DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS
64

VESAL VOJDANI

Static Data Race Analysis of Heap-Manipulating C Programs

Institute of Computer Science, Faculty of Mathematics and Computer Science, University of Tartu, Estonia

Dissertation accepted for public defense of the degree of Doctor of Philosophy (PhD) on October 14, 2010 by the Council of the Institute of Computer Science, University of Tartu.

Supervisors:

Prof.	Varmo Vene University of Tartu Tartu, Estonia
Prof.	Helmut Seidl Technische Universität München Garching, Germany

Opponents:

Prof.	Alan Mycroft University of Cambridge Cambridge, UK
Dr.	Keiko Nakata Institute of Cybernetics at Tallinn University of Technology Tallinn, Estonia

The public defense will take place on December 10, 2010 at 14:15 in Liivi 2-405.

The publication of this dissertation was financed by Institute of Computer Science, University of Tartu.

ISSN 1024-4212
ISBN 978-9949-19-508-4 (trükis)
ISBN 978-9949-19-509-1 (PDF)

Tartu Ülikooli Kirjastus
http://www.tyk.ee
Tellimus nr. 642

Contents

LIST OF ORIGINAL PUBLICATIONS

1. Vojdani, V., Vene, V.: Goblint: Path-sensitive data race analysis. In: SPLST'07. pp. 130–141. Eötvös Lorand Univ. (2007)

2. Vojdani, V., Vene, V.: Goblint: Path-sensitive data race analysis. Annales Univ. Sci. Budapest., Sect. Comp. 30, 141–155 (2009)

3. Seidl, H., Vojdani, V., Vene, V.: A smooth combination of linear and Herbrand equalities for polynomial time must-alias analysis. In: FM'09. LNCS, vol. 5850, pp. 644–659. Springer (2009)

4. Seidl, H., Vojdani, V.: Region analysis for race detection. In: SAS'09. LNCS, vol. 5673, pp. 171–187. Springer (2009)

5. Kreiker, J., Seidl, H., Vojdani, V.: Shape analysis of low-level C with overlapping structures. In: VMCAI'10. LNCS, vol. 5944, pp. 214–230. Springer (2010)

ABSTRACT

A particularly elusive flaw in shared-memory concurrent systems is the *data race*, a situation where multiple threads may simultaneously access the same memory location, and at least one of the accesses is a write. This dissertation contends that static analysis can be used to verify the absence of data races in real world systems, especially operating system modules like Linux device drivers. The difficulty in analyzing such code is that both data structures and locks protecting the data are created at run-time. This naturally poses a problem for static analysis; indeed, most commercial race detection tools are based on run-time analysis. In this dissertation, we present three key innovations that together enable static data race analysis of heap-manipulating C programs.

First, in order to deal with fine-grained locking, a scheme in which each data structure contains its own dedicated lock, we present an abstract domain to infer *must-equalities* between address expressions. Second, we address medium-grained locking schemes, such as having a common list lock protecting elements of a linked list, by associating accesses into the heap with static owners of the *region* in which an element resides. This allows the application of the techniques for fine-grained locking to now correlate the locks with the owners of regions. Third, in order to deal with migrating elements and *unsharing*, we refine the global invariant approach of Seidl, Vene and Müller-Olm to deal with temporary violations of heap invariants within critical sections, and we provide a shape analysis capable of determining whether elements have been removed from embedded linked list structures, as used in the Linux kernel.

We have implemented these techniques in the Goblint analyzer and used it to experimentally validate the contention that verification of race-freedom in real-world systems is possible by means of static analysis.

CHAPTER 1

INTRODUCTION

Writing and debugging concurrent programs is notoriously difficult: due to the non-deterministic nature of thread scheduling, a system may contain latent flaws which are not detected by conventional approaches to quality assurance. In the classic example of the Therac-25 radiation therapy machine, only experienced practitioners could elicit the murderous race condition because testers were not "quick-fingered" enough to trigger the underlying concurrency flaw.

Given the increasing prevalence of multi-core architectures, tools and programming languages are needed to support the development of concurrent software systems. Most safety-critical systems, the programs whose correctness we most urgently need to verify, are written in low-level languages that are extremely hard to reason about. These programming languages give the programmer complete control over the synchronization mechanisms to more efficiently manage the limited resources of an embedded system. The drawback to such fine-grained control is that the programmer must use the low-level concurrency primitives *correctly* in order to avoid concurrency hazards. The two main sources of problems specific to low-level concurrent programming are race conditions and deadlocks. This thesis is concerned with race detection for low-level C code.

More precisely, this thesis is concerned with sound, static and fully automatic race detection. Static race detection means we attempt to detect flaws without executing the program. Just as structural engineers can predict how a construction will support and resist imposed loads, we approximate the run-time behaviour of a program by solving a system of data flow equations. Soundness implies that we approximate this behaviour conservatively, thus when successful, we certify that the program is free from data races. Finally, our approach is fully automatic and can in principle be run on a raw piece of C code by pressing a big red button.

Our primary goal is the verification of system modules, such as Linux device drivers, without analyzing the rest of the system. Thus we make certain worst-case assumption about the environment. When we succeed in verifying a module,

we can be sure the module under analysis is safe; that is, the current module is not to blame for any potential race occurring due to external code disrespecting the synchronization protocol of the analyzed module. For this application domain, we need techniques to analyze modules of up to ten thousand lines of code precisely enough to verify absence of races. Thus, we are interested in comparatively small programs, rather the challenge is that the code is fairly complicated and low-level.

1.1 Race conditions

A race condition is a general term which refers to any flaw in a concurrent software system where the result of a computation may depend on the scheduling of the threads in a way not anticipated by the developers. Netzer and Miller [65] distinguish between *general races*, which introduce undesired non-determinism in programs intended to be deterministic, and *data races*, which lead to data inconsistency in (intentionally non-deterministic) programs because sections manipulating shared data are not executed atomically.

Most work on automated data race detection is concerned with an even narrower condition, namely the *multiple access data race*. This is a specific condition that excludes many atomicity flaws, and it is a necessary condition to avoid potential data corruption; however, there are atomicity violations, such as stale-value concurrency errors [2], which are not captured by this notion of a data race. Nevertheless, in this thesis (and in most papers on race detection), when we speak of race conditions and data races, we mean the following.

Definition 1 (Data Race)**.** A multiple access data race is a condition in multithreaded programs where different threads may simultaneously access the same shared memory location, and at least one of the accesses is a write.

We now dissect this definition into the three key requirements for two accesses to qualify as a race. First, the definition requires that the access statements are executed by *different threads*. If a shared memory location is only accessed by a single thread, it is *thread-local* and cannot be subject to a data race. By default, local variables are thread-local and global variables are shared. However, local variables may escape the confines of a single thread, for example, when a parent thread passes a reference to one of its locals as argument to a spawned thread. Conversely, global data may be used by just a single thread; in particular, dynamically allocated memory, which in principle belongs to the shared heap, may actually be thread-local. Deciding which shared memory locations are exclusively accessed by a single thread increases precision; failing to detect if a child thread may access a local variable of its parent would be entirely unsound.

Second, the definition requires the accesses to occur *simultaneously*. Strictly speaking, concurrent computing does not necessarily require a parallel architecture. When we speak of simultaneous accesses, we mean that there are no constraints on how the scheduler may order the machine instructions which realize the two access statements. Thus, if the scheduler is free to interleave the execution of two statements, we think of these accesses as occurring simultaneously.

There are a number of different ways to ensure that statements do not occur simultaneously. For example, there may be temporal constraints induced by the program logic which ensure that one access will always happen before the other. The most common method for enforcing that accesses do not occur simultaneously is to ensure that critical segments of code are *mutually exclusive*. Posix C provides mutexes (mutual excluders) for the programmer to delimit the section of code which may access the same shared memory location by the locking and unlocking of a common mutex. The system allows at most one thread to hold any given mutex, thereby ensuring that accesses will not be simultaneous, provided the programmer correctly uses the same mutex to guard the two accesses.

Finally, the definition requires that the threads access the *same memory location*. Identifying the memory location that a given program statement may access is complicated due to the use of pointers. When dynamically allocated data structures are accessed indirectly through pointers into the heap, determining whether two access statements may refer to the same location requires precise information about the locations that pointer variables may point to.

1.2 Challenges in static race detection for C

The basic technique in static data race detection is to ensure the following condition: for each shared memory location, there exists (at least) one lock which is held whenever a thread accesses that memory location. In order to determine whether a common lock exists, one may compute the set of locks that are held by the executing thread at each program point. As execution may reach a given program point along different execution paths, a sound analysis only takes into account those locks which are held across all paths reaching that point. Having computed the sets of definitely held locks, one checks for each shared memory location the intersection of the locksets at the points where that memory location is accessed. If the intersection is non-empty, one can conclude that there is no race at that given memory cell; otherwise, the analysis warns that there is a potential data race.

In order to apply the above basic idea to analysing real C programs, one has to address the following challenges.

```
int x;   mutex m1 = MUTEX_INIT;
int y,z; mutex m2 = MUTEX_INIT;

void munge(int *v, mutex *m) {
    lock(m); (*v)++; unlock(m); }

thread t1() {                  thread t2() {
    munge(&x, &m1);                munge(&x, &m1);
    munge(&y, &m2);                munge(&y, &m1);
    munge(&z, &m2); }              munge(&z, &m2); }
```

Figure 1.1: Illustrating the need for context-sensitive pointer analysis [68].

Context-sensitive alias analysis. Determining statically the memory locations that are being accessed is not a trivial task. Even without dynamic memory allocation, pointers to static global variables need to be resolved. If two distinct pointers, `p` and `q`, may alias, i.e., point to the same memory location, then syntactically distinct accesses, e.g., `p→data` and `q→data`, may participate in a data race. What makes this particularly challenging is that the locking and unlocking operations of C are not lexically scoped, hence the information about pointers needs to be tracked *context-sensitively*, as is illustrated through the following example.

Figure 1.1 contains a simple program with two threads that execute calls to a `munge()` function. This function increments a shared variable while acquiring and releasing a mutex; both the variable and the mutex are given through pointer parameters. The effect of the function depends on the parameters with which it is called. If these calling contexts are conflated, the analysis will fail to deduce anything sensible about the program: `v` may point to of the three shared variables, while `m` may point to any of the two locks. As such functions commonly occur in real programs, context-sensitive propagation of pointer values is critical.

Races in the heap. It should come as no surprise that *dynamic* memory allocation is extremely difficult for *static* analyzers. Since many serious program errors relate to memory safety, the analysis of the heap is currently a highly active area of research. When it comes to race detection tools, most of them still rely on summarizing all data allocated at a given program point into a single representative *blob*; that is, a single static name, such as "alloc@file.c:38", is used to represent all elements created at a given allocation site.

The problem with summarization for race detection is that on the one hand, we have to treat an accesses to a blob as an access to *all* the objects it represents;

on the other hand, we must consider the locking of a blob as taking *none* of the locks. This asymmetry is a consequence of having to ensure that if two threads *may* access the same element, they *must* lock the same lock. This problem can be illustrated even without dynamic memory. It is quite natural for an object to contain a dedicated lock which ensures mutually exclusive access to its data fields, as in the following example:

```
struct { int datum; mutex mtx; } A, B;
if (test) p = &A; else p = &B;
lock(&p→mtx); p→datum++; unlock(&p→mtx);
```

After the non-deterministic branching the pointer `p` may point to either A or B, so when we acquire the lock `p→mtx` we may hold either one of them, and thus neither of them is *definitely* held. Although most sound race detection tools will raise a false alarm for this example, it is perfectly clear that the code is safe; after all, the same pointer is used for both the access and for the locking. The situation is analogous when `p` points to a blob of dynamically allocated memory.

Temporary violations of invariants. While most approaches to race detection attempt to infer invariants on the heap that hold throughout the entire program execution, dynamically allocated objects often have a life cycle: initialization, consumption, and destruction. A freshly allocated object is only accessible by the thread that allocated it until the thread *publicizes* the element by connecting it to the rest of the heap, e.g., by adding it to a shared linked list. Similarly, near the end of an objects life-time, it may be *privatized*, i.e., removed from the data structure where it resided, so that it is only accessible by a single thread.

Discovering when a freshly allocated object becomes shared is much easier than dealing with privatization. *Escape analysis* is a common technique to discover when a freshly allocated object escapes the thread and becomes accessible to other threads. Once an object has escaped and is part of the heap, one can only discover its removal through careful analysis of the heap because one has to ensure that the element is no longer reachable from any shared heap element.

Conditional locking and variations on locking. A program may check a condition to determine if it needs to acquire a lock, and then based on a related condition decide if an access should occur. Path-sensitivity is the ability of the analysis to distinguish feasible paths from infeasible ones:

```
if (do_work) lock(&mtx); ...; if (do_work) work++;
```

There are 4 paths in the above code, but only two are valid paths. This is also an issue when thread creation and mutex locking operations may fail, such as pthread's `trylock` and the kernel's `lock_interruptible`. The return value

must then be used to correlate the different locksets that result from potentially failing locking operation with the values of program variables.

Synchronization-sensitivity. Even in a multi-threaded program, a thread may not be running in parallel with all other threads at all times. There are many mechanisms to achieve lock-free synchronization, but these are often very hard to analyze. Something can be done by attempting to track thread identities and inferring which threads may possibly run in parallel. As this has not been the focus of our work, we will discuss this in the conclusion as an area of future work.

1.3 An overview of our solution

Since the static analysis of multithreaded software is notoriously difficult [72], most race detection tools are either based on dynamic analysis, which cannot prove the absence of bugs, or use type-based approaches, which rely on time-consuming programmer annotations [31]. More recently, however, some impressive static data race analyzers have been presented [47, 61, 68]. In contrast to these race detection methods, we track the values of program variables. This allows us to solve the path-sensitivity problem by using a technique called *property-simulation* [23]. This technique analyzes the state of the program for each configuration of the relevant property, namely locksets. However, tracking the values of variables comes with a price. As we are dealing with multi-threaded programs, the price could be very high: the number of possible interleavings of threads grows exponentially with the size of concurrently executable code. Many researchers have tackled this problem, and one promising approach is the design of *thread-modular* analyses [30]. The idea is to compute the influence of each thread on the rest of the program, and then use this information to analyze each thread in isolation.

Seidl, Vene and Müller-Olm [76] propose a sound framework for thread-modular analysis based on computing global invariants. Posix threaded applications communicate through global variables and heap allocated data. Hence, they compute the *side-effects* of each thread on the global state, and use this information for the sequential analysis of each thread. These two computations are performed simultaneously using a demand-driven fix-point solving algorithm [29]. This has been implemented in the Goblint analyzer and the benefits and details of the approach are discussed in Chapter 2.

The key contributions of this thesis are the techniques for race detection of heap-manipulating programs based on combining may and must-alias information. As a general approach, we use pointer must-equality analysis to reason about

the access and the lock relative to a root pointer (e.g., `p→datum` is accessed while holding `p→mtx`) and then instantiate the found invariant to all elements that `p` may point to, including blobs. In the example above, we would instantiate the inferred invariant to obtain that `A.datum` is accessed holding `A.mtx` and `B.datum` is accessed holding `B.mtx` because the root pointer `p` has the may point-to set $\{\&A, \&B\}$. For this one needs to primarily reason about must-equalities between pointer expressions, which is the subject of Chapter 3.

The above method also works for a summarized blob of memory if the locking scheme is *fine-grained*, i.e., each element allocated at a program point has its own lock. Then, an existentially quantified invariant can be associated with the entire blob of memory. However, the locking granularity is often slightly more coarse. Linked list data structures may have a list-lock which protects access to all elements in the list. We can deal with this by associating dynamically allocated elements not with their allocation site, but rather with the list head into which they are placed. The list head is then a static owner of the disjoint portion of the heap reachable from it. If we can infer disjoint regions in the heap, we know that pointers into different regions may not alias, hence an access through a pointer can be associated with the owner of the region into which it points. The must-equality information can then be used to infer correlations between the owners and held locks. This approach to medium-grained locking is explained in full detail in Chapter 4.

While these techniques work well for the analysis of simpler device drivers, for more complicated modules, the heap abstraction is too static. In our region-based analysis, once an element is inserted into a list, it is part of that region and cannot be moved out. If an element is moved from one list to another, we conservatively collapse the two regions. In Chapter 5, we relax the condition that global invariants must hold at all times during the program execution. We allow a region for which a thread has obtained exclusive access to be analyzed sequentially using shape analysis techniques. If we can then ensure that an element has been removed from a list, we would no longer need to flag a warning when a privatized element is accessed nor collapse regions when the element is inserted into another list.

Deciding whether an element has truly been removed from a data-structure is easier said than done. The embedded linked list data structure used, e.g., in operating system code makes this even harder because an element may reside simultaneously in multiple linked lists. In Chapter 6, we describe the problem with *overlapping* structures and present shape analysis techniques for low-level C which can infer whether an element has been removed from such complicated data structures.

1.4 Publications used in this thesis

This thesis is based on the original publications listed on page 7. The papers covered in Chapters 2–4 are co-authored with my supervisors, Prof. Varmo Vene and Prof. Helmut Seidl. For the two most recent papers with additional co-authors, my contribution is clarified below.

Chapter 2 is based on our first paper about the Goblint analyzer [87], presented in 2007 at the Finno-Ugric Symposium on Programming Languages and Software Tools in Hungary with post-proceedings *two years later* for which we updated the paper significantly [88]. For this thesis, the text has been extended with additional background on the side-effecting approach implemented in the Goblint analyzer and discussions of other approaches to race detection for C.

Chapter 3 is based on a paper presented at the Formal Methods Symposium in Eindhoven [78]. It has been extended to include discussion of how it may be implemented in our analyzer which uses a different approach to inter-procedural analysis than the original paper.

Chapter 4 is based on a paper presented at the Static Analysis Symposium in Los Angeles [77]. I have since worked out the correctness proof in more detail, and found that we need to refine our definition of the abstraction relation. This is not a serious flaw because the analysis itself is correct, but the reason it works is slightly more subtle than we originally thought.

Chapter 5 is based on our most recent work with co-authors Kalmer Apinis and Jörg Kreiker. My contribution is the underlying framework for side-effecting with atomic section. Kreiker has worked out the concrete shape instance and Apinis evaluated the technique on real code.

Chapter 6 is based on a paper presented at the Conference on Verification, Model Checking, and Abstract Interpretation in Madrid [52]. Here, Jörg Kreiker is primary author and did the implementation work, while my contribution is limited to the writing; however, our paper is largely motivated by the problems I identified when analyzing the Linux kernel.

CHAPTER 2

GLOBAL INVARIANTS FOR RACE DETECTION

Of the challenges discussed in the previous chapter, the most fundamental is the need for context-sensitive pointer alias analysis. The approach to context-sensitivity influences the entire design of the race detection tool. In this chapter, we discuss the approach used in our analyzer, Goblint, as well as three other solutions used in recent static race detection tools. As stated in the introduction, Goblint is unique among these race detection tools in relying on a sound inter-procedural constant propagation and points-to analysis.

The immediate benefit of building our race detection on top of more traditional analyses is that tracking the values of integer variable allows the analysis of conditional locking schemes and possibly failing locking operations. More importantly, the pointer analysis and the region-based heap abstractions of subsequent chapters rely on the framework for inter-procedural analysis of multi-threaded programs based on partial global invariants explained in this chapter. This is worth keeping in mind as we here illustrate this idea in the simple, though practically less compelling, case of integer variables.

2.1 Programming model

We begin by presenting a concrete semantics of thread interleavings. This will allow us to formalize the notion of a data race and thereby express very clearly what race analysis is supposed to compute. Our programming model is inspired by open systems, such as device drivers, operating within an environment. Each module contains an initialization function where execution of the module begins. This function will eventually register a set of call-back functions and interrupt handlers with the environment which can then call these exported functions when required. From the moment these functions are registered, they can potentially

run in parallel. Thus, in our formal model, we assume that after execution of some initialization code, we have a set of threads $\mathcal{T}$ running in parallel. Since this thesis focuses exclusively on lock-based synchronization, we do not include in our formalism any constructs for spawning new threads, joining with child threads, or any such facilities for threads to wait for each other (except when waiting for a lock to be released.)

The code of each thread is given as a control flow graph (N, E) where each edge (u, s, v) is labelled with an elementary instruction s. These instruction are either assignments, conditional guards, or primitive operations related to locking. We shall gradually extend the language constructs as we proceed through this thesis. We begin with the following grammars for integer and address expressions:

$$e ::= n \mid x \mid *a \qquad\qquad a ::= p \mid \&x \mid \&m$$

An integer-valued expression e is either a constant n, a variable x, or the dereferencing of an address expression a. An address expression is either a pointer variable p, or the address of an integer variable x or a mutex m. The complete grammar for edges in our flow graphs are then defined as follows:

$$\begin{array}{rcll} s & ::= & x := e \mid p := a \mid *p := e & \text{(assignments)} \\ & \mid & \mathsf{true}(e) \mid \mathsf{false}(e) & \text{(conditional guard)} \\ & \mid & \mathsf{lock}(a) \mid \mathsf{unlock}(a) & \text{(locking primitives)} \end{array}$$

This minimal language is sufficient to illustrate the key features and some of the challenges in static race detection.

