptimised version executes twice as many interference steps. Interestingly, we also find that the optimised version executes more sequential steps. Although this looks like a bug, it is not. This artefact stems from the order in which sequential steps and interference steps are executed. The unoptimised version simply generates certain views *earlier* than the optimised version. That is, there are some views that are merged and then a sequential step is conducted. The optimised version, however, may execute the sequential step first and later merge the view with another one. Hence, a second sequential step is required in order to capture the impact of the merge.

Lastly, we want to stress the most notable finding from our experiments. The comparison of the ownership-respecting and the memory managed semantics substantiates the usefulness of the new memory semantics introduced by [19]: a speed-up of over two orders of magnitude. Not only the runtime improvements are outstanding. We also experience a search space that is smaller by two orders of magnitude. In addition, the number of sequential and interference steps saved lies in the same order. This further emphasises the usefulness of the ownership-respecting semantics as it improves the scalability of every analysis relying on the thread-modular framework.

Having Treiber's stack verified successfully lets us turn towards more complex algorithms. In the following section we focus on the well-known lock-free queue implementation from Michael and Scott.

5.2.4 Michael&Scott's Lock-Free Queue

As the next case study we want to discuss Michael&Scott's lock-free queue [27]. An example implementation is given in Algorithm 7. The linearisation points `@1` and `@3` follow the same principles as those from Treiber's stack. Linearisation point `@2`, however, is a bit more involved. We may only emit a linearisation

event here if the following execution detects that the queue is empty. It is not sufficient to guard this event with the conditions of the following conditionals. This is the case since thread-local pointers are compared to global pointers. Hence, the outcome of the conditionals depends on interfering threads. That is, the condition of the linearisation point must consider possible interferences. To overcome this problem, we employ an oracle variable [33] that tells us whether or not those comparisons of local and global pointers succeed. As a consequence, we have to ensure that the comparisons adhere to the prophecy of the oracle. As discussed briefly in Section 5.1.1, our tool is able to handle those oracle variables. However, we skip an elaboration of the technical details as the implementation is straight forward.

The complexity of Michael&Scott's queue is not only reflected in the above linearisation point. Comparing the algorithm with Treiber's stack, one finds that it features more sophisticated means of detecting whether a thread was influenced by an interferer. While in Treiber's stack this is done via the version counter of the global top of stack pointer, this queue implementation relies on the version counters of next fields from multiple entries. Hence, we expect the analysis of our tool to be much more expensive. Unfortunately, we are not able to establish correctness for this algorithm at all. The reason for the failure is imprecision. The shape analysis in combination with merging views ultimately yields false positives that cannot be avoided in our implementation. Since [2] was able to establish correctness with a similar technique, we conclude that the overall approach is very sensitive to subtle changes. A in-depth analysis of the precise reasons for the imprecision are, however, hardly possible. Collecting debug data for this algorithm easily yields gigabytes of data that one can impossibly inspect by hand. We would need to automatically generate traces from the debug data that show the underlying problem and its *development*. Unfortunately, such a study is out of scope for the present thesis.

However, we want to stress that we cannot provide experimental results here due to imprecision in our instantiation of the thread-modular framework with shape analysis. The fact that both, the ownership-respecting and the memory managed semantics, suffer from the same problem suggests that this is no shortcoming of the overall approach of relying on the ownership-respecting semantics. That is, the inability of our tool establishing correctness for Michael&Scott's queue does not diminish the findings of our previous experiments.

As this case study explored the limitations of our prototype implementation it remains to stress test our tool in the following section.

5.2.5 Detecting Defects

As a last case study we want to provide experimental evidence for the soundness of our analysis. Therefore, we stress test our tool with programs where we purposely inserted bugs. A description of the tested programs is in order.