Assume G is a set of *global* variables shared among threads, while L contains *thread-local* variables. Let $\mathcal{M}$ denote the set of mutexes. We are interested in mappings of global variables $\mathcal{D}_G = G \to \mathbb{N} \cup \mathcal{A}$ where $\mathcal{A} = \mathcal{M} \cup L \cup G$, and for each thread, similar mappings of local variables $\mathcal{D}_L = L \to \mathbb{N} \cup \mathcal{A}$. Here, we only consider two types of values: integers and addresses. As we extend our approach to deal with dynamic memory, it is primarily by means of extending the kinds of address values we consider. In our most simple setting, we do not consider dynamically allocated memory at all: legitimate targets of pointers are mutexes and integer variables.

The state of executing a multi-threaded program is represented in our model as the 4-tuple $d = \langle \vec{u}, \vec{\mu}, \vec{\sigma}, \varphi \rangle$ where $\vec{u} \in \mathcal{T} \to N$ maps each thread to the program point it is about to execute, $\vec{\mu} \in \mathcal{T} \to 2^{\mathcal{M}}$ indicates the set of mutexes held by each thread, $\vec{\sigma} \in \mathcal{T} \to \mathcal{D}_L$ contains the state of local variables for each thread, and $\varphi \in \mathcal{D}_G$ is the state of the shared globals.

In order to evaluate expressions, we need to look up variables in the environment. First, we introduce a convenient notations for "updating" a function, which

we will use throughout this thesis:

$$f[x:n](y) = \begin{cases} n & \text{if } y = x \\ f(y) & \text{otherwise} \end{cases}$$

In addition, we want to apply and update a pair of functions $\langle \sigma, \varphi \rangle$ such that the first component pertains to local variables, while the second relates to globals:

$$\langle \sigma, \varphi \rangle (x) = \begin{cases} \sigma(x) & \text{if } x \in L \\ \varphi(x) & \text{if } x \in G \end{cases} \quad \langle \sigma, \varphi \rangle [x:n] = \begin{cases} \langle \sigma[x:n], \varphi \rangle & \text{if } x \in L \\ \langle \sigma, \varphi[x:n] \rangle & \text{if } x \in G \end{cases}$$

With this machinery, we can give a concrete semantics to the programming model. We begin with an intra-thread semantics for only assignments and guards, and then lift this to an interleaving semantics for multi-threaded execution. For intra-thread execution, the distinction between shared and thread-local variables is not relevant, hence we can define the semantic functions using only a single mapping φ as argument. In the multi-threaded setting, however, these functions operate on the pair $\langle \sigma, \varphi \rangle$ by relying on the above definitions to view the pair as a single mapping. First, we need to evaluate expressions:

$$\begin{aligned} [\![n]\!](\varphi) &= n & [\![p]\!](\varphi) &= \varphi(p) \\ [\![x]\!](\varphi) &= \varphi(x) & [\![\&x]\!](\varphi) &= x \\ [\![*a]\!](\varphi) &= \varphi([\![a]\!](\varphi)) & [\![\&m]\!](\varphi) &= m \end{aligned}$$

Note that this function is partial; for example, dereferencing mutexes is not defined by our semantics because $\varphi(m)$ is undefined. We can now define the effect assignments and conditional guards:

$$\begin{aligned} [\![x \coloneqq e]\!](\varphi) &= \varphi[x : [\![e]\!](\varphi)] \\ [\![p \coloneqq a]\!](\varphi) &= \varphi[p : [\![a]\!](\varphi)] & [\![*p \coloneqq e]\!](\varphi) &= \varphi[\varphi(p) : [\![e]\!](\varphi)] \\ [\![\mathsf{true}(e)]\!](\varphi) &= \varphi \text{ if } [\![e]\!](\varphi) \neq 0 & [\![\mathsf{false}(e)]\!](\varphi) &= \varphi \text{ if } [\![e]\!](\varphi) = 0 \end{aligned}$$

Again we emphasize that this is a partial function; in particular, the conditional guards do not update the program state, rather they make sure flow is propagated to either the true- or the false-branch.

In our formal model, the set of initial configurations D_0 can be characterized as follows. A configuration $d_0 = \langle \vec{u}_0, \vec{\emptyset}, \vec{\sigma}, \varphi_0 \rangle$ is an initial configuration if $\vec{u}_0$ maps each thread to an entry of a function, $\vec{\emptyset}$ maps all threads to the empty lockset, σ is arbitrary since uninitialized locals can take any value, and φ_0 is the global state after running the initialization code.

The transition rules for the interleaving semantics is given in Figure 2.1. Note that despite the grammar specifying commands s, here the locking operations are

$$\textsc{Command}\quad \frac{\langle \vec{u}(i), s, u' \rangle \in E \qquad [\![s]\!]\,\langle \vec{\sigma}(i), \varphi \rangle = \langle \sigma', \varphi' \rangle}{\langle \vec{u}, \vec{\mu}, \vec{\sigma}, \varphi \rangle \to \langle \vec{u}[i : u'], \vec{\mu}, \vec{\sigma}[i : \sigma'], \varphi' \rangle}$$

$$\textsc{Lock}\quad \frac{\langle \vec{u}(i), \mathsf{lock}(a), u' \rangle \in E \qquad [\![a]\!]\,\langle \vec{\sigma}(i), \varphi \rangle = m \qquad m \notin \bigcup_{j \in \mathcal{T}} \vec{\mu}(j)}{\langle \vec{u}, \vec{\mu}, \vec{\sigma}, \varphi \rangle \to \langle \vec{u}[i : u'], \vec{\mu}[i : \vec{\mu}(i) \cup \{m\}], \vec{\sigma}, \varphi \rangle}$$

$$\textsc{Unlock}\quad \frac{\langle \vec{u}(i), \mathsf{unlock}(a), u' \rangle \in E \qquad [\![a]\!]\,\langle \vec{\sigma}(i), \varphi \rangle = m \qquad m \in \vec{\mu}(i)}{\langle \vec{u}, \vec{\mu}, \vec{\sigma}, \varphi \rangle \to \langle \vec{u}[i : u'], \vec{\mu}[i : \vec{\mu}(i) \setminus \{m\}], \vec{\sigma}, \varphi \rangle}$$

Figure 2.1: Interleaving semantics of regular commands and locking operations.

treated separately. For all edges other than (un)locking edges, any thread i which can execute a statement, i.e., there is an outgoing edge $\langle \vec{u}(i), s, u' \rangle$ from its current program counter $\vec{u}(i)$ to some other node u' such that the intra-thread semantics for the edge label s is defined, a transition is made which updates the global state and the local state of the given thread. The most significant transition rule is the locking operation, where the rightmost side condition only permits the acquisition of a lock if no thread already holds it. This ensures non-interleaving execution of sections protected by the same lock. The unlocking transition requires that a thread holds the lock it attempts to release. In Posix C, requesting the reacquisition of a lock one already holds or the release of a lock one does not hold both result in failures.

In order to define a race condition within this formalism, we require the notion of *accessed* global variables. Here, we do not distinguish between read or write accesses. This distinction, though practically important, is neither conceptually interesting nor difficult, but introduces tedious additional notation. Thus, we define a function $[\![s]\!]_{acc} \colon \mathcal{D}_L \times \mathcal{D}_G \to 2^G$ which returns the globals accessed in the statement s. The definitions are straightforward, hence we only give the more interesting cases:

$$[\![\&x]\!]_{acc}(\varphi) = \emptyset \qquad [\![*a]\!]_{acc}(\varphi) = \{[\![a]\!](\varphi)\} \cup [\![a]\!]_{acc}(\varphi)$$
$$[\![p]\!]_{acc}(\varphi) = \{p\} \qquad [\![e_1 := e_2]\!]_{acc}(\varphi) = [\![e_1]\!]_{acc}(\varphi) \cup [\![e_2]\!]_{acc}(\varphi)$$

Note that nothing within an address computation constitutes a memory access, whereas dereferencing an address expression requires both reading the pointer variable as well as accessing the destination. With these definitions in place, we formalize the notion of a race condition as follows.

Definition 2 (Race condition)**.** There is a race in configuration $\langle \vec{u}, \vec{\mu}, \vec{\sigma}, \varphi \rangle$ at variable $x \in G$ if distinct threads i and j may execute edges $\langle \vec{u}(i), s, \_ \rangle \in E$ and $\langle \vec{u}(j), s', \_ \rangle \in E$ both with possible accesses to x, i.e., $x \in [\![s]\!]_{acc} \langle \vec{\sigma}(i), \varphi \rangle$ and $x \in [\![s']\!]_{acc} \langle \vec{\sigma}(j), \varphi \rangle$.

We say there is a race at variable $x \in G$ if there exists a configuration d such that there is a race in configuration d and the configuration is reachable from a starting state, i.e., $\exists d_0 \in D_0 : d_0 \to^* d$. Computing whether there exist races in a program based on the above semantics is naturally infeasible. The number of interleavings grows exponentially with lines of code even for just two threads. In what follows, our goals is to provide efficient analyses which can detect for each variable $x \in G$, and more generally, each memory location, whether there is a race at that memory location.

2.2 Static data race analysis

We formulate the basic algorithm for data race analysis in the framework of abstract interpretation [21]. Facts about the state of the program are represented by elements from a lattice $(\mathbb{D}, \sqsubseteq)$. We conventionally order $x \sqsubseteq y$ if the fact x implies y. Let us assume until Section 2.4 that we do not care about the values stored in shared global variables; we only want to ensure the coherence of whatever is written in shared memory. Then, our abstract domain need only track the values of local variables $\mathbb{D} = L \to 2^{\mathbb{N} \cup \mathcal{A}}$. For each variable, these sets represent the set of all values that the variable may take whenever execution reaches a given program point. Further abstraction is required to effectively analyze programs, but as the specific choice of value domain is currently irrelevant, we illustrate the ideas using the powerset domain.

For each basic statement s, the analysis specification should define a transfer function $[\![s]\!]^\sharp \colon \mathbb{D} \to \mathbb{D}$ describing how the abstract state $\rho \in \mathbb{D}$ is modified by the statement. Although operating on sets of values, the abstract semantics of expressions is similar to the definitions in the concrete semantics. When dereferencing pointers, one has to consider all possible targets of the pointer:

$$[\![n]\!]^\sharp(\rho) = \{n\} \qquad [\![p]\!]^\sharp(\rho) = \rho(p)$$
$$[\![x]\!]^\sharp(\rho) = \rho(x) \qquad [\![\&x]\!]^\sharp(\rho) = \{x\}$$
$$[\![*a]\!]^\sharp(\rho) = \bigcup \{\rho(y) \mid y \in [\![a]\!](\rho)\} \qquad [\![\&m]\!]^\sharp(\rho) = \{m\}$$

The abstract transfer functions for assignments then only differ from the concrete

semantics when it comes to pointer assignments, where we have:

$$[\![*p \coloneqq e]\!]^\sharp(\rho) = \begin{cases} \rho[y : [\![e]\!](\rho)] & \text{if } \rho(p) = \{y\} \\ \rho[y : \rho(y) \cup [\![e]\!](\rho)]_{y \in \rho(p)} & \text{otherwise} \end{cases}$$

where the notation $f[x : n]_{x \in X}$ updates a set of values. The point is that unless we are certain what the pointer target may be, we have to conservatively assume all destinations may have been updated. Thus, we cannot truly update any of the variable, but must retain the old value as well. This is an important concept, called *weak update*, and is a major cause of imprecision in static analysis.

The conditional guards are nearly identical to the concrete semantics, but involve a minor approximation:

$$[\![\mathsf{true}(e)]\!]^\sharp(\rho) = \rho \text{ if } [\![e]\!](\rho) \not\subseteq \{0\} \qquad [\![\mathsf{false}(e)]\!]^\sharp(\rho) = \rho \text{ if } 0 \in [\![e]\!](\rho)$$

However, these functions should not be partial, rather it is more convenient to extend these functions so that when the conditions are not met, the least element of our lattice $\bot$, which maps all variables to the empty value-set, is returned.

For race detection, we are interested in a mapping $\Lambda \colon G \to 2^{\mathcal{M}}$ assigning to each variable $x \in G$ the set of mutexes always held when accessing this variable. To compute this mapping, we need to know for each program point u, the set of locks λ_u definitely held whenever execution reaches that point. Thus, the abstract domain is the powerset of locks $(2^{\mathcal{M}}, \supseteq)$, where the ordering is reversed, i.e., $x \sqsubseteq y$ iff $x \supseteq y$ and the least element $\bot = \mathcal{M}$. The ordering is reversed so that $x \supseteq y$ whenever x implies y: while the set of values that a variable *may* take is ordered by subset inclusion, the set of mutexes that *must* be held at a program point is ordered by superset inclusion.

Finally, let $[\![s]\!]^\sharp_{acc} \in \mathbb{D} \to 2^V$ approximate the set of variables that may be accessed by the statement s. Its definition is identical to the concrete version, except relying on the above abstract definitions to evaluate expressions. We may now specify the transfer functions for the lockset analysis. If the operations that manipulate the lockset only use explicit named locks, the analysis is nearly trivial: a lock operation adds its argument to the lockset and an unlock operation removes it. As we saw in the munge example from Figure 1.1, however, locks and shared variables may be accessed indirectly. Thus, we have the following definitions:

$$[\![\mathsf{lock}(a)]\!]\,\langle \lambda, \rho \rangle = \begin{cases} \langle \lambda \cup \{m\}, \rho \rangle & \text{if } [\![a]\!]^\sharp(\rho) = \{m\} \\ \langle \lambda, \rho \rangle & \text{otherwise} \end{cases}$$

$$[\![\mathsf{unlock}(a)]\!]\,\langle \lambda, \rho \rangle = \langle \lambda \setminus [\![a]\!]^\sharp(\rho), \rho \rangle$$

Consider the first line of the munge function: `lock(m)`. If the target of the pointer variable `m` can be ascertained by our pointer analysis; that is, if we know

the location it *must* point to, then we can add it to the lockset. The next line, `(*v)++`, requires that we constrain the lockset $\Lambda(x)$ for all variables x that `v` *may* point to. The unlock operation, `unlock(m)`, should remove all possible locks that could be pointed to by `m`. To do this computation, we set up the following constraint system:

$$\Lambda(x) \sqsupseteq (x \in [\![s]\!]_{acc}(\rho_u)) \ ? \ \lambda_u : \bot \qquad \text{for } (u, s, v) \in E \text{ and } x \in G$$
$$\langle \lambda_v, \rho_v \rangle \sqsupseteq [\![s]\!]^\sharp \langle \lambda_u, \rho_u \rangle \qquad \text{for } (u, s, v) \in E$$

where the ternary conditional operator $(c) \ ? \ e_1 : e_2$, present in languages like C and Java, is defined as follows:

$$(\mathsf{true}) \ ? \ e_1 : e_2 = e_1 \qquad\qquad (\mathsf{false}) \ ? \ e_1 : e_2 = e_2$$

Thus, the first constraint will, for every possibly accessed variable x, induce the constraint $\Lambda(x) \sqsubseteq \lambda_u$, which in this reversed ordering means $\Lambda(x) \subseteq \lambda_u$. The second constraint requires that the information at a given node takes into account the information from all incoming nodes. Now, let Λ be the least solution to the above constraint system, i.e., the largest locksets that satisfy the constraints. Then, a global variable x is safely accessed if $\Lambda(x) \neq \emptyset$. When the associated lockset is empty, there may be a race and a warning is emitted.

This is naturally a simplification. In practice, we accumulate information about the usage of global variables and determine the races in a separate post-processing phase based on additional information, such as abstract thread identity and whether the access is a read or a write. Thus, $\Lambda(x)$ will return a complete list of all accesses to the variable x with information about the abstract state during the access. This information is then used to determine whether there is a race and to generate sensible error messages pointing out the lines in the source code which may be guilty in producing a data race.

2.3 Analyzing procedures

Recall that it is crucial to propagate information context-sensitively. Assume now that our program comprises a finite set Proc of procedures. Execution starts with a call to the distinguished procedure $main \in \mathsf{Proc}$. Each procedure $q \in \mathsf{Proc}$ is given through a CFG $G_q = (N_q, E_q, e_q, r_q)$ which consists of a set N_q of *program points*; a set of edges $E_q \subseteq N_q \times (\mathsf{Stmt} \cup \mathsf{Proc}) \times N_q$ annotated with assignments or procedure calls; a special *entry point* $e_q \in N_q$; and a special *return point* $r_q \in N_q$. We currently use the so-called functional approach to inter-procedural analysis [79]. For an abstract domain $\mathbb{D}$, consider the function space $\mathbb{F} = \mathbb{D} \to \mathbb{D}$. If we can effectively represent elements in $\mathbb{F}$ and compute compositions and least

upper bounds of these elements, we can compute the effect of a procedure once and for all, and then use this summary as a transfer function for call edges of the summarized procedure.

This computation proceeds thus in two steps. One first requires an abstraction of the effect of running each individual procedure. In the presence of recursive procedures, care must be taken to only consider *same-level* executions, i.e., all recursive calls need to be balanced in the following sense. Let the sequence of instruction π denote an execution trace of the program. In the sequence π, we delimit the instructions executed due to a call of a procedure f within brackets $(_f$ and $)_f$. A same-level execution is an execution trace of the program which can be assembled through the following context-free grammar:

$$\begin{array}{rll} \pi ::= & (_f\ \pi_1\ )_f\ \pi_2 & \text{for each procedure } f \\ \mid & \varepsilon & \end{array}$$

The effect of all same-level runs of a procedure q can be captured as the least solution to the following constraint system.

$$\begin{array}{rll} \mathbf{S}[e_q] \sqsupseteq & \mathsf{Id} & \\ \mathbf{S}[v] \sqsupseteq & [\![s]\!]^{\sharp\sharp} \circ \mathbf{S}[u] & \text{if } (u, s, v) \text{ is an assignment edge} \\ \mathbf{S}[v] \sqsupseteq & \mathbf{S}[r_q] \circ \mathbf{S}[u] & \text{if } (u, q, v) \text{ is a call edge} \end{array}$$

The first constraint makes sure the entry state is at least an identity function, and unless there is a loop back to the entry node, the effect of executing statements from the beginning of the procedure to the entry node is indeed only the identity function. The second constraint then composes the effect of a single edge with the computed summary at the predecessor. Note that for this on needs to specify functional transfer functions for each statement. The third constraint handles function calls, and in this setting the effect of a single function is not of a different type than the effect of a statement. The entire effect of a procedure is accumulated at the return node.

This information can be used to compute the abstract value at a given program point as follows:

$$\begin{array}{rll} \mathbf{R}[e_{main}] \sqsupseteq & d_0 & \\ \mathbf{R}[e_q] \sqsupseteq & \mathbf{R}[u] & \text{if } (u, q, \_) \text{ is a call edge} \\ \mathbf{R}[v] \sqsupseteq & [\![s]\!]^{\sharp}(\mathbf{R}[u]) & \text{if } (u, s, v) \text{ is an assignment edge} \\ \mathbf{R}[v] \sqsupseteq & \mathbf{S}[r_q](\mathbf{R}[u]) & \text{if } (u, q, v) \text{ is a call edge} \end{array}$$

Here, we start with a predefined entry state for the main function. For other procedures, their entry nodes join information from all their call sites. Although

information is joined when pushed into procedure bodies, the analysis is context-sensitive because the summary of the procedure is used to update the state of the callee.

This approach is ideally suited for domains with compact representation of the function space, such as linear equalities, copy-constants, or classical bit-vector analyses. When such representation is not available, Sharir and Pnueli [79] provide a tabulating algorithm that analyzes procedures for each relevant call context, and here we consider this approach as implemented in the Goblint analyzer.

For this, we use a general purpose constraint solver [29], so that we are free to redefine what constitutes a variable in the constraint system. For intra-procedural analysis it suffices to take constraint variables to be the nodes of the control flow graph; however, by attaching some context information to each node, one can effectively compute partial function summaries:

$$\begin{aligned}
\mathbf{R}\langle e_q, d\rangle &\sqsupseteq d && \text{for entry node } e_q \text{ and } d \in \mathbb{D}\\
\mathbf{R}\langle v, d\rangle &\sqsupseteq [\![s]\!](\mathbf{R}\langle u, d\rangle) && \text{for edge } (u, s, v) \text{ and } d \in \mathbb{D}\\
\mathbf{R}\langle v, d\rangle &\sqsupseteq \langle r_f, \mathbf{R}\langle u, d\rangle\rangle && \text{for edge } (u, f(), v) \text{ and } d \in \mathbb{D}
\end{aligned}$$

As there is (at least) one instance of each constraint for each element $d \in \mathbb{D}$, the above system is infinite. However, not all variables in the constraint system need to be solved in order to obtain a meaningful analysis result. As we are interested in the analysis result at the end of program execution, we can use *local* constraint solving to only solve those constraint system variables required to obtain a meaningful analysis result at the end of the program.

In order to deal with local program variables and parameter passing, we rely on the two functions

$$\mathsf{enter}\colon \mathbb{D} \to \mathbb{D} \qquad\qquad \mathsf{combine}\colon \mathbb{D} \times \mathbb{D} \to \mathbb{D}$$

where enter prepares the entry state by initializing the local variables and passing parameters and combine integrates the result of calling the function with the local state of the callee. Thus, the constraint for function call, for an edge $(u, f(), v)$ takes the following form:

$$\mathbf{R}\langle v, d\rangle \sqsupseteq \mathsf{combine}(\mathbf{R}\langle u, d\rangle, \mathbf{R}\langle r_f, \mathsf{enter}(\mathbf{R}\langle u, d\rangle)\rangle)$$

The tabulating approach can be extended to deal with widening/narrowings. More importantly, it can be extended to a multi-threaded setting, as we shall now see.

2.4 Analysing multithreaded C

As we saw by the end of Section 2.2, data race analysis requires information about, at the very least, pointer variables. The difficulty lies in dealing with mul-

```
int global;                          int main() {
void race() { global++; }                spawn(tfun);
void nice() { printf("mu"); }            f = race;
void (*f)() = nice;                      global++;
void tfun() { f(); }                     return 0; }
```

Figure 2.2: A simplified example of storing function pointers in shared data structures.

tithreaded code. Consider the example in Figure 2.2. This is a simplification of a fairly common scenario in low-level code where call-back functions are stored in shared heap data structures. The program starts by spawning a thread which executes the function `tfun`. This thread makes an indirect call through the function pointer `f`. This pointer is at the beginning of the program initialized to the harmless function `nice()`. However, by the time the pointer is dereferenced, the main thread might have already updated it, so that instead the dangerous function `race()` is called. A sound analyzer must assume the worst and take such interleavings of thread execution into account.

The problem, as mentioned in the introduction, is that considering all possible interleavings is computationally unfeasible. Instead, we analyze each thread in isolation by identifying the effect it has on the rest of the program. As the communication between threads in Posix C is through shared memory location, we accumulate the updates to shared memory cells. This information can then be used to analyse each thread in isolation. The updates to shared memory are thus tracked flow-independently, while the analysis of the local state is flow-sensitive.

More concretely, let $\mathbb{D}_L$ and $\mathbb{D}_G$ denote the abstract domains for the local and global states, respectively. In addition to the locksets, the analysis now tracks for each program point u an abstract value $\rho_u \in \mathbb{D}_L$ and a single global invariant $\Psi \in \mathbb{D}_G$ which over-approximates the state of the shared memory for the entire program execution. The set of variables accessed by a statement now depends on the state of the program, and the transfer function also operate these abstract values:

$$[\![s]\!]_{acc} \colon \mathbb{D}_L \times \mathbb{D}_G \to 2^G \qquad [\![s]\!]^\sharp \colon 2^{\mathcal{M}} \times \mathbb{D}_L \times \mathbb{D}_G \to 2^{\mathcal{M}} \times \mathbb{D}_L \times \mathbb{D}_G$$

We then seek the least solution to the following constraint system:

$$\begin{aligned} \Lambda(x) &\sqsupseteq (x \in [\![s]\!]^\sharp_{acc} \langle \rho_u, \Psi \rangle) \ ? \ \lambda_u : \bot && \text{for } (u, s, v) \in E \text{ and } x \in G \\ \langle \lambda_v, \rho_v, \Psi \rangle &\sqsupseteq [\![s]\!]^\sharp \langle \lambda_u, \rho_v, \Psi \rangle && \text{for } (u, s, v) \in E \end{aligned}$$

This constraint system is intra-procedural, and it can be solved with any iterative fix-point solver. Taking procedures into account is somewhat tricky for consider the following procedures which read and write to a global x:

```
void write(int p) { x = p; }
int read() { return x; }
```

The effect of the write function depends on the context in which it is called, whereas the summary for the read function must take into account the value of the global variable. Therefore, the tabulating constraint system cannot really be written in the simple form:

$$\langle \mathbf{R}\langle v, d\rangle, \Psi\rangle \sqsupseteq [\![s]\!]^\sharp \langle \mathbf{R}\langle u, d\rangle, \Psi\rangle \qquad \text{for edge } (u, s, v) \text{ and } d \in \mathbb{D}$$

This would imply the global invariant should be constrained by all possible entry states in $\mathbb{D}$. We would prefer to only constrain the global invariant based on those call contexts reached during the partial tabulation approach. This is the essence of *partial global invariants*, which was developed for the Trier analyzer [76], the grandfather of Goblint.

In general, solving systems with partial global invariants require finding a nested fix-point: one has to solve a system of control flow equations for the local state, but whenever the global state changes, one must recompute the analysis with respect to a new global invariant. However, when the global state is such that it maps each global variable to a certain abstract value, we can use a more efficient algorithm that tracks dependencies between globals and the nodes that use them. Thus, we re-evaluate as few nodes as possible, while still remaining sound.

In the example, we would start by analysing the main function, but as a thread is spawned, the solver first looks into the execution of the thread code. This is initially analysed such that only the call to `nice()` is considered, but the solver records that the call depends on the value of the global function pointer. As the solver returns to the analysis of the main function, the pointer is updated. This triggers the re-evaluation of all nodes depending on the variable `f`. Since it may now also point to `race()`, this function is therefore analyzed as well and the result of its call is joined with the previously analyzed function.

2.5 Three other approaches

Before discussing our approach to conditional locking, it is worth exploring other solutions to the problem of context-sensitive pointer analysis. This section consists of a detailed technical review of three existing systems. First, we consider the type-based label flow used by LOCKSMITH. Then, we turn to bootstrapping to speed up pointer analysis in the CoBE framework. Finally, we look at relative locksets which enable RELAY to scale to millions of lines of code.

2.5.1 Type-based flow analysis

The LOCKSMITH analyzer [68] annotates the program with a type and effect system which computes everything needed for sound race detection. Type-based program analysis is an interesting topic which is extensively studied in its own right. Formulating an analysis as a type system allows the application of many techniques from type theory to reason about properties of the analysis. For race detection, *polymorphism* allows the context-sensitive propagation of points-to information into functions, such as `munge` in Figure 1.1, without needing to clone the body of the function [28]. Instead, a polymorphic function can be given a parametric type which is instantiated at each call site. As type systems are formulated for languages with higher-order functions, context-sensitive handling of function pointers comes for free.

The general approach of the LOCKSMITH analyzer is to accumulate access-lock correlation constraints. The analyzer collects for each access to a memory location ρ with the set of held locks L a *correlation constraint* $\rho \triangleright L$. Due to indirect accesses via pointers and parameter passing, other forms of constraints are also required, as will be explained below. Given a set of constraints C, the notation $C \vdash \rho \triangleright L$ indicates that the correlation $\rho \triangleright L$ can be derived from the constraints in C. The set $S(C, \rho) = \{L \mid C \vdash \rho \triangleright L\}$ denotes the set of all locksets that were held when accessing ρ. The location ρ is safely protected by a mutex, whenever the intersection of all locksets is non-empty: $\bigcap S(C, \rho) \neq \emptyset$. The accessed data is then said to be *consistently correlated* with the lockset. For the example program of the introduction (Figure 1.1), the following correlation constraints are inferred:

`t1` :	$\rho_x \triangleright \{m_1\}$	$\rho_y \triangleright \{m_2\}$	$\rho_z \triangleright \{m_2\}$
`t2` :	$\rho_x \triangleright \{m_1\}$	$\rho_y \triangleright \{m_1\}$	$\rho_z \triangleright \{m_2\}$

The locations ρ_x and ρ_z are consistently correlated with the locks m_1 and m_2, respectively. As the intersection for ρ_y is empty, LOCKSMITH reports a race on the variable `y`. In order to obtain such information, one has to compute the set of held locks, generate all the constraints, and solve them.

As the focus of this survey is on the context-sensitive propagation of pointer information, we will first briefly discuss a few other features of the analyzer. The set of *definitely* held locks are computed flow-sensitively, meaning the control flow of the program is taken seriously; in contrast, pointer information is propagated flow-insensitively, meaning the analysis computes an over-approximation of all assignments within the body of a function independent of the order in which the assignments may be executed. Consider the following example:

```
void f() { int *p;
    p = &x; lock(&m1); *p = 41; unlock(&m1);
    p = &y; lock(&m2); *p = 42; unlock(&m2); }
```

LOCKSMITH will infer that `p` may point to either `x` or `y` all over this function. On the other hand, it computes the set of locks for each program point: when assigning 41 to `x` it is $\{m_1\}$, and when assigning 42 to `y` it is $\{m_2\}$. Due to flow-insensitivity, LOCKSMITH will infer the false constraints $\rho_x \triangleright \{m_2\}$ and $\rho_y \triangleright \{m_1\}$, which lead to a false alarm being reported. However, the coding style in this example, traversing the same pointer over different stack-allocated variables, is not that common, hence it is becoming increasingly popular in static analysis to ignore the flow within functions.

The flow-sensitive computation of the locksets is essentially achieved through a data flow analysis over the control flow graph of the program. In the type-based approach this amounts to using *state variables* to achieve flow-sensitive analysis. This allows the use of instantiation constraints for context-sensitive propagation of locksets, and there is additional cleverness with respect to function calls.

The flow-insensitive propagation of pointers within a function is achieved through *sub-typing* [69]. The idea is that each location has a type which associates it with a location label ρ; for example, the type of `&x` is $ref^{\rho_x}(int)$, a cell ρ_x containing an integer. Whenever there is a read or write to a variable of type $ref^{\rho}(\tau)$, one generates the constraint $\rho \triangleright L$ where L is the current lockset. The question is how to deal with indirect accesses through pointers.

Figure 2.3 contains the relevant rules for intra-procedural pointer analysis. These are here "translated" to C syntax, while the original paper develops a typed lambda-calculus. The auxiliary function *loc* assumes that addresses are assigned unique labels, whereas in the original setting locations are introduced at let-binding. The rule for assignment states that a value of type τ can be stored into a memory cell of the same type $ref^{\rho}(\tau)$. Thus, in order to type the statements `p = &x; p = &y;`, we need `&p` to be of type $ref^{\rho_p}(\tau)$ where τ is equal to $ref^{\rho_x}(int)$ as well as $ref^{\rho_y}(int)$. This is only possible using the sub-typing rules, with which we can give the type $C; \Gamma \vdash \texttt{\&p} : ref^{\rho_p}(ref^{\rho_{xy}}(int))$ if $C \vdash \rho_x \leq \rho_{xy}$ and $C \vdash \rho_y \leq \rho_{xy}$ for a freshly generated location label ρ_{xy}. Inferring the type of the program thus requires that we *generate* such constraints. Then, these constraints are resolved using resolution rules such as the following:

$$C \cup \{\rho \leq \rho'\} \cup \{\rho' \triangleright L\} \quad \cup\!\Rightarrow \quad \{\rho \triangleright L\}$$

where $X \cup\!\Rightarrow Y$ is short-hand for $X \Rightarrow X \cup Y$. This rule propagates an access through a pointer to all its sub-types, so that from $\{\rho_{xy} \triangleright L,\ \rho_x \leq \rho_{xy},\ \rho_y \leq \rho_{xy}\}$, we also have $\rho_x \triangleright L$ and $\rho_y \triangleright L$.

$$\textsc{Ref}\ \frac{C;\Gamma \vdash e : \tau \qquad \rho = loc(\Gamma, e)}{C;\Gamma \vdash \&e : ref^{\rho}(\tau)} \qquad \textsc{Assign}\ \frac{C;\Gamma \vdash \&e_1 : ref^{\rho}(\tau) \qquad C;\Gamma \vdash e_2 : \tau}{C;\Gamma \vdash e_1 := e_2 : \tau}$$

$$\textsc{Seq}\ \frac{C;\Gamma \vdash e_1 : \tau_1 \qquad C;\Gamma \vdash e_2 : \tau_2}{C;\Gamma \vdash e_1; e_2 : \tau_2} \qquad \textsc{Sub}\ \frac{C;\Gamma \vdash e : \tau_1 \qquad C \vdash \tau_1 \leq \tau_2}{C;\Gamma \vdash e : \tau_2}$$

$$\textsc{Sub-Ref}\ \frac{C \vdash \rho \leq \rho' \qquad C \vdash \tau \leq \tau' \qquad C \vdash \tau' \leq \tau}{C \vdash ref^{\rho}(\tau) \leq ref^{\rho'}(\tau')}$$

Figure 2.3: Selection of monomorphic (intra-procedural) typing rules.

We now consider how adding polymorphism to the type system enables the context-sensitive treatment of function calls. The traditional approach to polymorphic type inference universally quantifies all type variables that do not occur freely in the environment when a function is defined. Polymorphic types are then instantiated at each usage site by generating fresh variables and substituting in the type all occurrences of universally quantified types with the newly generated ones. In the context of constraint-based type inference, this involves the copying of the constraint sets, which can be quite large. Furthermore, copying the set of constraints for each call site would not constitute a significant gain over the brute force approach of analyzing a separate copy of the function at each call. Instead, the flow of parameters into and out of a function can be captured as *instantiation constraints* $\tau_1 \preceq^i_p \tau_2$ where p is the polarity (direction of flow) and i is the unique identifier for each call-site. Note that this is a true instantiation in the sense that for each call site, there must exist a substitution ϕ_i such that $\phi_i(\tau_1) = \tau_2$; additionally, it expresses flow of information through the use of polarities.

It may be helpful to look at the two critical constraint resolution rules to understand the use of instantiation constraints.

$$C \cup \{\rho_1 \preceq^i_- \rho_0\} \cup \{\rho_1 \leq \rho_2\} \cup \{\rho_2 \preceq^i_+ \rho_3\} \quad \cup\Rightarrow \quad \{\rho_0 \leq \rho_3\} \tag{2.1}$$

$$C \cup \{\rho \preceq^i_p \rho'\} \cup \{\rho \triangleright L\} \cup \{L \preceq^i L'\} \quad \cup\Rightarrow \quad \{\rho' \triangleright L'\} \tag{2.2}$$

The first rule propagates flow information in and out of a function, while the second propagates location and lock information into the function such that correlations constraints within the function are related to the values of parameters that went in. Locks do not need polarities because locks are unified as soon as there is flow between two lock labels. Polarities must take care of the flow in the presence

of higher-order functions. Consider first the case where we have the following definitions:[1]

```
int *bar () { return &x; }
int  foo (int *(*fp)()) { return *fp(); }
```

When we now apply `foo(bar)`, we need to register that the location ρ_x has flowed into the function `foo` from the parameter `bar`. On the other hand, for the following definitions we have flow in the opposite direction:

```
int bar (int *p) { return *p; }
int foo (int (*fp)(int *)) { return fp(&x); }
```

Here, when we apply `foo(bar)`, the location ρ_x flows from `foo` into `bar`. We can summarize the two cases in a table and then generalize.

type of foo	type of bar	constraint
$\forall\rho.(void \to ref^{\rho}(int)) \to int$	$void \to ref^{\rho_x}(int)$	$\rho \preceq^i_- \rho_x$
$(ref^{\rho_x}(int) \to int) \to int$	$\forall\rho.ref^{\rho}(int) \to int$	$\rho_x \preceq^i_+ \rho$

In general, we obtain constraints between the quantified variables in the function and the type variables generated for each application instance by positively correlating the types as shown in the first rule below, and then we propagate the instantiation constraints by flipping the polarity of the function argument, denoted with $\bar{p}$, according to the second rule:

$$\textsc{Inst}\quad \frac{C \vdash \tau \preceq_+ \tau'}{C; \Gamma, f : \forall\vec{l}.\tau \vdash f^i : \tau'} \qquad \textsc{Inst-Fun}\quad \frac{C \vdash \tau_1 \preceq^i_{\bar{p}} \tau_2 \qquad C \vdash \tau_1' \preceq^i_p \tau_2'}{C \vdash \tau_1 \to \tau_1' \preceq^i_p \tau_2 \to \tau_2'}$$

The rule for instantiation is simplified to ignore the free variables that could not be universally quantified when the function was defined.

We now return to the motivating example of Figure 1.1. The type of the `munge()` function is $C; \Gamma \vdash \forall\rho_v, m\,.\,ref^{\rho_v}(int) \times lock(m) \to void$ where $C \vdash \rho_v \triangleright \{m\}$. We consider the instantiation constraints generated in order to type thread `t1`:

$$ref^{\rho_v}(int) \times lock(m) \to void \preceq^1_+ ref^{\rho_x}(int) \times lock(m_1) \to void$$
$$ref^{\rho_v}(int) \times lock(m) \to void \preceq^2_+ ref^{\rho_y}(int) \times lock(m_2) \to void$$
$$ref^{\rho_v}(int) \times lock(m) \to void \preceq^3_+ ref^{\rho_z}(int) \times lock(m_2) \to void$$

We resolve the first one to $ref^{\rho_v}(int) \times lock(m) \preceq^1_- ref^{\rho_x}(int) \times lock(m_1)$ which (by obvious rules omitted here) simplifies to $\{\rho_v \preceq^1_- \rho_x,\ m \preceq^1 m_1\}$. When this

[1] Definitions in C are read from the inside out; `()` has a higher priority than `*`. The parameter `int *(*fp)()` is a pointer to – a function that returns – a pointer to – an integer.

is conjoined with the correlation constraint $\rho_v \triangleright \{m\}$, the constraint resolution rule (2) allows us to infer $C \vdash \rho_x \triangleright \{m_1\}$. Analogously, we obtain all the other constraints required to check for races.

This is *almost* the whole story; a very prominent feature of the type system has been deliberately ignored in this presentation and eradicated from the typing rules. LOCKSMITH uses an effect system to enforce linearity among locks. This is required for sound analysis because a dynamically allocated lock might be re-allocated:

```
mutex *m = malloc(); mutex_init(m);
lock(m); x++; unlock(m);
m = malloc(); mutex_init(m);
lock(m); x++; unlock(m);
```

Here, the lock pointer m refers to multiple locks. A sound analyzer should consider this, even if it rarely occurs in real programs. Unfortunately, LOCKSMITH does not terminate on certain device drivers with linearity turned on. The authors claim this could be fixed with improvements to their unification algorithm.

2.5.2 Bootstrapping pointer analyses

Kahlon et. al [47] present a technique for fast must-alias analysis of lock pointers and a shared variable discovery algorithm. Unfortunately, it is unclear how the ideas presented in that paper apply to the example of Fig 1.1, where may-aliasing of pointers to shared variables and must-aliasing of lock pointers are to be jointly propagated context-sensitively.

The general approach is to first identify shared variables and the location where these shared variables are accessed. If the same shared variable can be accessed by two different threads simultaneously and the set of locks they hold are disjoint, a race warning is emitted. The second step is, therefore, to identify the set of held locks. Here, a sophisticated must-alias analysis is proposed based on bootstrapping and procedure summarization. Finally, warning reduction techniques are applied.

The suggested method for shared variable discovery is somewhat puzzling. The idea seems to be that one should conservatively consider all global variables and pointers passed to external functions as shared. Thus much makes sense, but in order to deal with local aliases to shared data, aliased pointers are also considered as shared (with the minor refinement that only those pointers are added which are instrumental in resulting in a true access rather than just propagating address information.) This is surprising because when there is an indirect access to a shared variable through a pointer, we would expect an attempt to resolve the pointer, rather than register the access with the pointer and then ensure that

accesses to the pointer are safe.

The must-alias analysis of pointers is based on the idea of bootstrapping alias analyses [44]. This is an approach marketed by Vineet Kahlon to "leverage the combination of *divide and conquer*, *parallelization* and *function summarization*." The key idea is to use a succession of alias analyses of increasing precision such that the rough partitioning of the first alias analysis allows the more precise ones to run on a much smaller problem instances. Thus, each subsequent analysis is "bootstrapped" by the previous one. This only works if one can prove that the equivalence classes computed by the more coarse-grained analyses and the slices of programs that one considers for each cluster suffice to correctly compute the refined alias information at the next stage.

One suitable pointer analysis to begin the bootstrapping process is Steensgaard's alias analysis [83]. This is nearly identical to the flow-insensitive pointer analysis from Section 2.5.1, except instead of introducing sub-typing when a pointer `p` may refer to two distinct locations, the locations are unified, i.e., considered as a single abstract location. This results in a partitioning of pointers into equivalence classes. Practically, the partitioning means that one only need to deal with a single equivalence class at a time, and for locks this typically involves 2–3 pointers.

Having clustered the set of pointers, a more expensive must-alias analysis can be applied. The analysis proposed in the article is based on *Maximally Complete Update Sequences* which can be used to characterize aliasing. These can then be used to compute a procedure summary, as discussed in Section 2.3. Recall that a summary is computed once and for all and then applied whenever the function is called, achieving context-sensitivity without the cost of cloning, as was previously achieved through polymorphism. We begin with the notion of update sequences which are central to this approach. The goal is to characterize must-aliasing in terms of chains of assignment: two pointers p and q must alias precisely when there exists some location a and chains of assignments π_1 and π_2 that are semantically equivalent to $p = a$ and $q = a$. This idea is formalized as follows.