ABA-Stack This program coincides with Treiber's stack but omits the version counters. Hence, it suffers from the ABA problem and we expect our tool to find a specification violation. Intuitively, we expect that the stack losses some values, i.e. $\mathcal{A}_{loss}$ reaches a final state. However, there is another

Algorithm 7 Michael&Scott's lock-free queue, adapted from [27].

```
global VPtr Head, Tail;
local  VPtr node, head, tail, next;

void init() {
   Head = malloc();
   Head.next = NULL;
   Tail = Head;
}

void enq(data_type _in_) {
   node = malloc();
   node.data = _in_;
   node.next = NULL;
   while (true) {
      tail = Tail;
      next = tail.next;
      if (tail == Tail)
         if (tail.age == Tail.age)
            if (next == NULL) {
               if (dCAS(tail.next, next))       // @1
                  break;
            } else dCAS(Tail, tail, next);
   }
   dCAS(Tail, tail, node);
}

data_type deq() {
   while (true) {
      head = Head;
      tail = Tail;
      next = head.next;                          // @2
      if (head == Head)
         if (head.age == Head.age)
            if (head == tail) {
               if (next == NULL) {
                  _out_ = ∅;
                  break
               }
               dCAS(Tail, tail, next);
            } else {
               _out_ = next.data;
               if (dCAS(Head, head, next)) {    // @3
                  free(head);
                  break;
               }
            }
   }
}
```

Test		Time in seconds	Result
ABA-Stack		0.072	$\mathcal{A}_{dupl}$ reaches final state
Fifo-Stack	coarse	0.001	$\mathcal{A}_{fifo}$ reaches final state
	Treiber's	0.002	$\mathcal{A}_{fifo}$ reaches final state
Lifo-Queue	coarse	0.002	$\mathcal{A}_{lifo}$ reaches final state
Bad-LinP	`@1`, early	0.09	$\mathcal{A}_{loss}$ reaches final state
	`@1`, late	0.11	$\mathcal{A}_{air}$ reaches final state
	`@2`, early	0.12	error: multiple linearisation events emitted
	`@2`, late	0.92	$\mathcal{A}_{loss}$ reaches final state
	`@3`, early	0.02	$\mathcal{A}_{dupl}$ reaches final state
	`@3`, late	0.03	$\mathcal{A}_{air}$ reaches final state

Table 5.4: Experimental results for erroneous programs.

possibility where some data values are duplicated. This scenario occurs for the same reasons as the classical ABA problem but leads to the $\mathcal{A}_{dupl}$ observer reaching a final state.

Fifo-Stack A sanity check for the specification: we check whether a stack implements the first-in-first-out principle known from queues. Hence we run the coarse stack and Treiber's stack with the observer $\mathcal{A}_{queue}$. We expect to reach the final state of $\mathcal{A}_{fifo}$.

Lifo-Queue As above, we check whether a queue adheres to the last-in-first-out principle. The expected behaviour is that $\mathcal{A}_{lifo}$ reaches its final state.

Bad-LinP We move the linearisation point in Treiber's stack. Any subtle change should result in a specification violation. For every linearisation point we conduct two tests where we fire it earlier and later than the correct one. We expect that every erroneous linearisation point results in a specification violation. The actual accepting observer depends on the test case.

We conduct the experiments in the same environment as above. However, we only stress test the ownership-respecting semantics as the analysis for the other semantics is not part of our contribution. The results can be found in Table 5.4. We see that in all cases our tool is able to find the intentionally added bugs in short time. As discussed before, our tool does not provide a back trace of the found bugs. That is, the user is only prompted with the fact that a certain error occurred or a certain observer reached its final state.

This last set of test cases concludes the case studies on the performance of our novel thread-modular analysis relying on the ownership-respecting semantics. Altogether, we provided substantial evidence for the usefulness of the ownership-respecting semantics and the accompanied notion of strong pointer races: a speed-up of up to two orders of magnitude. Moreover, we provided experiments supporting the soundness of our analysis.

Chapter 6

Future Work and Conclusion

In the following we provide a discussion on possible directions of future work and then conclude this thesis.

6.1 Future Work

Our developed understanding and experience with thread-modular verification for explicit memory management and the drawbacks of our prototype implementation yield several areas of potential improvement and thus possible future work. First, we suggest to investigate the impact of a preciser analysis. Since our tool suffers from imprecision for algorithms like Michael&Scott's lock-free queue, it would be interesting to see how our analysis performs when we abandon the concept of merging (c.f. Section 3.1). Although works like [2] have shown that this results in out-of-memory problems for the full memory managed semantics, we believe that relying on the ownership-respecting semantics could work well. The much smaller search space of the latter semantics might compensate the increased memory requirements. Moreover, this would make the expensive shape splitting superfluous.