Definition 3 (Complete Update Sequence [44])**.** Let $\Pi\colon u_0, \ldots, u_m$ be a sequence of successive program points and let π be the following sequence of pointer assignments along Π:

$$u_{i_1}\colon p_1 = a_0; \qquad u_{i_2}\colon p_2 = a_1; \qquad \ldots \qquad u_{i_k}\colon p_k = a_{k-1};$$

Then π is called a complete update sequence from p to q leading from program points u_0 to u_m if the following conditions hold: a_0 is semantically equivalent to (i.e., evaluate to the same value as) p at program point u_0; p_k is semantically equivalent to q at u_0; for each j, a_j is semantically equivalent to p_j at u_{i_j}; and for each j, there does not exist any (semantic) assignment to pointer a_j between

```
1 int main() {
2     int *a, *b, *c, **x, **y;
3     b = c;
4     x = &a;
5     y = &b;
6     *x = b; }
```

Figure 2.4: Complete complete update sequence.

locations u_{i_j} and $u_{i_{j+1}}$, to a_0 between u_0 and u_{i_1}, and to p_k between u_{i_k} and u_m along Π.

Thus, a complete update sequence from p to q (leading from u_0 to u_m) means that executing the code snippet between u_0 and u_m has an effect on q which is equivalent to performing the assignment $q = p$ at location u_m.

Consider the example shown in Figure 2.4 to illustrate the idea. We can see that line 6 by itself is a complete update sequence from b to a (leading from 3 to 6) because executing the snippet will result in assigning b to a though the indirect assignment on line 4. Note that the sequence is effectively equivalent to $a = b$ being performed at the end, but the single assignment does not adequately capture the effect of these lines on the variable a because b has obtained its value from c in the first line, which we have ignored. In contrast, the update sequence $3, 6$ is also a complete update sequence leading from 3 to 6, but it is from c to a. This update sequence really captures what happens to the pointer a when executing lines 3 to 6. Such update sequences are called *maximally complete*.

Formally, the maximally complete update sequence for a pointer q leading from location u_0 to u_m along Π is the complete update sequence π of maximum length, over all pointers p, from p to q (leading from locations u_0 to u_m) occurring along Π. We can now characterize aliasing as follows: pointers p and q must alias at program point u if and only if there exists a pointer a with maximally complete update sequences to both p and q. Since the goal is to obtain an efficient summary of a procedure's effect on aliasing, the summaries track maximally complete update sequences.

The summary of a function f is a set of triples of the form (p, u, A), where p is the pointer of interest, u is an (important) program point, and A is the set of all pointers q such that there is a complete update sequence from q to p along every path leading from the entry of the function to the program point u. This summary is computed through a process that is reminiscent of weakest pre-condition computation: we start with the summary $(p, u, \{p\})$ and work backwards in the control flow graph so that an assignment $p = q$ has the effect of replacing p with

q in the set A giving us $(p, u, \{q\})$. When we reach the entry point of the function we have computed the summary for the aliases of p at location u. The effect of applying a function on the pointer p is the summary for that pointer at the exit location. Thus, when the analysis needs to consider the effect of a function call, it looks up the summary for each pointer in A. It would be interesting to apply the algorithm to the example in Figure 2.4, but unfortunately the algorithm, as described in the paper, does not consider the case of indirect updates.

A more serious problem is that this approach does not handle the running example from the introduction: although pointers are updated context-sensitively at each call site, the different calling contexts are not distinguished within the body of the function. The may-aliasing of shared variables are not really discussed in the paper. In a more recent paper, Kahlon et al. [46] propose a Context Sensitive Call-Graph construction, but in order to deal with our motivating example, the must- and may-alias information need to be propagated *into* functions to infer correlations context-sensitively. In the next section, we will study a summarization approach which achieves this goal by summarizing locksets together with *guarded accesses*.

2.5.3 Relative locksets

The RELAY analyzer [89] provides a very simple and elegant solution to the problem of context-sensitive pointer analysis. It relies on the concept of a *relative lockset* to describe the changes in the locksets relative to the function entry point. It also accumulates accessed memory locations relative to the entry point, thus obtaining a set of guarded accesses which are expressed in relation to the parameters of the function. This summarizes the effect of the function, which is then used at call sites by plugging in the values of the parameters at any given calling context. Consider the following function.

```
void foo(struct node *x) {
    lock(&x→mtx1); x→f = 7; unlock(&x→mtx2); }
```

Its summary would consist of two components: first, the relative lockset at the end of the call, which is obtained by adding `x→mtx1` and removing `x→mtx2` from the set of mutexes of the caller; and second, a list of relative accesses, which in this case is only the access to `x→f` together with the relative lockset at access, again obtained by adding `x→mtx1` to set of mutexes of the caller.

Functions are processed bottom-up in the call graph. Any function whose callees have been summarized can be analyzed in separation; this allows parallelization of the analysis. For each function, three analyses are performed: symbolic execution, relative lockset analysis, and guarded access analysis. The foundation for the other analyses is laid by the symbolic execution which aims to

express the values of program variables in terms of the "incoming" values of the function's parameters. The analysis tracks for each program point a symbolic map $\Sigma : \mathbb{O} \to \mathbb{V}$ from symbolic L-values to symbolic R-values defined as follows.

$$o ::= x \mid p \mid o.f \mid *o \qquad \text{L-values}$$
$$v ::= \bot \mid \top \mid i \mid \mathit{init}(o) \mid \mathit{may}\{o_1, o_2, \ldots\} \qquad \text{R-values}$$

R-Values include $\bot$ (unassigned), $\top$ (unknown), integers, the incoming (initial) value of some L-Value, and a may-points-to set of L-Values. For the symbolic L-Values, the meta-variable x ranges over program variables, while p ranges over representative summary nodes computed by an external flow-insensitive (hence very fast) points-to analysis. This is primarily to ensure termination. Consider the following example:

```
void foo(struct node *x) {
    struct list *y = x→first;
    while (y) { y→data = 5; y = y→next; } }
```

Before executing the loop the mapping is $[\mathtt{x} \mapsto \mathit{init}(\mathtt{x}),\ \mathtt{y} \mapsto \mathit{may}\{\mathtt{x}{\to}\mathtt{first}\}]$, where we use $o{\to}f$, which is not part of our language describing L-values, as a synonym for $(*o).f$. After executing the loop, the pointer p may now point to a possibly infinite set $\{\mathtt{x}{\to}\mathtt{first}, \mathtt{x}{\to}\mathtt{first}{\to}\mathtt{next}, \mathtt{x}{\to}\mathtt{first}{\to}\mathtt{next}{\to}\mathtt{next}, \ldots\}$, which one may want to simply replace with a single summary node. The main idea of relative locksets is independent of the precise symbolic execution analysis.

The second step is the computation of relative locksets using the information from the symbolic execution to evaluate lock expressions. A relative lockset L is the pair (L_+, L_-) of definitely acquired and possibly released locks since the beginning of the function. The ordering is as expected with a must- and may-set:

$$(L_+, L_-) \sqsubseteq (L'_+, L'_-) \iff L_+ \supseteq L'_+ \text{ and } L_- \subseteq L'_-$$

The relative lockset at the exit of a function f is the summary lockset L_f which is used whenever the function is invoked. The analysis treats everything as function calls with `lock(&l)` and `unlock(&l)` being simulated as functions with summaries $(\{l\}, \emptyset)$ and $(\emptyset, \{l\})$, respectively. Summaries are applied according to the following scheme: rebind the formals in L_f to the values of the arguments computed by the symbolic execution and update the lockset before the call L with the effect of the summary L_f. The effect of a call $f(a)$, where a is the argument list and p is the list of formal parameters of f, is computed as $\mathit{update}(L, L_f[p \mapsto [\![a]\!](\Sigma)])$ using the symbolic map Σ and the function

$$\mathit{update}((L_+, L_-), (L'_+, L'_-)) = ((L_+ \cup L'_+) \setminus L'_-,\ (L_- \cup L'_-) \setminus L'_+)$$

This updates the lockset by adding the effect of the summary and then removing any lock that may have been released from the set of definitely held locks as well as removing any lock that the called function definitely ends up holding from the set of released locks.

Finally, the set of guarded accesses are computed for each function. A guarded access is a triple $a = (o, L, k)$, where $o \in \mathbb{O}$ is an L-value being accessed, L is the relative lockset and k indicates whether the access was a read or a write. The propagation of guarded accesses by using summaries is very similar to the relative locksets, although computing the set of accesses does not need to be flow-sensitive, but can simply traverse assignments in any order.

Let us return to the example program from the introduction. The guarded access for `munge` is $\{(*\mathtt{v},\ (\{\mathtt{m}\}, \emptyset), \text{write})\}$ which at each instantiation is rebound to the arguments and the lockset is updated with the relative lockset of the caller, which in this case is empty. Instantiating the arguments requires resolving the may-points-to sets, e.g., $[\![*\mathtt{v}]\!](\Sigma) = \mathtt{x}$ when the argument is `&x` because then $\Sigma(v) = may\{\mathtt{x}\}$. This is a singleton points-to set, but in principle, a guarded access of a function may need to be instantiated to multiple accesses if the caller gave an ambiguous pointer as parameter. In our simple example, we obtain for `t1` and `t2`:

$$\mathtt{t1}: \quad \{(\mathtt{x},\ (\{\mathtt{m1}\}, \emptyset),\ \text{write}),\ (\mathtt{y},\ (\{\mathtt{m2}\}, \emptyset),\ \text{write}),\ (\mathtt{z},\ (\{\mathtt{m2}\}, \emptyset),\ \text{write})\}$$
$$\mathtt{t2}: \quad \{(\mathtt{x},\ (\{\mathtt{m1}\}, \emptyset),\ \text{write}),\ (\mathtt{y},\ (\{\mathtt{m1}\}, \emptyset),\ \text{write}),\ (\mathtt{z},\ (\{\mathtt{m2}\}, \emptyset),\ \text{write})\}$$

Race warnings are generated by considering pairs of thread entry points (here there is only one such pair: `t1` and `t2`) and identifying whether there exists a pair of accesses a_1 and a_2 that conflict, i.e., their L-values alias, but the locksets are disjoint (and at least one is a write). In the example, there is a conflict between accesses $\{(\mathtt{y},\ (\{\mathtt{m2}\}, \emptyset),\ \text{write})$ and $\{(\mathtt{y},\ (\{\mathtt{m1}\}, \emptyset),\ \text{write})$, hence the correct warning is flagged.

2.6 Path-sensitive data race analysis

The techniques of the previous section preform race detection without caring deeply about the value of integer variables. In this section, we give one good reason to care: path-sensitivity. This feature is needed to deal with possibly failing locking operations, non-blocking primitives, as well as conditional locking schemes. In general, path-sensitivity is the ability to distinguish between real executable paths in the control flow graph and imaginary paths that are unreachable due to logical constraints. The `gcc` manual has the following example, where it states that "GCC is not smart enough" to see that the code is bug free:

```
int save_y;
if (change_y) save_y = y, y = new_y;
...
if (change_y) y = save_y;
```

Being "smart enough" would in this case require detecting the relationship between the conditional guards. Among the four possible paths in the CFG, one should only analyse the two logically possible paths.

For race detection, path-sensitivity is important because we cannot assume that Posix locking operations will always succeed. The standard practice in POSIX threaded C is to always check the returning value of the locking function:

```
status = lock(m);
if (status != 0)
    err_abort(status, "Lock mutex");
```

Sound and precise analysis of this situation requires that we be path-sensitive, so that when the user checks the `status` variable, the body of the conditionals are analyzed with the correct lockset. We will now consider how to be path-sensitive in an efficient way. Consider the simplified example in Figure 2.5, which will raise a false alarm on line 6, unless the analysis is path-sensitive.

The problem, again, is that there are now eight potential paths, but only two logically possible paths, and we must eliminate the false paths. We want to do this inter-procedurally, and without sacrificing the soundness of the analysis. Engler and Ashcraft [26] propose what they call "unlockset analysis" to achieve path-sensitive analysis of locked mutexes. This was needed to deal with the complicated control flow in FreeBSD code and although practical, it is an ad-hoc solution that does not aim to be sound.

```
1 void foo (int do_work) {
2     if (do_work)
3         lock(&mtx);
4     ...
5     if (do_work)
6         work++;
7     ...
8     if (do_work)
9         unlock(&mtx); }
```

Figure 2.5: Example of conditional locking.

In order to treat this in a sound way, we could use a powerset domain $2^{\mathbb{D}}$, but this is not feasible since the constant propagation domain is infinite. One could limit the number of paths that are distinguished with some arbitrary constant, and just merge any further branching. Early experiments showed that this is not a feasible approach either, and one really must decide which paths to distinguish.

We denoted with $\mathbb{D}_L$ the domain used by our base analysis and with $2^{\mathcal{M}}$ the lockset domain. If we combine them trivially using the product domain $\mathbb{D}_L \times 2^{\mathcal{M}}$, then information is lost, and the eight paths are mixed together. In order to discriminate different paths, we need to track the state of the conditional variables on lines 2, 5 and 8 because their correlation logically prohibits all but the two real executions (assuming the omitted code does not modify the `do_work` variable).

The immediate idea would be to track the lockset for each value of the conditional variables $\mathbb{D}_L \to 2^{\mathcal{M}}$, but since the base domain is essentially infinite, this is no improvement over the powerset domain. Instead, the domain $2^{\mathcal{M}} \to \mathbb{D}_L$ yields as much precision as needed to solve the problem at hand without creating a potentially infinite domain (assuming the sets of locks used by the program are finitely represented). As one simulates the execution of the program for each relevant set of facts, this approach has been called *property simulation* [23].

The analyzer can use this information to distinguish the paths by performing a Conditional Constant Propagation [91]: when reaching conditionals with an unknown guard (line 2), the true-branch is analyzed assuming the conditional is true resulting in the domain element $[\{\&\texttt{mtx}\} \mapsto [\texttt{do\_work} \mapsto \mathsf{true}]]$ and assuming the opposite for the false-branch, which in the current example trivially results in $[\emptyset \mapsto [\texttt{do\_work} \mapsto \mathsf{false}]]$. By merging in the domain $2^{\mathcal{M}} \to \mathbb{D}_L$ we keep the states separated when exiting a branch, if they contain different locksets, as is currently the case:

$$[\emptyset \mapsto [\texttt{do\_work} \mapsto \mathsf{false}],\ \{\&\texttt{mtx}\} \mapsto [\texttt{do\_work} \mapsto \mathsf{true}]]$$

When the relevant states are kept separate, subsequent conditional guards can be evaluated as constants and the wrong paths will be considered dead code for the states with irrelevant locksets. Thus, when analyzing line 5, the true-branch is considered dead code in the state $[\emptyset \mapsto [\texttt{do\_work} \mapsto \mathsf{false}]]$, and line 6 is only analyzed with the correct mutex set. The false alarm is therefore avoided.

The integer value domain we use in our analyzer is specifically designed to deal with heavily branching code. It is similar to the commonly used Kildall domain, but it is topped by finite exclusion sets rather than a single unknown. Thus, it can embed the boolean domain (it can express true as the exclusion set containing zero) and it can also express some other conditions that are useful when analysing `switch`-constructions.

More formally, values in this abstract domain $\mathbb{D} = \{\bot, \top\} \cup \mathbb{Z} \cup 2^{\mathbb{Z}}$ are either top/bottom, a known specific number, or an exclusion set. These elements

are ordered in the way that respects the following concretization to the powerset domain $2^{\mathbb{Z}}$:

$$\gamma(\top) = \mathbb{Z} \qquad \gamma(\bot) = \emptyset$$
$$\gamma(n) = \{n\} \qquad \gamma(X) = \mathbb{Z} \setminus X$$

Note that we use no special notation for the sets in the abstract domain, although these are interpreted as exclusion sets, i.e., their concretization is the set of all integers that excludes the given set. The least upper bound is defined as follows for the non-trivial cases:

$$X \sqcup n = X \setminus \{n\}$$
$$X_1 \sqcup X_2 = X_1 \cap X_2$$
$$n_1 \sqcup n_2 = \begin{cases} n_1 & \text{if } n_1 = n_2 \\ \top & \text{if } n_1 = 0 \text{ or } n_2 = 0 \\ \{0\} & \text{otherwise} \end{cases}$$

The special treatment of zero is to support the embedding of the boolean domain. Although this domain contains infinite descending chains, these exclusion sets only come into play when evaluating boolean guards hence the analysis always terminates.

2.7 The Goblint analyzer

We now have a brief look at Goblint, the static analyzer where most of the ideas of this thesis have been implemented. The analyzer is based on the multithreaded inter-procedural framework by Seidl et al. [76], which we briefly discussed earlier in this chapter. Goblint consists of three parts, a user interface component, an analysis module and a C frontend (Figure 2.6). The frontend uses CIL [63] to parse and simplify C into an intermediate form that can easily be turned into our representation of a control flow graph. Based on specifications of the analyses, we generate a constraint system that we solve using a general purpose constraint solver [29]. The result is then mapped back to the original program and warnings about potential bugs are reported.

The data race analysis itself can also be roughly divided into three separate components. As outlined in the previous section, we perform a base analysis to

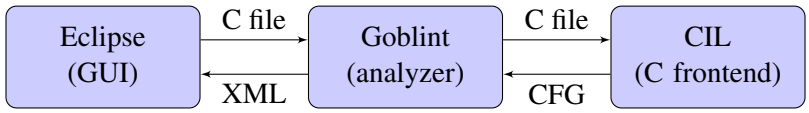

Figure 2.6: The Components of Goblint

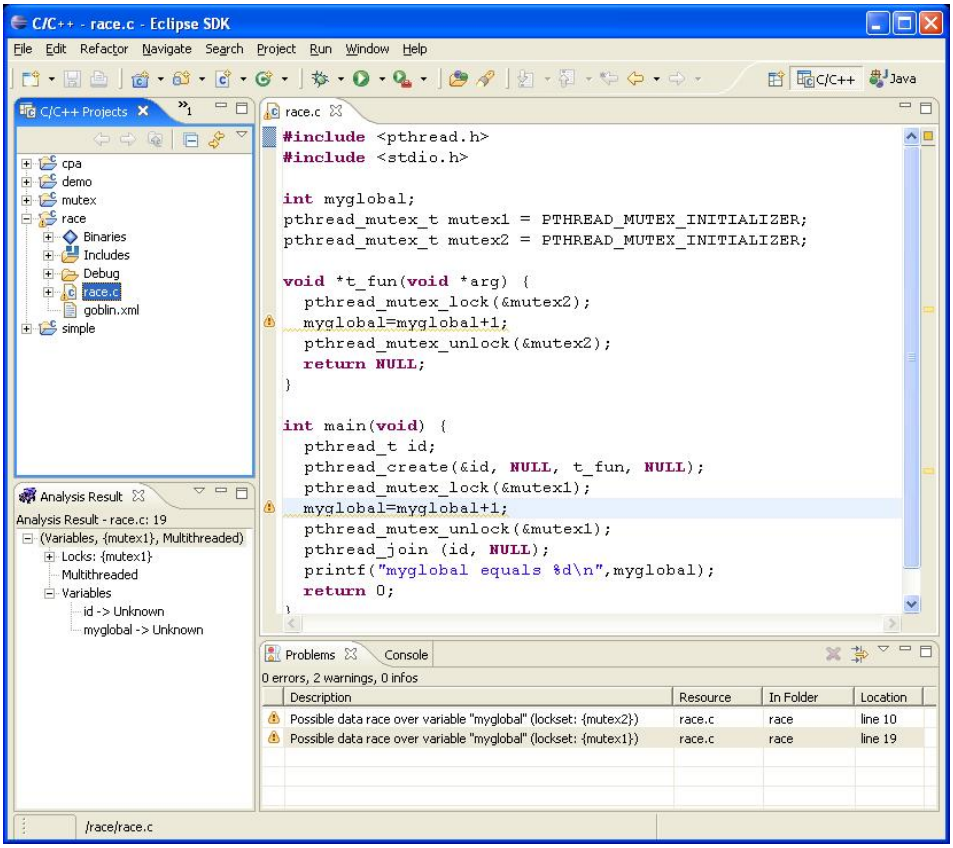

Figure 2.7: A screenshot of Goblint

deal with the complications of C control flow. Simultaneously with the base analysis, and drawing heavily on the information it provides, we perform the lockset analysis, where we register what mutexes are held when a global variable is accessed. When this is done, we post-process the information about the globals and determine any potential races, and then report this information and display it on the Goblint User Interface.

The Goblint analyzer itself is a command line tool with many options for displaying the output. Goblint can present the result of the analysis as an XML file, which can be parsed by the Eclipse plug-in. This leverages on all the benefits that a proper IDE has to offer, such as easy navigation to warnings and much more (see Figure 2.7). When the user views the program and clicks on a line, the state of the analysis at that program point is displayed. This process is not entirely trivial, as one has to map the result from the CFG back to lines in the program code while distinguishing different calling contexts. The current user interface is reasonably

Benchmark	Size (kloc)	Goblint		LOCKSMITH		Races
		Time	Warn.	Time	Warn	
aget	1.2	0.3	5	1.0	4	4
knot	1.3	0.3	7	9.1	8	7
pfscan	1.3	0.1	2	0.6	2	0
ctrace	1.4	0.3	2	3.0	2	0
smtprc	5.7	12	2	8.2	0	0

Table 2.1: Summary of Experimental Results

successful at these tasks.

When Goblint was being developed, we experimented with Goblint on a set of open-source programs of up to 25 thousand lines of code. In order to compare more easily with the state of the art at the time, we used the same benchmarks as were used by the authors of LOCKSMITH. We ran these experiments on an Intel Core 2 Duo @ 1.83GHz PC with 2GB of RAM. The latest benchmarking results on our complete set of test programs are available on the Goblint homepage, and subsequent chapters of this thesis includes more recent benchmarking. Here, our interest was to compare with the LOCKSMITH tool. This evaluation was obtained using version 0.9.3 of Goblint and version 0.4 of LOCKSMITH.

The following is a description of the test programs that we used: *aget* is a multithreaded download accelerator; *pfscan* is a multithreaded parallel file scanner; *knot* is a multithreaded web server distributed with the Capriccio threads package; *ctrace* is a fast, lightweight trace/debug C library containing a sample program with many data races; *smtprc* is a fully configurable, multithreaded open mail relay scanner.

We compared the number of warnings generated by Goblint with the number of warnings generated by LOCKSMITH under two restrictions. We did not consider at this point any dynamically allocated memory and our analysis was field-insensitive. The latter means that we failed to distinguish locks that are correlated to particular fields, but rather we saw the entire struct as a single memory location. To be fair to LOCKSMITH, we turned (its forms of) these features off, since they have a negative impact on its performance.[2]

The results are summarized in Table 2.1, where we indicate for each program the runtime in seconds for its analysis and the number of warnings raised by the different analyzers. We also indicate what we believe is the correct number of real races. This number is based on our manual analysis of the programs and is therefore completely subjective.

The analysis of *aget* and *smtprc* were more accurately handled by LOCK-

[2]We ran LOCKSMITH with the flags no-linearity, no-existentials, and field-insensitive.

SMITH. We had a couple of false alarms due to our more naive treatment of dynamic data structures at that time; the pointer analyses of subsequent chapters were not yet implemented. The test-program *knot* is more interesting from our perspective, because it has many global configuration options, and one of them is to turn caching on or off. Goblint can see this difference, and therefore it does not warn on two caching related data races. There is also a potential race over the setting of a *global* thread attribute, which we correctly warn about but LOCKSMITH fails to detect.

Similarly, for *ctrace* the initialization function has a parameter to determine whether its tracing is asynchronous or not. In the code we analyze, tracing is not performed by a separate server thread, so the two races reported by LOCKSMITH can not occur in the code we are analyzing. If we *change* the code to make tracing asynchronous, then Goblint does raise the corresponding warnings for *such* a program. Unfortunately, we give two other warnings for this program. These are false alarms because the variables are protected by semaphores, which we do not handle, yet.

2.8 Conclusions

We have reviewed the underlying framework of the Goblint analyzer as well as a few other race detection tools. Unlike the other analyzers discussed in this chapter, we perform a sound inter-procedural abstract interpretation of the state of the program. This admits a straightforward solution to path-sensitivity, which is important for sound race detection. Locking operations that may possibly fail, non-blocking and probing primitives, and conditional locking all introduce dependencies between integer variables and the locksets. Our experimental evaluation at the time indicated that in spite of our additional precision with respect to the values of program variables, the run-time of our analyzer is comparable to the LOCKSMITH tool.

More importantly, perhaps, relying on a generic abstract interpretation framework has enabled us to conveniently extend the analyzer with more powerful alias analyses. In the following chapters, we consider the additional analyses which take Goblint beyond the competition when it comes to finding races in the heap.

CHAPTER 3

HERBRAND AND AFFINE EQUALITIES FOR MUST-ALIAS ANALYSIS

As discussed in the introduction, we analyze races in the heap by first using must-equality analysis to infer correlations between access expressions and corresponding lock expressions. This access pattern is then instantiated for some set of shared memory locations based on may points-to information. The success of this approach depends on the precision of the must-equality information used to infer correlations between address expressions.

In this chapter, we consider an abstract domain for analyzing the required *must*-equalities between addresses. The domain is a smooth combination of Herbrand and affine equalities which enables us to describe field accesses and array indexing. While the full combination of uninterpreted functions with affine arithmetics results in intractable assertion checking algorithms, our restricted domain allows us to construct an analysis of address *must*-equalities that runs in polynomial time. We indicate how this analysis can be applied to infer access patterns in programs manipulating arrays and structs.

3.1 Introduction

In order to verify absence of data races in multi-threaded programs, accesses to memory locations need to be correlated with locks that guard them. However, inferring address equalities is more generally applicable. Consistent correlations between memory locations used by a program lies at the heart of many safety properties. In a language with pointer variables, correlating address expressions requires knowing when two expressions *must* alias, i.e., evaluate to the same memory location. In general, techniques for verifying the correct use of interface methods (e.g., [13]) can be refined with must-alias information to check that calls in a syntactically correct sequence consistently refer to the right data elements: a

sequence such as `open(`e_1`); ...; close(`e_2`);` should access the same file handle when referring to the address expressions e_1 and e_2.

More recently, program-specific correlations have been studied: the length of a list is, perhaps, maintained in a separate variable which is thus semantically correlated. Lu et al. [53] apply statistical techniques to detect plausible multi-variable correlations of this kind. Their methods, although successful in detecting real bugs, are flow-insensitive and essentially syntactic; hence not ideal for formal verification. As the precise control flow as well as equalities between variables in the program are ignored, syntactically similar expressions may not represent the same *semantic* correlation, while syntactically different expressions could very well be correlated. In order to enable sound inference of semantic correlations between addresses, we propose a novel analysis of *must-equalities*.

Our analysis is able to interprocedurally relate address expressions which use array indexing and field selection in structs. An access to a nested struct consists in the base address of the data element followed by sequences of selectors, such as $A\,.\,person\,.\,name$. Two such expressions are definitely equivalent if they are *textually identical*. This corresponds to the *Herbrand* interpretation of the binary operator "`.`" and the selector labels. In order to deal with arrays as well, we enhance this base domain by affine expressions for indexed accesses. Two index expressions are equivalent iff they are equivalent w.r.t. the *arithmetic interpretation*. We show that the resulting combination of theories allows to infer *all* valid address equalities in polynomial time.

3.2 The programming model

One key abstraction on which our method relies is that we only track the values of **int** variables and pointers. Thus, we ignore the values stored in arrays or structs. To simplify our setting, we make the additional assumption that the tracked variables themselves are never accessed indirectly through pointers; a common coding practice when developing safety-critical code [42]. Programs to be analyzed are modeled by systems of flow graphs as in Figure 3.1. Recall (from page 23) that a *program* comprises a finite set Proc of *procedure names*. Execution starts with $main \in \mathsf{Proc}$, and each procedure $q \in \mathsf{Proc}$ is given through a *control flow graph* $G_q = (N_q, E_q, e_q, r_q)$, consisting of program points, edges, the entry and the return node, respectively.

Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ denote the set of **int**-variables and $\mathbf{A} = \{\mathbf{a}_1, \ldots, \mathbf{a}_m\}$ the set of pointer variables used by the program. For the moment, we assume all variables to be global, but we will present methods for local variables in Section 3.7. In addition, we assume that we are given a set of names $\mathcal{C}$ denoting the global static data-structures of the program. Each of these data-structures is built

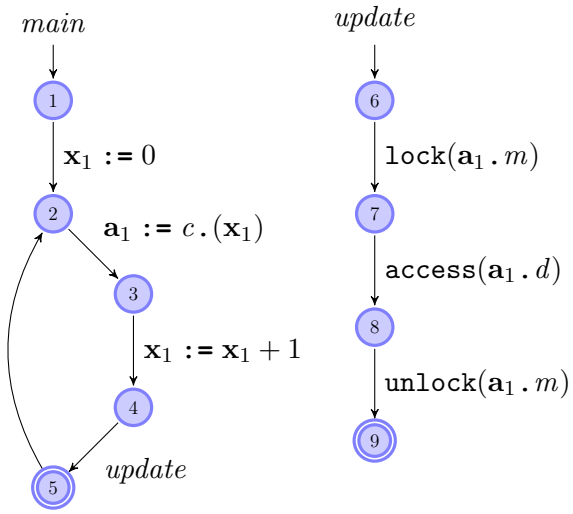

Figure 3.1: Example flow-graph for a *main*-function.

up by forming structs and arrays from a set of base types, such as **int**, **float** or **mutex**. In the presence of dynamic memory allocation, we infer must-equality relationships between pointer variables while also relying on may-alias pointer analysis, as further explained in Section 3.9; until then, we only deal with static data structures.

As we are only interested in assignments to integer and pointer variables, the set of statements Stmt at edges of programs in our model consists of:

- Affine assignments of the form $\mathbf{x}_j$ `:=` $t_0 + \sum_{i=1}^{k} t_i \mathbf{x}_i$ (with $t_i \in \mathbb{Z}$ and $\mathbf{x}_i \in \mathbf{X}$).
- Address assignments of the form $\mathbf{a}_j$ `:=` adr where adr is an address expression possibly involving variables from $\mathbf{X}$ and $\mathbf{A}$ in a way we will specify below.
- Non-deterministic assignments, $\mathbf{x}_j$ `:=` ? and $\mathbf{a}_j$ `:=` ?, which are used to abstract assignments that our analysis cannot handle.

An address expressions adr is constructed from constants $B \in \mathcal{C}$ and address variables $\mathbf{a}_i$ according to the grammar:

$$adr \ ::= \ B \mid \mathbf{a}_i \mid adr\,.b \mid adr\,.(l)$$

where b is a field selector and l is an index expression of the form $l \equiv t_0 + t_1\mathbf{x}_1 + \ldots + t_1\mathbf{x}_k$. We assume that address expressions are *well-typed*. In particular, a

selector b can only be applied to an address expression denoting a pointer to a struct with component b; likewise, only a pointer to an array can be indexed.

Every address pointing into the global data-structures can be uniquely represented by an expression $B\,.\,s_1\,.\,\ldots\,.\,s_r$ where B is the base address of a global data-structure and each s_i is either a field selector or an array index in $\mathbb{Z}$. Since we consider addresses in fixed global data-structures only, the length r is bounded by some global constant d. Let $\mathcal{A}$ denote the set of all these addresses. Since we ignore the values stored in the global data-structures, a program state can be represented by a pair $\langle x, a\rangle$ where $x \in \mathbb{Z}^k$ and $a \in \mathcal{A}^m$ describe the values of the **int** variables and the address variables, respectively. We denote the set of all states by $\mathbb{S} = \mathbb{Z}^k \times \mathcal{A}^m$. Throughout this chapter, we use k and m to denote the number of the (global) integer and address variables, and we use d to denote the maximal depth of data structures!

For an affine combination $t = t_0 + t_1\mathbf{x}_1 + \ldots + t_k\mathbf{x}_k$ and a state $\sigma = \langle x, a\rangle$, we write $[\![t]\!]\sigma$ for the value $t_0 + t_1x_1 + \ldots + t_kx_k \in \mathbb{Z}$. Likewise, for an address expression adr we write $[\![adr]\!]\sigma$ to denote the address obtained from adr by *substituting* the address variables in adr (if there are any) with their values in σ and by *evaluating* all affine index expressions w.r.t. the values of the **int**-variables in σ. Thus, the semantics of assignments for *sets* of states S is defined by:

$$[\![\mathbf{x}_j := t]\!]S = \{\langle(x_1, \ldots, x_{j-1}, [\![t]\!]\langle x, a\rangle, x_{j+1}, \ldots, x_k), a\rangle \mid \langle x, a\rangle \in S\}$$
$$[\![\mathbf{x}_j :=?]\!]S = \{\langle(x_1, \ldots, x_{j-1}, z, x_{j+1}, \ldots, x_k), a\rangle \mid \langle x, a\rangle \in S, z \in \mathbb{Z}\}$$

$$[\![\mathbf{a}_j := adr]\!]S = \{\langle x, (a_1, \ldots, a_{j-1}, [\![adr]\!]\langle x, a\rangle, a_{j+1}, \ldots, a_k)\rangle \mid \langle x, a\rangle \in S\}$$
$$[\![\mathbf{a}_j :=?]\!]S = \{\langle x, (a_1, \ldots, a_{j-1}, a'_j, a_{j+1} \ldots, a_k)\rangle \mid \langle x, a\rangle \in S,$$
$$a'_j \in \mathcal{A} \text{ of appropriate type}\}$$

Every program execution π can be considered as a transformation $[\![\pi]\!] : 2^{\mathbb{S}} \to 2^{\mathbb{S}}$ of the set of states before the execution into the set of states after the execution. Here, we find it convenient to define the semantics as the transformation $\mathbf{R}[u] : 2^{\mathbb{S}} \to 2^{\mathbb{S}}$ that describes which program states can be attained at program point u when program execution starts in a given set of states. Given the transformation $\mathbf{R}[u]$, we can recover the *collecting semantics* of u, i.e., the set of all program states possibly attained during program execution when reaching u, as the set $\mathbf{R}[u](\mathbb{S})$.

In order to define the transformations $\mathbf{R}$, we additionally consider for every procedure q, the transformation of a set of program states before a call to q into the set of program states after the call. Therefore, we introduce for every program point u of q, the auxiliary transformation $\mathbf{S}[u]$ which collects the transformation induced by the executions from u to the end point r_q of q at the same level, i.e., all

recursive calls on its path towards the end of the procedure have returned. Then, the transformation of q is given by $\mathbf{S}[e_q]$ for the start point e_q of q, and we have:

$$
\begin{array}{lllll}
[\mathrm{S1}] & \mathbf{S}[r_q] & \supseteq \mathsf{Id} & & \\
[\mathrm{S2}] & \mathbf{S}[u] & \supseteq \mathbf{S}[v] \circ [\![s]\!] & \text{if } (u,s,v) \text{ is an assignment edge} \\
[\mathrm{S3}] & \mathbf{S}[u] & \supseteq \mathbf{S}[v] \circ \mathbf{S}[e_q] & \text{if } (u,q,v) \text{ is a call edge}
\end{array}
$$

$$
\begin{array}{lllll}
[\mathrm{R0}] & \mathbf{R}[e_{main}] & \supseteq \mathsf{Id} & & \\
[\mathrm{R1}] & \mathbf{R}[e_q] & \supseteq \mathbf{R}[u] & \text{if } (u,q,\_) \text{ is a call edge} \\
[\mathrm{R2}] & \mathbf{R}[v] & \supseteq [\![s]\!] \circ \mathbf{R}[u] & \text{if } (u,s,v) \text{ is an assignment edge} \\
[\mathrm{R3}] & \mathbf{R}[v] & \supseteq \mathbf{S}[e_q] \circ \mathbf{R}[u] & \text{if } (u,q,v) \text{ is a call edge}
\end{array}
$$

Here, the ordering "$\supseteq$" on transformers $f, g : 2^{\mathbb{S}} \to 2^{\mathbb{S}}$ is defined by $f \supseteq g$ iff for every set of states S, $f(S) \supseteq g(S)$.

Example 1. Our example program from Figure 3.1 has only one integer variable and one address variable. Hence a program state is a pair $\langle x, a\rangle \in \mathbb{Z} \times \mathcal{A}$, and we have for program points 1 through 4 and $S \neq \emptyset$:

$$
\begin{array}{ll}
\mathbf{R}[1](S) = S & \mathbf{R}[2](S) = \{\langle 0, a\rangle \mid \langle \_, a\rangle \in S\} \cup \mathbf{R}[5](S) \\
\mathbf{R}[3](S) = \{\langle n, c\,.\,(n)\rangle \mid n \geq 0\} & \mathbf{R}[4](S) = \{\langle n, c\,.\,(n-1)\rangle \mid n \geq 1\}
\end{array}
$$

As the sub-procedure does not change the program state, its same-level transformers are all equal to identity, and thus, $\mathbf{R}[i] = \mathbf{R}[4]$ $(i = 5, 6, \ldots, 9)$. □

3.3 Address equalities

Our goal is to detect equalities between address expressions. In order to do so, we additionally need to track affine equalities between **int** variables. An affine equality is an assertion $t_0 + t_1\mathbf{x}_1 + \ldots + t_k\mathbf{x}_k \doteq 0$ for $t_0, \ldots, t_k \in \mathbb{Q}$. An address equality is an assertion of the form: $adr \doteq adr'$ of address expressions adr, adr'. Here, "$\doteq$" serves as a formal equality symbol. A program state σ *satisfies* the affine equality $t \doteq 0$ iff the left-hand side evaluates to zero: $[\![t]\!]\sigma = 0$. Likewise, the state σ satisfies the address equality $adr \doteq adr'$ iff $[\![adr]\!]\sigma = [\![adr']\!]\sigma$. This means that we consider the Herbrand interpretation for the operator "." as well as for base addresses and field selectors, but use an arithmetic interpretation for index expressions. The latter allows us to identify semantically equal index expressions, such as $\mathbf{x}_1 + 5 + 2\mathbf{x}_1$ and $5 + 3\mathbf{x}_1$.

The state σ satisfies a finite conjunction E of affine and address equalities iff σ satisfies every equality in E. In this case, we write $\sigma \models E$. Likewise for a set

S of states, we write $S \models E$ iff $\sigma \models E$ for all $\sigma \in S$. The conjunction E is *valid* at a program point u, if E is satisfied by all states possible at u, i.e., $\mathbf{R}[u] \models E$.

Example 2. In the program from Figure 3.1, we are interested in the equalities which hold at program point 4. The set of states possible at this point is given by $\mathbf{R}[4] = \{\langle n, c\,.\,(n-1)\rangle \mid n > 0\}$, and thus the equality $\mathbf{a}_1 \doteq c\,.\,(-1 + \mathbf{x}_1)$ is valid at this program point. □

Given this notion of satisfiability, we say that a conjunction of equalities E implies another conjunction of equalities E', iff for all states $\sigma \in \mathbb{S}$, $\sigma \models E$ implies $\sigma \models E'$. Thus, the conjunctions of address and affine equalities can be ordered by implication "⇒". The greatest element $\top$ w.r.t. this ordering is the empty conjunction or true, as it is satisfied by all states. The bottom element $\bot$ in the ordering is false, denoting an unsatisfiable conjunction of equalities.

Consider a finite conjunction E with affine equalities $t_{i0} + t_{i1}\mathbf{x}_1 + \ldots + t_{ik}\mathbf{x}_k \doteq 0$, $i = 1, \ldots, h$. Assume that the conjunction E is satisfiable. Then, we say that it is in *canonical form* iff the following conditions are satisfied:

1. the affine equalities — more precisely, the corresponding coefficient matrix (t_{ij}) — is in row echelon form;[1]
2. the left-hand sides in the address equalities of E are pairwise distinct variables; and
3. no variable that is on the left-hand side of an address equality in E occurs in any of the right-hand sides.

By these restrictions, any conjunction in canonical form comprises at most k affine equalities as well as at most m address equalities.

Example 3. Take the conjunction $(\mathbf{a}_1\,.\,d \doteq c\,.\,(2\mathbf{x}_1)\,.\,d) \wedge (\mathbf{a}_1\,.\,m \doteq c\,.\,(\mathbf{x}_1)\,.\,m)$. An equivalent conjunction in canonical form is $(\mathbf{a}_1 \doteq c\,.\,(2\mathbf{x}_1)) \wedge (\mathbf{x}_1 \doteq 0)$. □

Lemma 1. *For every finite conjunction of equalities E, a finite conjunction in canonical form which is equivalent to E can be constructed in polynomial time.*

Proof. Assume that the conjunction is of the form $E = E_a \wedge E_x$ where E_a is a conjunction of address equalities and E_x is a conjunction of affine equalities. We proceed in three steps. First, we replace every index expression t occurring in the conjunction E_a with the expression $\mathbf{x}_t$ for a fresh variable $\mathbf{x}_t$. Let E'_a denote the resulting conjunction of address equalities.

[1] A matrix is said to be in row echelon form if all zero rows are at the bottom, the leading entry of each non-zero row except the first occurs to the right of the leading entry of the previous row, and the leading entry of any non-zero row is 1.

In the second step, we compute a most general unifier σ for E'_a w.r.t. the Herbrand interpretation. If unification succeeds, then due to the specific form of address expressions, the substitution σ will map each auxiliary variable $\mathbf{x}_t$ either to a field selector or to another auxiliary variable $\mathbf{x}_{t'}$. If there exists an $\mathbf{x}_t$, such that $\sigma(\mathbf{x}_t)$ is a field selector, then the conjunctions are inconsistent and the whole conjunction is equivalent to false.

Otherwise, let E'_x denote the conjunction of all equalities $t_1 - t_2 \doteq 0$ for which the corresponding auxiliaries $\mathbf{x}_{t_i}$ were unified, i.e., $\sigma(\mathbf{x}_{t_1}) = \sigma(\mathbf{x}_{t_2})$. Then E_a is equivalent to the conjunction of E'_x with $E''_a = \bigwedge_i(\mathbf{a}_i \doteq adr_i)$ where the address expressions adr_i are obtained from $\sigma(\mathbf{a}_i)$ by substituting back the affine index expressions t for the auxiliary variables $\mathbf{x}_t$.