Addressing the same direction as above, one could replace the shape analysis with another heap abstraction technique. Here, a promising approach are the symbolic memory graphs (SMGs) introduced by [13]. They represent the heap as a graph where objects and values correspond to nodes and their relations, e.g. a certain object contains certain values, are modelled with edges. Using this technique could additionally allow to lift the thread-modular approach to support an industry-grade programming language like C. This could be done since SMGs are byte-precise and thus able to handle arrays and low-level memory operations like pointer arithmetic.

Another area of future work is an improved implementation of our thread-modular analysis. Currently, we explicitly store the building blocks of views. One could think about symbolically representing an entire view as a single formula. Then, sequential and interference steps could be lifted from the set theoretic representation to a logic level. This could allow for the use of SAT solvers to compute the actual *post* operator or even the entire sequential and interference steps. Since those solvers have seen severe improvements over the last decade, this could lead to a dramatical speed-up of the analysis. As an alternative to SAT solvers, one could investigate the possibilities of re-building our analysis on top of the TVLA framework [24].

Picking up on the described shortcomings of our prototype regarding the case study on Michael&Scott's lock-free queue, one could improve the user experience of our tool. Therefore, it would be interesting to compute traces that precisely

show why certain bugs in a program occur. This would make our tool useful for fault-localisation during developer tests. Moreover, this could help programmers understand the fine-grained effects of non-blocking implementations.

Lastly, one can think about a parallel implementation of our prototype. The fixed point iteration of the thread-modular framework is inherently parallel. It generates new views from the existing ones. Hence, multiple worker threads could concurrently generate those new views. The main challenge is to properly synchronise the worker threads such that no corrupt views are read. The problem here is that views can be merged, i.e. they might be updated after they were added to the `Encoding`. Hence, a thread has to ensure that a read view is consistent in the sense that it has not been update during the read.

With the above directions of future work it remains to conclude this thesis.

6.2 Conclusion

In Chapter 2 we have discussed how to automatically prove linearisability of concurrent stacks and queues for garbage collection and explicit memory management. We presented several techniques to overcome the numerous obstacles of this task. First, we introduced the data independence argument from Wolper. It allowed us to restrict our verification efforts to a subset of all possible program executions, namely those where data values are considered to occur no more than once. This led to very elegant specifications based on observer automata.

With this powerful specification technique at hand, we could turn towards the analytical part of the program analysis. Therefore, we used the thread-modular framework to handle an unbounded number of concurrently working threads. This framework suggests to abstract the state space to views representing single threads. This representation of the state space, however, lacks information about possible interleavings of those threads. Hence, a pure thread-modular analysis has to account for all possible interferences – even those that cannot occur. We discussed that this gives rise to false positives and makes proving reasonable programs impossible as one needs to relate the local states of threads properly. For garbage collection, one can address this shortcoming by exploiting ownership information. For explicit memory management, however, this approach is not feasible. Hence, one has to increases the number of threads kept in a view which leads to a quadratic blow-up of the state space.

Chapter 3 instantiated a thread-modular analysis building upon the techniques from Chapter 2. We provided a formal description covering all parts of the analysis. Altogether, this chapter was then used as a basis for the later adaption and implementation of the prototype.

In Chapter 4 we discussed how to cure the poor scalability of the thread-modular analysis from Chapters 2 and 3 for explicit memory management. Therefore, we introduced the notion of strong pointer races from [19]. By restricting the verification efforts to strong pointer race free programs, it was possible to put ownership back into the thread-modular approach. We then adapted the thread-

modular analysis from Chapter 3. Here we used a similar approach as for garbage collection. We instantiated the thread-modular framework with views keeping information about one thread only. To overcome the problems described before, we exploited the ownership model from the ownership-respecting semantics.

In Chapter 5 we then showed that the new analysis for strong pointer race free programs is amenable to thread-modular reasoning. That is, we were able to establish correctness for data structures like Treiber's lock-free stack. Moreover, we showed that the analysis provides a speed-up of two orders of magnitude compared to the analysis for explicit memory management from Chapter 3.