Thus, a canonical form of the conjunction E is given by $E''_a \wedge E''_x$, where E''_x is the echelon form for the conjunction $E_x \wedge E'_x$. Using a linear unification algorithm [66] for computing σ, we conclude that the canonical form of E can be computed in time $\mathcal{O}((|E_x| + |E'_x|) \cdot k^2) = \mathcal{O}((s + r \cdot d) \cdot k^2)$ if E consists of s affine equalities and r address equalities. □

Note that we give the complexity estimates under the uniform cost measure, i.e., we assume a constant cost for arithmetic operations.

Lemma 2. *Assume E is a satisfiable conjunction of equalities in canonical form with k* **int***-variables, and addresses of length at most d. Then the following holds:*

1. *For every affine combination t, $E \Rightarrow (t \doteq 0)$ can be decided in time $\mathcal{O}(k^2)$.*
2. *For every address expression adr, $E \Rightarrow (\mathbf{a}_i \doteq adr)$ can be decided in time $\mathcal{O}(d \cdot k^2)$.*

Proof. As the first statement is immediate from linear algebra, we only prove the second. Let us assume that $adr \equiv A\,.\,s_1\,.\,\ldots\,.\,s_h$, i.e., adr does not contain an address variable. Then the implication holds iff E contains an equality $\mathbf{a}_i \doteq A\,.\,s'_1\,.\,\ldots\,.\,s'_h$, and for each $\lambda = 1, \ldots, h$, the access expressions s_λ and s'_λ are equal under E: either both s_λ and s'_λ are field selectors and identical, or both s_λ and s'_λ are index expressions and $E \Rightarrow (s_\lambda - s'_\lambda \doteq 0)$.

Now assume that $adr \equiv \mathbf{a}_j\,.\,s_1\,.\,\ldots\,.\,s_h$ for some address variable $\mathbf{a}_j$. Unless $adr \equiv \mathbf{a}_i$, the implication can only hold if E also contains an equality for $\mathbf{a}_i$. Moreover, this equality is of the form $\mathbf{a}_i \doteq a\,.\,s'_1\,.\,\ldots\,.\,s'_{h+l}$ for some $l \geq 0$ where a is either an address constant A or an address variable $\mathbf{a}_r$. Then the implication holds iff E also contains an equality $\mathbf{a}_j \doteq a\,.\,s''_1\,.\,\ldots\,.\,s''_l$ where for $\lambda = 1, \ldots, l$, the accesses s'_λ and s''_λ are equal under E, and for $\lambda = l+1, \ldots, h$, the accesses s'_λ and $s_{\lambda-l}$ are equal under E. Assuming that the address equality in E for particular address variables can be retrieved in constant time, at most d affine equalities must be checked for subsumption by E — giving us the stated complexity bound. □

Thus, both logical implication and equivalence between satisfiable conjunctions E, E' in canonical form can be decided in time $\mathcal{O}((m^2 \cdot d + k) \cdot k^2)$.

Let $\mathbb{E}$ denote the set of equivalence classes of finite conjunctions ordered by implication. The greatest lower bound of (the equivalence classes of) two conjunctions $E, E' \in \mathbb{E}$ is (the equivalence class containing) the conjunction of all the equalities in E and E'. The partial order $\mathbb{E}$ thus is a complete lattice — given that all descending chains are finite.

Corollary 1. *Every chain $E_0 \Rightarrow \ldots \Rightarrow E_p$ of pairwise inequivalent conjunctions E_j using k* **int** *variables and m address variables has length $p \leq m + k + 1$.*

This follows because any two inequivalent conjunctions E_i and E_j have counterparts in canonical form, E_i' and E_j', respectively. The implication $E_i' \Rightarrow E_j'$ can only hold, if E_i' contains strictly more equalities than E_j'. Therefore, all chains in the lattice will eventually stabilize after at most $m + k + 1$ steps.

In summary, we have proven that the set of equivalence classes of conjunctions of address equalities ordered with implication $(\mathbb{E}, \Rightarrow)$ is a complete lattice.

3.4 Weakest pre-conditions

We compute all valid equalities through weakest pre-condition computation. For a conjunction of equalities E, the weakest pre-condition for an assignment and a non-deterministic assignment is given by substitution and universal quantification, respectively:

$$[\![\mathbf{x}_i := t]\!]^{\mathsf{T}}(E) = E[t/\mathbf{x}_i] \qquad [\![\mathbf{a}_i := a]\!]^{\mathsf{T}}(E) = E[a/\mathbf{a}_i]$$
$$[\![\mathbf{x}_i := ?]\!]^{\mathsf{T}}(E) = \forall\, \mathbf{x}_i.\, E \qquad [\![\mathbf{a}_i := ?]\!]^{\mathsf{T}}(E) = \forall\, \mathbf{a}_i.\, E$$

While our domain is closed under substitution, it does not directly support universal quantification. We are rescued by the fact that in the sub-domain of linear arithmetic, determining the weakest pre-condition for a non-deterministic assignment to an **int** variable $\mathbf{x}_i$, it suffices to consider the conjunction of the weakest pre-conditions of the assignments $\mathbf{x}_i := 0$ and $\mathbf{x}_i := 1$ [57]. On the other hand, $\forall\, \mathbf{a}_i.\, E$ for a conjunction E in canonical form involving the address variable $\mathbf{a}_i$ is necessarily false, if $\mathbf{a}_i$ can range over at least two addresses [60]. For simplicity of presentation, let us assume there are no singleton types. Thus, the weakest pre-conditions for non-deterministic assignments can be simplified:

$$[\![\mathbf{x}_i := ?]\!]^{\mathsf{T}}(E) = E[0/\mathbf{x}_i] \wedge E[1/\mathbf{x}_i]$$
$$[\![\mathbf{a}_i := ?]\!]^{\mathsf{T}}(E) = \begin{cases} \mathsf{false} & \text{if } \mathbf{a}_i \text{ occurs in } E \\ E & \text{otherwise} \end{cases}$$

Note that these results do not hold for the general combination of linear arithmetic with uninterpreted functions. Consider the following equality, which is ruled out in our restricted combination of theories:

$$\mathbf{a}_1 \,.\, (0) + \mathbf{a}_1 \,.\, (1) \doteq \mathbf{a}_1 \,.\, (\mathbf{x}_1) + \mathbf{a}_1 \,.\, (1 - \mathbf{x}_1)$$

Here, the operator "+" is applied to address expressions. As an arithmetic operator, "+" is commutative. Hence, the above equality is true when $\mathbf{x}_1$ is substituted by 0 and 1, but not for other integers.

We now set up a constraint system to characterize the weakest pre-condition *transformers* $\mathbf{R}^{\mathsf{T}}[v]$, which transform conjunctions of equalities at the program point v into the weakest pre-condition for their validity at *program start*. The constraint system uses auxiliary transformers $\mathbf{S}^{\mathsf{T}}[v]$, which transform the post-condition of a procedure q into the weakest pre-condition at the program point v of the same procedure q.

$$\begin{array}{lll}
[\mathrm{S1}^{\mathsf{T}}] & \mathbf{S}^{\mathsf{T}}[r_q] & \Rightarrow \mathsf{Id} \\
[\mathrm{S2}^{\mathsf{T}}] & \mathbf{S}^{\mathsf{T}}[u] & \Rightarrow [\![s]\!]^{\mathsf{T}} \circ \mathbf{S}^{\mathsf{T}}[v] \quad (u, s, v) \text{ an assignment edge} \\
[\mathrm{S3}^{\mathsf{T}}] & \mathbf{S}^{\mathsf{T}}[u] & \Rightarrow \mathbf{S}^{\mathsf{T}}[e_q] \circ \mathbf{S}^{\mathsf{T}}[v] \quad (u, q, v) \text{ a call edge}
\end{array}$$

$$\begin{array}{lll}
[\mathrm{R0}^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[e_{main}] & \Rightarrow \mathsf{Id} \\
[\mathrm{R1}^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[e_q] & \Rightarrow \mathbf{R}^{\mathsf{T}}[u] \quad (u, q, _) \text{ a call edge} \\
[\mathrm{R2}^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[v] & \Rightarrow \mathbf{R}^{\mathsf{T}}[u] \circ [\![s]\!]^{\mathsf{T}} \quad (u, s, v) \text{ an assignment edge} \\
[\mathrm{R3}^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[v] & \Rightarrow \mathbf{R}^{\mathsf{T}}[u] \circ \mathbf{S}^{\mathsf{T}}[e_q] \quad (u, q, v) \text{ a call edge}
\end{array}$$

Here, the ordering "$\Rightarrow$" on transformers $f, g \colon \mathbb{E} \to \mathbb{E}$ is defined by $f \Rightarrow g$ iff for all conjunctions of equalities E, $f(E) \Rightarrow g(E)$. The greatest solution to the system will be the weakest pre-condition transformers. We state this as a theorem.

Theorem 1. *For every program point u, set of states $S \subseteq \mathbb{S}$, and conjunction of equalities $E \in \mathbb{E}$,*

$$\mathbf{S}[u](S) \models E \iff S \models \mathbf{S}^{\mathsf{T}}[u](E) \quad \textit{and} \quad \mathbf{R}[u](\mathbb{S}) \models E \iff \mathbf{R}^{\mathsf{T}}[u](E) = \mathsf{true}$$

Proof. The identity and weakest pre-condition transformers for individual edges are defined in a standard way. Relating the least fixed point of the system $\mathbf{S}$ with the greatest fixed point of the system $\mathbf{S}^{\mathsf{T}}$, we are only required to show that the following conditions are satisfied:

$$\begin{aligned}
f(S) \cup g(S) \models E &\iff S \models f^{\mathsf{T}}(E) \wedge g^{\mathsf{T}}(E) \\
(f \circ g)(S) \models E &\iff S \models (g^{\mathsf{T}} \circ f^{\mathsf{T}})(E).
\end{aligned}$$

These follow from the properties of weakest pre-condition transformers. The second equivalence follows from an analogous fixed-point induction and the fact that $\mathbb{S} \models E$ only if $\mathsf{true} \Rightarrow E$. □

Example 4. In our example program, the weakest predicate transformers for program points 2, 3 and 4 are given by the constraints:

$$\mathbf{R}^{\mathsf{T}}[2] \Rightarrow [0/\mathbf{x}_1] \qquad \mathbf{R}^{\mathsf{T}}[2] \Rightarrow \mathbf{R}^{\mathsf{T}}[4]$$
$$\mathbf{R}^{\mathsf{T}}[3] \Rightarrow \mathbf{R}^{\mathsf{T}}[2] \circ [c\,.\,(\mathbf{x}_1)/\mathbf{a}_1] \qquad \mathbf{R}^{\mathsf{T}}[4] \Rightarrow \mathbf{R}^{\mathsf{T}}[3] \circ [\mathbf{x}_1 + 1/\mathbf{x}_1]$$

Using methods described below, we find that $\mathbf{R}^{\mathsf{T}}[2]$ maps the post-condition $\mathbf{a}_1 \doteq c\,.\,(-1 + \mathbf{x}_1)$ to the pre-condition $\mathbf{a}_1 \doteq c\,.\,(-1)$. □

Solving such constraint systems requires effective computation of function comparisons, greatest lower bounds and compositions. Thus, we need an finite and effective representation of these predicate transformers.

3.5 Finite representation

Inspired by order-theory, let us call single address equalities $\mathbf{a}_i \doteq adr$ and affine equalities $t \doteq 0$ *atomic*. Let $\mathbb{E}_A$ denote the set of atomic equalities. According to Lemma 1, every conjunction has a canonical form, which is a conjunction of atomic equalities. Hence, every transformer $f\colon \mathbb{E} \to \mathbb{E}$, which is completely distributive, i.e., preserves true and distributes over conjunctions, is uniquely determined by its restriction $f|_{\mathbb{E}_A}$ to atomic equalities.

This observation, though, does not yet provide a finite representation of weakest pre-condition transformers because the number of single equalities is still infinite. The second idea, therefore, is not to track weakest pre-conditions for each equality separately, but to consider *generic equalities*. Every generic equality serves as a template which covers a range of equalities of similar form simultaneously.

In order to infer weakest pre-conditions for all affine equalities, we consider the generic post-condition $p \equiv \mathbf{p}_0 + \mathbf{p}_1\mathbf{x}_1 + \cdots + \mathbf{p}_k\mathbf{x}_k \doteq 0$, where $\mathbf{p}_0, \ldots, \mathbf{p}_k$ are fresh variables not occurring in the program. The weakest pre-conditions for p can be represented as conjunctions of equalities

$$\sum_{i=0}^{k} c_{i0}\mathbf{p}_i + \sum_{i=0}^{k}\sum_{j=1}^{k} c_{ij}\mathbf{p}_i\mathbf{x}_j \;\doteq\; 0 \tag{3.1}$$

for constants $c_{ij} \in \mathbb{Q}$.

Example 5. Since our running example has just one **int** variable, the generic affine post-condition is $e_{\mathit{aff}} \equiv \mathbf{p}_0 + \mathbf{p}_1\mathbf{x}_1 \doteq 0$. The parametric pre-condition for e_{aff} w.r.t. the assignment $\mathbf{x}_1 \texttt{ := } \mathbf{x}_1 + 1$ is then $\mathbf{p}_0 + \mathbf{p}_1 + \mathbf{p}_1\mathbf{x}_1 \doteq 0$. □

A generic address post-condition is of the form $\mathbf{a}_i \doteq a\,.\,s_1\,.\,\ldots\,.\,s_r$ (for some $r \leq d$) where a is either an address constant in $\mathcal{C}$ or another address variable in $\mathbf{A}$, and each s_l is either a field name or an indexing pattern $\mathbf{p}_{l0} + \mathbf{p}_{l1}\mathbf{x}_1 + \cdots + \mathbf{p}_{lk}\mathbf{x}_k$. Weakest pre-conditions for such a generic address post-condition will be conjunctions of parametric affine equalities and parametric address equalities. The generic coefficients to be considered in the parametric affine equalities now are elements from the set $\mathbf{P}_r = \{\mathbf{p}_{li} \mid l \in [1,r], i \in [0,k]\}$. Thus, the affine equalities are of the form:

$$c_{000} + \sum_{j=1}^{k} c_{00j}\mathbf{x}_j + \sum_{l=1}^{r}\sum_{i=0}^{k} c_{li0}\mathbf{p}_{li} + \sum_{l=1}^{r}\sum_{i=0}^{k}\sum_{j=1}^{k} c_{lij}\mathbf{p}_{li}\mathbf{x}_j \;\doteq\; 0 \qquad (3.2)$$

for constants $c_{lij} \in \mathbb{Q}$. Also, the parametric address equalities will be address equalities where index expressions are of the same form as left-hand sides in (3.2).

Example 6. For the address variable $\mathbf{a}_1$, a generic post-condition is of the form $e_{\mathit{adr}} \equiv \mathbf{a}_1 \doteq c\,.\,(\mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1)$. The parametric pre-condition for e_{adr} w.r.t. the assignment $\mathbf{a}_1 \texttt{ := } c\,.\,(\mathbf{x}_1)$ is given by $c\,.\,(\mathbf{x}_1) \doteq c\,.\,(\mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1)$, whose canonical form is $-\mathbf{x}_1 + \mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1 \doteq 0$. □

The conjunction of parametric equalities forms a lattice $\mathbb{E}_d$, which has the same structure as the lattice $\mathbb{E}$ – except that the set of **int** variables is now extended with the set of parameters $\mathbf{p}_{l0}$ and products $\mathbf{p}_{li}\mathbf{x}_j$ of parameters and **int** variables. The height of the complete lattice $\mathbb{E}_d$ therefore is bounded by $\mathcal{O}(d \cdot k^2 + m)$.

In our application, generic post-conditions suffice to arrive at a finite specification of weakest pre-condition transformers. Let T denote the set of all well-typed generic address equalities between pointer variables in the program. Then the set T is finite and of cardinality $\mathcal{O}(m^2 \cdot t \cdot d)$, where t is the maximal size, i.e., number of fields, of a global data structure's type. This set T is *complete* in the sense that for any concrete atomic equality $e \in \mathbb{E}_A$, there exists a substitution $\sigma\colon \mathbf{P}_d \to \mathbb{Q}$ and a generic post-condition $e' \in T$ such that $e = e'\sigma$. Any function $f\colon T \to \mathbb{E}_d$ can be extended to a completely distributive function $\mathsf{ext}\,(f)\colon \mathbb{E} \to \mathbb{E}$ defined by $(\mathsf{ext}\,(f))(e) = (f(e'))\sigma$ for all atomic equalities $e = e'\sigma$ where e' is some generic equality, and σ a substitution.

Example 7. The only type-compatible generic post-condition in our example is e_{adr}. However, if we were to add an address variable $\mathbf{a}_2$ pointing to mutexes, the set of generic post-conditions $T = \{e_{\mathit{aff}},\, e_{\mathit{adr}}\}$ would additionally include the set $\{\mathbf{a}_2 \doteq c\,.\,(\mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1)\,.\,m,\ \mathbf{a}_2 \doteq \mathbf{a}_1\,.\,m\}$. □

We now show that the weakest predicate transformers that occur in our constraint system can indeed be obtained as extensions of functions from $T \to \mathbb{E}_d$. In order to do so, we set up a new constraint system $\mathbf{R}^\sharp$ over functions from $T \to \mathbb{E}_d$. This is obtained from the constraint system $\mathbf{R}^\mathsf{T}$ by replacing all operations by their parametric counterparts. Thus, implication "$\Rightarrow^\sharp$" and greatest lower bounds $\wedge^\sharp$ are now defined according to the domain $\mathbb{E}_d$. Also, the transfer functions for assignments are lifted to parametric equalities. It remains to define composition $\circ^\sharp$ for functions $f^\sharp, g^\sharp \colon T \to \mathbb{E}_d$.

First, we observe that every parametric equality can be obtained from one of the generic post-conditions by a transformation σ of the parameters. Therefore assume that e' is a generic post-condition and $g^\sharp(e') = e_1 \wedge \cdots \wedge e_r$ where $e_l = e'_l\sigma_l$ for generic post-conditions e'_l and linear transformations σ_l. Then we define

$$(f^\sharp \circ^\sharp g^\sharp)(e') = (f^\sharp(e'_1))\sigma_1 \wedge \cdots \wedge (f^\sharp(e'_r))\sigma_r \,.$$

If e' is the generic affine equality, this amounts to computing the canonical form of a conjunction of $\mathcal{O}(k^4)$ parametric equalities. If e' is a generic address equality, the canonical form must be computed for a conjunction of $\mathcal{O}(m^2)$ parametric address equalities and $\mathcal{O}(d^2 \cdot k^4)$ parametric affine equalities whose normalization may at worst consume time $\mathcal{O}(m^2 \cdot d^4 \cdot k^8)$.

Example 8. Let $f = [\![\mathbf{x}_1 \texttt{ := } ?]\!]^\mathsf{T}$ and $g = [\![\mathbf{a}_1 \texttt{ := } c\,.\,(\mathbf{x}_1)]\!]^\mathsf{T}$. We then compute the composition $(f \circ g)(e_{adr})$ as follows:

$$\begin{array}{rcl}
(f \circ g)(e_{adr}) & = & f(-\mathbf{x}_1 + \mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1 \doteq 0) \\
& = & (f(e_{\mathit{aff}}))\,\sigma \quad \text{for } \sigma = [\mathbf{p}_{10}/\mathbf{p}_0, (-1+\mathbf{p}_{11})/\mathbf{p}_1] \\
& = & ((\mathbf{p}_0 \doteq 0) \wedge (\mathbf{p}_0 + \mathbf{p}_1 \doteq 0))\sigma \\
& = & ((\mathbf{p}_0 \doteq 0) \wedge (\mathbf{p}_1 \doteq 0))\,\sigma = (\mathbf{p}_{10} \doteq 0) \wedge (-1 + \mathbf{p}_{11} \doteq 0)
\end{array}$$

This computation occurs during the analysis of our running example, because the while-loop has the same effect as the non-deterministic assignment of f. □

Theorem 2. *For any program point u, $\mathbf{R}^\mathsf{T}[u] = \mathsf{ext}\,(\mathbf{R}^\sharp[u])$.*

Proof. We proceed by fixpoint induction. A crucial step is to show that not only "$\wedge$", but also composition commutes with ext, i.e., that

$$\mathsf{ext}\,(f) \circ \mathsf{ext}\,(g) = \mathsf{ext}\,(f \circ^\sharp g)$$

To see that, we calculate:

$$\begin{array}{rcllcl}
(\mathsf{ext}\,(f) \circ \mathsf{ext}\,(g))(e'\sigma) & = & \mathsf{ext}\,(f)(\mathsf{ext}\,(g)(e'\sigma)) & = & \mathsf{ext}\,(f)(g(e'))\sigma) \\
& = & \mathsf{ext}\,(f)((\bigwedge e'_i\sigma_i)\sigma) & = & \mathsf{ext}\,(f)(\bigwedge e'_i(\sigma_i\sigma)) \\
& = & \bigwedge(f(e'_i))(\sigma_i\sigma) & = & (\bigwedge(f(e'_i))\sigma_i)\sigma \\
& = & ((f \circ^\sharp g)(e'))\sigma & = & (\mathsf{ext}\,(f \circ^\sharp g))(e'\sigma)
\end{array}$$

where $g(e') = \bigwedge e'_i\sigma_i$ as above. □

Example 9. We can now compute the solution to the constraint system by fixpoint iteration starting from true. The computation stabilizes after three iterations, giving the following pre-conditions for the address post-condition:

$$\mathbf{R}^{\mathsf{T}}[2](e_{adr}) = (\mathbf{a}_1 \doteq c\,.\,(\mathbf{p}_{10})) \wedge (\mathbf{p}_{10} + \mathbf{p}_{11} \doteq 0) \wedge (-1 + \mathbf{p}_{11} \doteq 0)$$
$$\mathbf{R}^{\mathsf{T}}[3](e_{adr}) = (\mathbf{p}_{10} \doteq 0) \wedge (-1 + \mathbf{p}_{11} \doteq 0)$$
$$\mathbf{R}^{\mathsf{T}}[4](e_{adr}) = (\mathbf{p}_{10} + \mathbf{p}_{11} \doteq 0) \wedge (-1 + \mathbf{p}_{11} \doteq 0)$$

For the affine post-condition e_{aff}, the pre-condition $(\mathbf{p}_0 \doteq 0) \wedge (\mathbf{p}_1 \doteq 0)$ is obtained, meaning no non-trivial affine equalities hold at these points. □

3.6 Computing all valid equalities

Given the weakest pre-condition transformer $\mathbf{R}^{\mathsf{T}}[v]$ for program point v, computing all equalities which are valid at v then boils down to solving a suitable inhomogeneous system of equations. We have:

Theorem 3. *The equalities that hold at each program point can be computed in polynomial time.*

Proof. Let e' denote a generic post-condition and $e = e'\sigma$ an atomic equality for some substitution σ of the parameters occurring in e'. By Theorem 1, e holds at program point u iff $\mathbf{R}^{\mathsf{T}}[u](e) = \mathsf{true}$, which, by Theorem 2, means that $(\mathbf{R}^{\sharp}[u](e'))\sigma = \mathsf{true}$. The latter means that $\mathbf{R}^{\sharp}[u](e')$ does not contain non-trivial address equalities, but is a conjunction of at most $\mathcal{O}(d \cdot k^2)$ affine equalities $t \doteq 0$ where $t\sigma \doteq 0$ is valid for all values $x \in \mathbb{Z}^k$.

Assume that $\mathbf{p}'_1, \ldots, \mathbf{p}'_r$ are the parameters occurring in t, the affine combination t is of the form: $t \equiv c_{00} + \sum_{i=1}^{k} (c_{0i} + \sum_{l=1}^{r} c_{li}\mathbf{p}'_l)\,\mathbf{x}_i$ for suitable $c_{li} \in \mathbb{Q}$. Then $t\sigma \doteq 0$ is valid for all values $x \in \mathbb{Z}^k$ iff $c_{00} = 0$ and σ is a solution of each of the equations $c_{0i} + \sum_{l=1}^{r} c_{li}\mathbf{p}'_l \doteq 0$ $(i = 1, \ldots, k)$. We conclude that finding all substitutions σ such that $e'\sigma$ is valid at program point u can be reduced to solving a system of $\mathcal{O}(k \cdot d \cdot k^2) = \mathcal{O}(d \cdot k^3)$ inhomogeneous equations over $\mathbb{Q}$ where the number of unknowns is bounded by $d \cdot (k + 1)$. The latter task can be done with a polynomial number of arithmetic operations. By repeating this procedure for every possible generic post-condition, we obtain a finite representation of all equalities which are valid at program point u. □

Example 10. As we saw in Example 9, at all points in the loop the parametric pre-condition for e_{aff} has $\mathbf{p}_0 = \mathbf{p}_1 = 0$ as its solution. The parametric pre-condition for the generic post-condition e_{adr}, on the other hand, is given by:

$$\mathbf{R}^{\mathsf{T}}[4](\mathbf{a}_1 \doteq c\,.\,(\mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1)) \;=\; (\mathbf{p}_{10} + \mathbf{p}_{11} \doteq 0) \wedge (-1 + \mathbf{p}_{11} \doteq 0)$$

As no **int**-variables $\mathbf{x}_i$ are involved here, this pre-condition is true iff $\mathbf{p}_{11} = 1$ and $\mathbf{p}_{10} = -1$. Therefore, the only non-trivial equality which holds at program point 4 is $\mathbf{a}_1 \doteq c\,.\,(-1 + \mathbf{x}_1)$. □

To summarize, the set of all equalities, which hold at a given program point, can be compactly represented by a polynomially sized set of triples $\langle e, \sigma, V\rangle$ — each consisting of a generic post-condition e together with one particular solution for the conjunction of parametric affine pre-conditions of e and a basis V of the vector space of solutions of the corresponding homogeneous system. Assuming that the basis V is in (column) echelon form, we can determine if a given equality holds at a certain program point in time $\mathcal{O}(d^2 \cdot k^2)$.

3.7 Local variables

All program variables have so far been considered *global*. Along the lines of [59], we now extend the analysis to possibly recursive programs with local variables as well. From the k integer variables, we consider the first $k' \leq k$ variables $\mathbf{x}_1, \ldots, \mathbf{x}_{k'}$ as local and the remaining ones as global. Similarly for pointers, the first $m' \leq m$ variables $\mathbf{a}_1, \ldots, \mathbf{a}_{m'}$ denote local variables while the remaining ones denote global pointer variables.

For passing of parameters, we adopt w.l.o.g. the convention that *all* locals of the caller are passed by value into the locals of the callee. This enables us to reason about equalities involving local variables of the caller.

We extend the concrete semantics with an extra operator H which transforms the effect of a procedure body into the effect of a procedure call:

$$\mathsf{H}(f)(S) = \{\langle (x_1, \ldots, x_{k'}, x'_{k'+1}, \ldots, x'_k), (a_1, \ldots, a_{m'}, a'_{m'+1}, \ldots, a'_m)\rangle \mid \langle x, a\rangle \in S, \langle x', a'\rangle \in f(\{\langle x, a\rangle\})\}$$

The constraint system for computing weakest pre-conditions of procedure calls is modified accordingly by introducing the operator H^T:

$$\begin{array}{lll} [\mathrm{S1}^\mathsf{T}] & \mathbf{S}^\mathsf{T}[r_q] \Rightarrow \mathsf{Id} & \\ [\mathrm{S2}^\mathsf{T}] & \mathbf{S}^\mathsf{T}[u] \Rightarrow [\![s]\!]^\mathsf{T} \circ \mathbf{S}^\mathsf{T}[v] & (u, s, v) \text{ an assignment edge} \\ [\mathrm{S3}^\mathsf{T}] & \mathbf{S}^\mathsf{T}[u] \Rightarrow \mathsf{H}^\mathsf{T}(\mathbf{S}^\mathsf{T}[e_q]) \circ \mathbf{S}^\mathsf{T}[v] & (u, q, v) \text{ a call edge} \end{array}$$

Here, the operator H^T must be defined such that the first statement of Theorem 1 holds for the new constraint system. Given the concrete transformer f of a procedure and the corresponding weakest pre-condition transformer f^T, the following condition must hold for all sets of states S and conjunctions of equalities E:

$$\mathsf{H}(f)(S) \models E \iff S \models \mathsf{H}^\mathsf{T}(f^\mathsf{T})(E)$$

Consider an arbitrary post-condition E for a procedure call to f. This post-condition may not only speak about globals, but also about locals of the caller as well as any local variable further down in the call-stack. All these locals, however, are inaccessible during the execution of the procedure f and thus can temporarily be considered as *constants*. In order to deal with these temporary constants, we introduce place holders $\bullet_\tau$ for every possible type of local pointer variables a_j or constant addresses.

Accordingly, we consider the following set of parametric post-conditions E':

$$(1) \quad \mathbf{a}_i \doteq \mathbf{a}_j \,.\, s \qquad (2) \quad \mathbf{a}_i \doteq \bullet_\tau \,.\, s$$
$$(3) \quad \bullet_\tau \doteq \mathbf{a}_i \,.\, s \qquad (4) \quad \bullet_{\tau_1} \doteq \bullet_{\tau_2} \,.\, s$$

for global pointer variables a_i, a_j and type-compatible parametric sequences of selectors s, where each parametric index is of the form $\mathbf{p}_{l0} + \mathbf{p}_{l(k'+1)}\mathbf{x}_{k'+1} + \cdots + \mathbf{p}_{lk}\mathbf{x}_k$. Furthermore, we consider the parametric affine post-condition:

$$(5) \quad \mathbf{p}_0 + \mathbf{p}_{k'+1}\mathbf{x}_{k'+1} + \cdots + \mathbf{p}_k\mathbf{x}_k \doteq 0$$

for global variables $\mathbf{x}_{k'+1}, \ldots, \mathbf{x}_k$. Assume now that we are given the weakest pre-conditions $f^{\mathsf{T}}(E')$ of the called procedure for all these post-conditions E' speaking about global variables (and perhaps $\bullet_\tau$).

We now define the weakest pre-condition $\mathsf{H}^{\mathsf{T}}(f^{\mathsf{T}})(E)$. In each case, we decompose $E = E'\sigma$ for a generic post-condition E' of one of the types (1) through (5) and a suitable substitution σ. Then, we define

$$\mathsf{H}^{\mathsf{T}}(f^{\mathsf{T}})(E) = (f^{\mathsf{T}}(E'))\sigma\,.$$

It only remains to explain the decomposition of E. We first consider a post-condition E of the form $\mathbf{a}_i \doteq \mathbf{a}_j \,.\, s$ for global variables $\mathbf{a}_i, \mathbf{a}_j$. Then E' is of the parametric post-condition of format (1). For every index expression $s_l = t_0 + t_1\mathbf{x}_1 + \cdots + t_k\mathbf{x}_k$ in s, σ maps p_{l0} to the affine combination consisting of t_0 together with all occurring multiples of locals, i.e., to $t_0 + t_1\mathbf{x}_1 + \cdots + t_{k'}\mathbf{x}_{k'}$.

If E is of the form $\mathbf{a}_i \doteq X \,.\, s$ where $\mathbf{a}_i$ is a global variable and X either is a local of the caller, a constant address or a place holder $\bullet_\tau$ all of the type τ, we choose E' of the parametric format (2) and σ is constructed as before, but moreover maps the place holder $\bullet_\tau$ to X. (When $X = \bullet_\tau$ no substitution is required for X.)

The case where E is of the form $X \doteq \mathbf{a}_i \,.\, s$ is treated analogously. In case where E is of the form $X_1 \doteq X_2 \,.\, s$ and each X_i is a local of the caller, constant address or place holder, then we choose the appropriate generic post-condition E' now of type (4). The substitution σ treats index expressions as before, but now maps $\bullet_{\tau_1}$ to X_1 and $\bullet_{\tau_2}$ to X_2.

Finally, if E is an affine equality $t_0 + t_1\mathbf{x}_1 + \cdots + t_k\mathbf{x}_k \doteq 0$, then we choose E' to be of format (5) where the substitution σ maps p_0 to $t_0 + t_1\mathbf{x}_1 + \cdots + t_{k'}\mathbf{x}_{k'}$, and p_i to t_i for $i > k'$.

Example 11. Consider the post-condition $\mathbf{a}_1 \doteq \mathbf{a}_2\,.\,(\mathbf{p}_{10}+\mathbf{p}_{11}\mathbf{x}_1+\mathbf{p}_{12}\mathbf{x}_2)$, where $\mathbf{a}_1$, $\mathbf{a}_2$, and $\mathbf{x}_1$ are local, but $\mathbf{x}_2$ is global and may be changed during the procedure call. Assume that the callee only performs the statement $\mathbf{x}_2$ `:=` ?. Since the post-condition is of the type (4), we compute the pre-condition as follows:

$$\begin{aligned}[\![\mathbf{x}_2 \doteq ?]\!]^{\mathsf{T}}(\bullet_{\tau_1} \doteq \bullet_{\tau_2}\,.\,(\mathbf{p}_{10} + \mathbf{p}_{12}\mathbf{x}_2)) = \\ (\bullet_{\tau_1} \doteq \bullet_{\tau_2}\,.\,(\mathbf{p}_{10})) \wedge (\bullet_{\tau_1} \doteq \bullet_{\tau_2}\,.\,(\mathbf{p}_{10} + \mathbf{p}_{21})) = \\ (\bullet_{\tau_1} \doteq \bullet_{\tau_2}\,.\,(\mathbf{p}_{10})) \wedge (\mathbf{p}_{21} \doteq 0)\end{aligned}$$

To obtain the weakest pre-condition of $\mathbf{a}_1 \doteq \mathbf{a}_2\,.\,(\mathbf{p}_{10}+\mathbf{p}_{11}\mathbf{x}_1+\mathbf{p}_{12}\mathbf{x}_2)$, we apply the substitution σ, which maps $\mathbf{p}_{10}$ to $\mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1$ and replaces the place-holders with the local address variables: $\mathbf{a}_1 \doteq \mathbf{a}_2\,.\,(\mathbf{p}_{10} + \mathbf{p}_{11}\mathbf{x}_1) \wedge \mathbf{p}_{21} \doteq 0$. □

The second part of our analysis applies the weakest pre-condition transformers of procedures, as defined through the first part of the constraint system, to construct a constraint system for the weakest pre-condition transformers for post-conditions at program points v:

$$\begin{array}{lll}
[\mathrm{R}0^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[e_{main}] \Rightarrow \mathsf{Id} & \\
[\mathrm{R}1^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[e_q] \quad \Rightarrow \mathbf{R}^{\mathsf{T}}[u] & (u, q, _) \text{ a call edge} \\
[\mathrm{R}2^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[v] \quad \Rightarrow \mathbf{R}^{\mathsf{T}}[u] \circ [\![s]\!]^{\mathsf{T}} & (u, s, v) \text{ an assignment edge} \\
[\mathrm{R}3^{\mathsf{T}}] & \mathbf{R}^{\mathsf{T}}[v] \quad \Rightarrow \mathbf{R}^{\mathsf{T}}[u] \circ \mathsf{H}^{\mathsf{T}}(\mathbf{S}^{\mathsf{T}}[e_q]) & (u, q, v) \text{ a call edge}
\end{array}$$

This time, however, the post-conditions for the weakest pre-condition transformer $\mathbf{R}^{\mathsf{T}}[v]$ for a program point of a procedure f need not use $\bullet$-variables to refer to variables deeper down in the call-stack. Instead, they may refer to the *locals* of f. Accordingly, occurring transformers are described by their weakest pre-conditions for the generic affine post-condition together with the generic address post-conditions $\mathbf{a}_i \doteq \mathbf{a}_j\,.\,s$ for local or global address variables a_i, a_j and suitable selector sequences s.

3.8 Intra-procedural forward analysis

Currently, the above analysis cannot be directly implemented in the Goblint framework. We do plan to extend the framework to support different approaches to inter-procedural analysis such that one could combine the procedure summary approach for the must-equality analysis with the other analyses used in the tool.

In the meantime, we can rely on a forward formulation of the analysis which can be directly implemented in the analyzer. This analysis is intra-procedural and relies on the analysis framework to apply the generic functional approach to inter-procedural analysis, as described in Section 2.4.

For computing with abstract values from the lattice $\mathbb{E}$, we require an effective algorithm for computing the least upper bound $E = E_1 \sqcup E_2$ of two satisfiable conjunctions E_1, E_2. For this, we need a different representation. The conjunction E is *complete* if the affine equalities in E are in row echelon form, and E contains exactly one equality for every pair of address variables $\mathbf{a}_i$ and $\mathbf{a}_j$ for which some equality is implied by E, as well as one equality for every address equality $\mathbf{a}_j$ and every global data-structure A for which some equality is implied.

Lemma 3. *The least upper bound of two complete conjunctions E_1 and E_2 can be computed in polynomial time.*

Proof. Assume that for $i = 1, 2$, E_i is of the form: $E_i = E_{i,a} \wedge E_{i,x}$ where $E_{i,x}$ and $E_{i,a}$ collect the affine and address equalities in E_i, respectively. Then the least upper bound E is of the form $E = E_a \wedge E_x$ where E_x is a conjunction of affine equalities which span the intersection of the affine spaces spanned by the conjunctions $E_{1,x}$ and $E_{2,x}$. By standard methods from linear algebra, this can be computed in time $\mathcal{O}(k^3)$.

It remains to determine the conjunction E_a of address equalities implied both by E_1 and E_2. For that, we consider each address equality $\mathbf{a}_i \doteq X \,.\, s_1$ occurring in E_1. In order to put an equality for $\mathbf{a}_i$ and X into E_a, we must find a similar equality $\mathbf{a}_i \doteq X \,.\, s_2$ in E_2 where both selector strings have the same structure, i.e., contain index expressions $t_1^{(i)}, \ldots, t_{d'}^{(i)}$ at the same positions in s_i as well as the same field selectors at the remaining positions. Furthermore for $\lambda = 1, \ldots, d'$, affine equalities $r_\lambda^{(i)} \doteq 0$ must exist with $E_{i,x} \Rightarrow (r_\lambda^{(i)} \doteq 0)$ such that

$$t_\lambda^{(1)} + r_\lambda^{(1)} = t_\lambda^{(2)} + r_\lambda^{(2)}$$

If this is the case, we add the equality $\mathbf{a}_i \doteq \mathbf{a}_j \,.\, s$ to E_a where the selector s is obtained from s_1 by replacing the λ-th index expression $t_\lambda^{(1)}$ with $t_\lambda^{(1)} + r_\lambda^{(1)}$. Note that the necessary linear algebra calculations can be performed in time $\mathcal{O}(d \cdot k^3)$. □

Example 12. Consider the conjunctions

$$E_1 = (\mathbf{x}_1 - \mathbf{x}_3 \doteq 0) \wedge (\mathbf{a}_1 \doteq \mathbf{a}_2 \,.\, (\mathbf{x}_1))$$
$$E_2 = (\mathbf{x}_2 - 5 \doteq 0) \wedge (\mathbf{x}_3 - 5 \doteq 0) \wedge (\mathbf{a}_1 \doteq \mathbf{a}_2 \,.\, (\mathbf{x}_2))$$

Since $E_{x,1} \Rightarrow (-\mathbf{x}_1 + \mathbf{x}_3 \doteq 0)$ and $E_{x,2} \Rightarrow (-\mathbf{x}_2 + \mathbf{x}_3 \doteq 0)$, we see that $\mathbf{x}_1 + (-\mathbf{x}_1 + \mathbf{x}_3) = \mathbf{x}_2 + (-\mathbf{x}_2 + \mathbf{x}_3)$, and therefore conclude $E \Rightarrow (\mathbf{a}_1 \doteq \mathbf{a}_2 \,.\, (\mathbf{x}_3))$.

As before, we provide the abstract effects of the four types of assignments (see page 47) onto abstract values from $\mathbb{E}$. Each non-$\bot$ abstract value can be considered as a conjunction $E_a \wedge E_x$ where E_x is a conjunction of affine equalities, and E_a is a *complete* conjunction of address equalities. On the affine equalities, our analysis proceeds precisely like Karr's analysis of affine equalities [48], while the treatment of conjunctions of address equalities is new. We begin with assignments to integer variables. For a non-deterministic assignment, we define:

$$[\![\mathbf{x}_i \texttt{ := } ?]\!]^\sharp E \quad = \quad \exists^\sharp \mathbf{x}_i.E$$

where $\exists^\sharp \mathbf{x}_i.E$ is the conjunction of all equalities implied by E that do not contain $\mathbf{x}_i$. To compute this, assume $E = E_a \wedge E_x$ as described above. Then $\exists^\sharp \mathbf{x}_i.E = E'_a \wedge E'_x$ with E'_a and E'_x defined as follows.

If $\mathbf{x}_i$ does not occur in E_x, then $E'_x = E_x$ and E'_a is obtained from E_a by removing all address equalities which contain $\mathbf{x}_i$. Otherwise, let $E_x = (t \doteq 0) \wedge E_1$ where $t \equiv t_0 + t_1\mathbf{x}_1 + \ldots + t_k\mathbf{x}_k$ and $t_i \neq 0$. We construct the arithmetic expression $t' = -t_i^{-1}(t - t_i\mathbf{x}_i)$ which no longer contains $\mathbf{x}_i$, and substitute it in E_a and E_1 for all occurrences of $\mathbf{x}_i$, i.e., we define: $E'_a = E_a[t'/\mathbf{x}_i]$ and $E'_x = E_1[t'/\mathbf{x}_i]$.