Lastly, in order to conduct a sound analysis, we also integrated a check for strong pointer races. We showed how to interleave this check with the actual analysis in Chapter 4. Due to this check we were able to show that common lock-free data structures like Treiber's stack are free from strong pointer races. Hence, we provided experimental evidence for the usefulness of relying on strong pointer race freedom for the task of verifying performance-critical code.

Bibliography

[1] Programming languages — C. ISO/IEC Standard 9899:2011, page 348, International Organization for Standardization, Geneva, Switzerland, 2011.

[2] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In N. Piterman and S. A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 324–338. Springer, 2013.

[3] D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 477–490. Springer, 2007.

[4] M. Barr. An update on toyota and unintended acceleration. `http://embeddedgurus.com/barr-code/2013/10/an-update-on-toyota-and-unintended-acceleration/`, October 2013. Accessed: 16. October 2015.

[5] M. Barr. Killer apps: Embedded software's greatest hit jobs. `http://www.barrgroup.com/files/killer_apps_barr_keynote_eelive_2014.pdf`, April 2014. Accessed: 16. October 2015.

[6] P. Barry and P. Crowley. *Modern Embedded Computing: Designing Connected, Pervasive, Media-rich Systems*. Morgan Kaufmann, 2012.

[7] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv. Thread quantification for concurrent shape analysis. In A. Gupta and S. Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2008.

[8] CBS News. Toyota "unintended acceleration" has killed 89. `http://www.cbsnews.com/news/toyota-unintended-acceleration-has-killed-89/`, May 2010. Accessed: 16. October 2015.

[9] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[10] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.

[11] I. Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 2 (2A, 2B&2C): Instruction Set Reference, A-Z. June 2015.

[12] I. B. M. C. R. Division and R. Treiber. *Systems Programming: Coping with Parallelism*. Research Report RJ. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[13] K. Dudka, P. Peringer, and T. Vojnar. Byte-precise verification of low-level list manipulation. In F. Logozzo and M. Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 215–237. Springer, 2013.

[14] M. Dunn. Toyota's killer firmware: Bad design and its consequences. `http://www.edn.com/design/automotive/4423428/Toyota-s-killer-firmware--Bad-design-and-its-consequences`, October 2013. Accessed: 16. October 2015.

[15] European Space Agency. Ariane 501 Inquiry Board report. `http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`, July 1996. Accessed: 16. October 2015.

[16] C. Flanagan and S. Qadeer. Thread-modular model checking. In T. Ball and S. K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, volume 2648 of *Lecture Notes in Computer Science*, pages 213–224. Springer, 2003.

[17] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In J. Ferrante and K. S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 266–277. ACM, 2007.

[18] A. Gotsman, N. Rinetzky, and H. Yang. Verifying concurrent memory reclamation algorithms with grace. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 249–269. Springer, 2013.

[19] F. Haziza, L. Holík, R. Meyer, and S. Wolff. Pointer race freedom. 2015. Accepted for VMCAI 2016.

[20] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[21] M. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 13–26. ACM Press, 1987.

[22] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[23] R. Lazic and D. Nowak. A unifying approach to data-independence. In C. Palamidessi, editor, *CONCUR 2000 - Concurrency Theory, 11th In-*

ternational Conference, University Park, PA, USA, August 22-25, 2000, Proceedings, volume 1877 of *Lecture Notes in Computer Science*, pages 581–595. Springer, 2000.

[24] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In J. Palsberg, editor, *Static Analysis, 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000, Proceedings*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer, 2000.

[25] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. Technical Report MSR-TR-2009-29, March 2009.

[26] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification is cartesian abstract interpretation. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *Theoretical Aspects of Computing - ICTAC 2006, Third International Colloquium, Tunis, Tunisia, November 20-24, 2006, Proceedings*, volume 4281 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2006.

[27] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, May 1998.

[28] F. L. Morris and C. B. Jones. An early program proof by alan turing. *IEEE Ann. Hist. Comput.*, 6(2):139–143, Apr. 1984.

[29] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, Jan. 1998.

[30] M. Segalov, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Abstract transformers for thread correlation analysis. In Z. Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2009.

[31] V. Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

[32] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In N. D. Jones and M. Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 335–348. Springer, 2009.

[33] V. Vafeiadis. Automatically proving linearizability. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2010.

[34] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*, pages 184–193. ACM Press, 1986.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Kaiserslautern, _____ _____

Date Signature