Now consider an assignment $s \equiv \mathbf{x}_i \texttt{ := } t$ to the variable $\mathbf{x}_i$ where $t \equiv t_0 + t_1\mathbf{x}_1 + \cdots + t_k\mathbf{x}_k$. If $\mathbf{x}_i$ does not occur in t, then:

$$[\![\mathbf{x}_i \texttt{ := } t]\!]^\sharp E \quad = \quad \exists^\sharp \mathbf{x}_i.E \wedge (\mathbf{x}_i \doteq t)$$

If $\mathbf{x}_i$ does occur in t, i.e., $t_i \neq 0$, then we construct $t' = -t_i^{-1}(t - t_i\mathbf{x}_i)$, which we use to substitute all occurrences of $\mathbf{x}_i$ in the pre-condition E before we add the new equality $\mathbf{x}_i \doteq t$. Thus, in this case, we define:

$$[\![\mathbf{x}_i \texttt{ := } t]\!]^\sharp E \quad = \quad E[t'/\mathbf{x}_i] \wedge (\mathbf{x}_i \doteq t)$$

We now consider assignments to an address variables. The abstract effect of a non-deterministic assignment is given by

$$[\![\mathbf{a}_j \texttt{ := } ?]\!]^\sharp E \quad = \quad \exists^\sharp \mathbf{a}_j.E$$

where $\exists^\sharp \mathbf{a}_j.E$ is the conjunction of all equalities implied by E that do not contain the address variable $\mathbf{a}_i$. In order to get an explicit construction, let $E = E_a \wedge E_x$ as before, then $\exists^\sharp \mathbf{a}_j.E = E'_a \wedge E_x$ where E'_a is obtained from E_a by simply removing all equalities containing $\mathbf{a}_i$.

Finally, consider an assignment $s \equiv \mathbf{a}_j \texttt{ := } adr$ to the address variable $\mathbf{a}_j$. If $adr = \mathbf{a}_j$, then we have the following definition.

$$[\![\mathbf{a}_j \texttt{ := } adr]\!]^\sharp E \quad = \quad \begin{cases} E & \text{if } adr = \mathbf{a}_j \\ \exists^\sharp \mathbf{a}_j.E \wedge (\mathbf{a}_j \doteq adr) & \text{otherwise} \end{cases}$$

For all these assignments, we find:

Lemma 4. *Assume s is a non-deterministic or deterministic assignment to an integer or address variable. Then we have:*

1. *For every set S of states, $S \models E$ iff $[\![s]\!](S) \models [\![s]\!]^\sharp E$.*
2. *The abstract effect $[\![s]\!]^\sharp$ is distributive, i.e., commutes with $\bot$ and preserves least upper bounds.*
3. *The abstract effect $[\![s]\!]^\sharp$ can be computed in polynomial time.*

Given the abstract effects of assignments, we now obtain a constraint system for the conjunctions of equalities that are valid at every program point by taking the constraint system for the collecting semantics $\mathbf{R}$ and replacing the complete lattice of sets of states with the abstract lattice $\mathbb{E}$ and the concrete transformations $[\![s]\!]$ induced by assignments s with the abstract transformations $[\![s]\!]^\sharp$. Thus, we consider the following constraint system $\mathbf{R}^\sharp$:

$$
\begin{array}{lllll}
[\mathrm{R0}]^\sharp & \mathbf{R}^\sharp[e_{main}] & \sqsupseteq \mathsf{true} & & \\
[\mathrm{R1}]^\sharp & \mathbf{R}^\sharp[e_q] & \sqsupseteq \mathbf{R}[u]^\sharp & \text{if } (u, q, \_) \text{ is a call edge} \\
[\mathrm{R2}]^\sharp & \mathbf{R}^\sharp[v] & \sqsupseteq [\![s]\!]^\sharp(\mathbf{R}^\sharp[u]) & \text{if } (u, s, v) \text{ is an assignment edge} \\
[\mathrm{R3}]^\sharp & \mathbf{R}^\sharp[v] & \sqsupseteq \mathbf{R}^\sharp[u] & \text{if } (u, q, v) \text{ is a call edge}
\end{array}
$$

Since the right-hand sides of this constraint system are monotonic in their arguments, the system has a least solution. This solution is also precise because the right-hand sides are composed from distributive functions:

Theorem 4. *Assume for all program points v, $\mathbf{R}^\sharp[v]$ is the least solution of the constraint system $\mathbf{R}^\sharp$. Then for every program point v and every equality e,*

$$\mathbf{R}[v] \models e \qquad \textit{iff} \qquad \mathbf{R}^\sharp[v] \Rightarrow e$$

Moreover, the least solution of $\mathbf{R}^\sharp$ can be computed in polynomial time.

The first statement of the theorem follows from ordinary fixpoint induction whereas the second holds since the value at every program point may increase only $m+k+1$ times, where m is the number of address variables and k is the number of integer variables, and every right-hand side of the constraint system can be evaluated in polynomial time.

3.9 Application: race detection

In order to avoid race, we ensure the following condition for every pair of accesses in the program: if the two access expressions *may* alias, then the acquired

lock expressions *must* alias [62]. We ensure this condition by inferring access correlations using the must-equality analysis and associating these correlations with may-alias equivalence classes, as we will illustrate through the following example.

Example 13. Assume the address variables $\mathbf{a}_{acc}$ and $\mathbf{a}_{lock}$ represent an access expression and a lock expression that need to be correlated, and our must-alias analysis provides the following information:

$$(\mathbf{a}_{acc} \doteq \mathbf{a}_1 \,.\, d \,.\, (\mathbf{x}_1)) \wedge (\mathbf{a}_{lock} \doteq \mathbf{a}_1 \,.\, m \,.\, (\mathbf{x}_1))$$

These equalities imply that the access to the data array of the structure pointed to by $\mathbf{a}_1$ is protected by a corresponding element in the mutex array. □

The access pattern we can infer in the above example depends on the information we have about $\mathbf{a}_1$. If the analysis can infer that $\mathbf{a}_1$ is definitely equal to some statically allocated structure c, a pattern for access to the elements of c is obtained. Otherwise, may-alias analysis [41] is called upon to divide the set of all pointer variables into equivalence classes. The simplest such approach, which suffices for some applications [53], equates all pointers of the same type. Then our method allows to infer access patterns for data structures of a given type. A more refined analysis distinguishes heap objects depending also on their allocation sites, in which case our analysis derives more refined patterns.

Note that must-equality information complements may-aliasing by ensuring that $\mathbf{a}_{acc}$ and $\mathbf{a}_{lock}$ are referring to the *same object* within the equivalence class of $\mathbf{a}_1$. This is crucial in order to verify per-element locking schemes, where each element in, e.g., a linked list has its own lock. Pratikakis et al. [67] describe a technique based on existentially typed label-flow to address this issue with the aid of programmer annotations; must-equality information allows one to infer per-element correlations automatically.

We now sketch a general method to *infer* linear correlations between accesses to shared variables and the mutexes associated with them. Assuring mutual exclusion requires that there exist for all accesses to a shared variable at the very least a consistent relationship between the field names of the accessed data and those of the locked mutex. Checking this is straightforward because the set of field names is finite. If the fields names are consistent, then in the presence of array indexing we must also detect whether the relationship between the indices are consistent across all accesses. Here, we show how linear relationship between integer indexes can be detected automatically.

Assume a data structure is accessed at p different locations. For each such access, we register the $\mathbf{a}_{lock}$ and $\mathbf{a}_{acc}$ variables, as well as (the matrix representation of) the affine equalities T. In order to relate the access and lock variables, we associate indexing matrices with the access paths. Let an address be given in the

following form

$$\mathbf{a}_{acc} \doteq c\,.\,s_1\,.\,\ldots\,.\,s_d, \quad \text{where } s_{i+1} = a_{0i} + a_{1i}\mathbf{x}_1 + \cdots a_{ki}\mathbf{x}_k.$$

This defines the indexing matrix $A = (a_{ij})$, where for each j, the j-th column of the matrix corresponds to the indexing expression s_j. If this expression is a field access, then the j-th column is zero. We denote with L the same construction for the indexing occurring in the lock variable $\mathbf{a}_{lock}$.

For each of the p accesses, we construct the two matrices A and L as above and we record the equalities T which held at the time of that access. There exists a linear access relationship between data and locks, if one can find a solution C together with auxiliary matrices C_i to the following system of equations:

$$A_k C + T_k C_k = L_k \qquad (\text{for } k = 1, \ldots, p).$$

The matrix C gives the correlation, while the matrices C_i, take into account the equalities between integer variables that hold during different accesses.

The system can be conveniently solved by Gauss-Jordan elimination on the following matrix equation.

$$\begin{bmatrix} A_1 & T_1 & 0 & \cdots & 0 \\ A_2 & 0 & T_2 & \cdots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ A_p & 0 & 0 & \cdots & T_p \end{bmatrix} \begin{bmatrix} C \\ C_1 \\ C_2 \\ \ldots \\ C_p \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \\ \ldots \\ L_p \end{bmatrix}$$

Example 14. Assume we have a program with an array of locks protecting a matrix of data. This is accessed by two threads and we find that during the accesses the following equalities held:

$$\begin{aligned} \mathbf{a}_{acc} &\doteq data\,.\,(\mathbf{x}_1)\,.\,(\mathbf{x}_2) & \mathbf{a}_{acc} &\doteq data\,.\,(\mathbf{x}_3)\,.\,(\mathbf{x}_2) \\ \mathbf{a}_{lock} &\doteq lock\,.\,(5\mathbf{x}_1 + \mathbf{x}_3) & \mathbf{a}_{lock} &\doteq lock\,.\,(\mathbf{x}_1 + \mathbf{x}_2) \\ 0 &\doteq \mathbf{x}_2 - \mathbf{x}_3 & 0 &\doteq \mathbf{x}_1 - 5\mathbf{x}_3 \end{aligned}$$

The system to solve would then be as follows:

$$\left[\begin{array}{cc|c|c} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & -5 \end{array}\right] \begin{bmatrix} C \\ C_1 \\ C_2 \end{bmatrix} = \left[\begin{array}{cc} 0 & 0 \\ 5 & 0 \\ 0 & 0 \\ 1 & 0 \\ \hline 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 0 \end{array}\right]$$

This system has a solution with $C = \left(\begin{smallmatrix} 5 & 0 \\ 1 & 0 \end{smallmatrix}\right)$, where the auxiliary matrices $C_1 = (-1\ 0)$ and $C_2 = (1\ 0)$. Thus, the lock and data are consistently correlated. □

3.10 Related work

Detecting affine equalities in programs was pioneered by Karr [48]. This algorithm was extended to the inter-procedural case by Müller-Olm and Seidl [57]. A long line of research has provided methods for intra-procedurally detecting Herbrand equalities precisely [36, 50, 56, 84] — while the inter-procedural case still remains unsolved. A precise analysis algorithm is known for *functions* without side effects [60] and for arbitrary procedures if only unary operator symbols are considered [39].

When it comes to combining affine and Herbrand equalities, the basic approach is inspired by methods of combining decision procedures [64]. However, Gulwani and Tiwari [37] have shown that assertion checking over the full combined domain is coNP-hard. Hence, they subsequently present a highly expressive domain that allows sound analysis of pointer arithmetic and recursive data-structures in the style of Deutsch [24], but their algorithm is no longer complete w.r.t. their chosen abstraction [38]. Our domain construction, based on a sufficiently restricted sub-class of Herbrand terms carefully enhanced with fragments of linear arithmetic, enables sound and complete analysis in polynomial time.

3.11 Conclusion

We have presented a must-alias analysis which infers all equalities between address expressions and can be proven to be sound and complete w.r.t. the chosen abstraction. In this abstraction, conditional branching is replaced with non-deterministic branching and pointers stored in the shared data-structures are not tracked. We indicated how these equalities can be used to infer correlations between locks and accesses.

For simplicity, we have assumed that index expressions are evaluated over the integral domain $\mathbb{Z}$. Instead, we could have chosen $\mathbb{Z}_{2^w}$, i.e., integers modulo a suitable power of 2, by replacing the linear algebra methods for vector spaces of affine equalities with the corresponding methods for modules over the principal ideal ring $\mathbb{Z}_{2^w}$ [58]. However, if the programs to be analyzed only employ simple forms of index expressions, it might be sufficient to replace tracking of affine equalities with tracking of variable equalities alone [59].

This analysis infers relevant must-equality information for dynamically allocated data which combined with may-alias information allows one to infer access patterns for fine-grained locking schemes, i.e., when the lock and the data it protects are within the same struct or array. We address this restriction in the following chapter.

CHAPTER 4

REGIONS AND STATIC OWNERS FOR MAY-ALIAS ANALYSIS

While must-equalities are important for fine-grained locking schemes, we now need to consider dynamically data structures, such as linked lists, where a single lock protects all the elements it contains. In this chapter, we present a region-based pointer analysis which seeks to identify disjoint *regions* of dynamically allocated objects to ensure that write accesses to the same region are always protected by the same mutexes. Our approach has been implemented in the Goblint analyzer and we have successfully applied it on code from the Linux kernel, such as the access vector cache. This code relies on a synchronized hash table where an array of doubly linked lists is protected by an array of locks.

4.1 Introduction

Writing multi-threaded code which both is correct and manipulates complicated data-structures can be cumbersome. Programmers of low-level software therefore mostly adhere to simple and conservative programming styles. Accordingly, dynamic shared data-structures are avoided whenever possible, and when dynamic allocation of memory is inevitable, one common idiom is to rely on *non-overlapping* data-structures and protect each of these *memory regions* by a dedicated lock. This occurs naturally when resources are maintained in hash-table-like data-structures, i.e., arrays of linked lists where each list is protected by its own lock as illustrated in Figure 4.1.

There are different levels of granularity at which locking schemes for shared data-structures operate: at one extreme, an individual mutex is maintained for each data element separately, known as *per-element locking* [67]; at the other extreme, coarse-grained locking schemes use a single mutex to protect all data nodes allocated at a given point in the program. In between, there are subtler

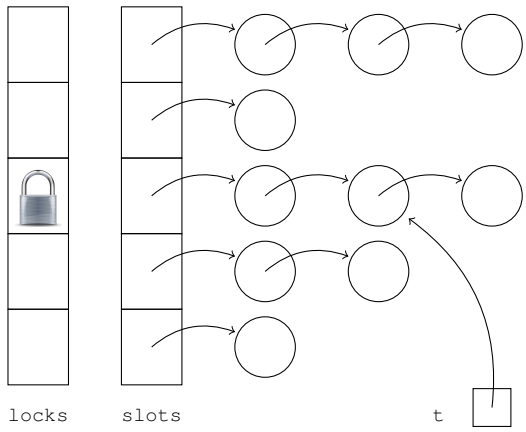

Figure 4.1: Memory regions

cases of medium-grained locking where certain dynamically allocated elements protect a bunch of other elements (not quite per-element), or elements allocated at a given point are not all protected by the same mutex (not quite coarse-grained). Here, we are concerned with the latter case. In many applications, we found that the dynamic data-structures protected by one mutex are disjoint from the data-structures protected by other mutexes. The *number* of protected disjoint data-structures, however, can be large. This is the case, e.g., for synchronized hash-tables where each bucket is protected by an individual mutex.

Consider the two-bucket hash-table in Figure 4.2 where elements allocated by the insert function end up in two distinct lists. The correctness of the locking scheme in this program hinges on the fact that the expressions `t1→data` and `t2→data` can never evaluate to the same address, i.e., they can never *alias*. We can be sure of this because the two lists are disjoint and thus closed under pointer reachability.

We call an analysis a *region analysis* if it infers a safe partitioning of the heap into disjoint regions. For region analysis, one could use sophisticated analyses to infer shapes of data-structures. Another approach would be to summarize dynamically allocated objects as *blobs* of memory associated with finitely many abstract locations such as allocation sites. While the first approach has difficulties scaling to larger programs, the second approach fails when elements allocated at the same program point end up in distinct data-structures protected by distinct mutexes, as in the above example.

We present a region analysis which is reasonably fast, yet sufficiently precise to deal with programs that manipulate disjoint heap regions. It identifies the set of static globals within the region accessed by local pointers. It also deals with arrays

```
typedef struct node { int data; struct node *next;} node;
node *even_list, *odd_list;

void insert(int data) {
  node *t = new(data);
  if (even(data)) { t→next = even_list; even_list = t; }
  else { t→next = odd_list; odd_list = t; } }

void even_worker() {
  node *t1 = even_list;
  while (t1 != NULL) {
    lock(even_mutex);
    access(t1→data);
    t1 = t1→next;
    unlock(even_mutex); } }

void odd_worker() {
  node *t2 = odd_list;
  while (t2 != NULL) {
    lock(odd_mutex);
    access(t2→data);
    t2 = t2→next;
    unlock(odd_mutex); } }
```

Figure 4.2: Elements placed into linked lists

of regions by allowing regions to be indexed with symbolic index expressions. For the example above, the analysis would maintain that the two lists are disjoint, `t1` is pointing into the region of `even_list`, and `t2` is pointing into the region of `odd_list`.

Our region analysis can be extended to a race detection method by adding two components. First, a must alias analysis which provides information on which global address are definitely pointed to by a pointer variable, e.g., provided by [59]. Second, a symbolic lock set analysis which determines for every program point a *representation*, which may involve symbolic address expressions, of the set of definitely held locks when reaching this program point.

4.2 Region inference

For the theoretical exposition, we spell out our approach for a minimalistic programming model which is just rich enough to exhibit the key ideas of our analysis of multi-threaded programs using dynamic data-structures and arrays. At first, we restrict ourselves to programs which consist of only a single procedure represented by a finite control-flow graph where each edge is labeled with a basic operation; in Section 4.3, we will extend the approach to an interprocedural setting. We only track the values of local variables pointing into the global memory. The global memory is shared between different processes and consists of blocks, which either may be statically allocated at program start or dynamically allocated

during program execution through some operator $\mathsf{new}\,\tau$ (for some type τ). For the moment, we rule out pointers into the stack as well as pointer arithmetic and assume that pointers always point to the beginning of blocks. In Section 4.5, we will add global arrays, and in Section 4.6, we indicate how the basic approach can be extended to work also in presence of (well-behaved) pointer-arithmetic as required for the analysis of, e.g., the Linux kernel API for doubly linked lists.

We assume that the frontend provides us with a normalized representation of assignments. For the beginning, we consider the following forms of expressions and assignments:

adr	$::=$	$\mathbf{y}$	local pointer variable
	$\mid$	$\&a$	static global address
$pexp$	$::=$	$\mathbf{y} \to b$	dereferencing of pointers
val	$::=$	$adr \mid \mathsf{null}$	pointer value
	$\mid$	$\mathsf{new}(\tau)$	memory allocation
$pass$	$::=$	$pexp = val;$	memory write
	$\mid$	$\mathbf{y} = val; \mid \mathbf{y} = pexp;$	variable assignment

Let L and G denote the set of local pointer variables and the set of addresses of static global memory cells, respectively. Region analysis aims at inferring *potential reachability* between elements from $G \cup L$. Our analyzer therefore maintains for every program point an equivalence relation π on globals. Two elements $x_1, x_2 \in G$ are put into the same equivalence class when some memory cell is jointly reachable from both x_1 and x_2 through iterated field selection and dereferencing. Additionally, we maintain for every program point a function $\rho : L \to 2^{G \cup \{\bullet\}}$ mapping each local $\mathbf{y}$ to a set of globals identifying the region into which $\mathbf{y}$ may possibly point. The bullet $\bullet$ identifies the region of all thread-local dynamically allocated memory cells. When a thread allocates an object and initializes its fields, the object is seen as residing within this thread-local region until it is reachable from, or can itself reach, one of the global regions.

Equivalence relations have also been used for *may-alias* analysis [41]. There, two expressions are considered equivalent if they may denote the same address. May-alias equivalence classes do not collapse when one is reachable from the other. On the other hand, while non-reachability *implies* non-equality, we cannot extract definite non-reachability information from non-equality. Thus, ensuring that pointers which traverse complicated structures may not alias is extremely difficult without the explicit notion of disjointness: one must precisely express the aliasing relationship, or all information about non-reachability is lost.

Here, an equivalence relation π is represented by the set of two-element subsets $\{x, y\}$ with $(x, y) \in \pi$ — implying that the *trivial* equivalence relation is represented by the empty set. Let $\mathbf{P}$ and $\mathbf{R}$ denote the set of all equivalence relations on G and the set of all maps from L to $2^{G \cup \{\bullet\}}$, respectively. Both sets form

complete lattices for the ordering induced by the subset orderings on the set of two-element subsets of G and $G \cup \{\bullet\}$, respectively. In particular, for equivalence relations $\pi_1, \pi_2 \in \mathbf{P}$, the greatest lower bound $\pi_1 \sqcap \pi_2$ is given by the intersection of the sets of unordered pairs corresponding to π_1 and π_2, respectively; whereas the least upper bound $\pi_1 \sqcup \pi_2$ is the least equivalence relation containing all pairs from π_1 and π_2.

Using a suitable data-structure for partitions, the operations "$\sqcap$" and "$\sqcup$" on elements of $\mathbf{P}$ can be executed in polynomial time. Consider a pair $T = \langle \pi, \rho \rangle$ describing the current program state. We assume that all sets $\rho(\mathbf{y})$ are *closed* under π. We call a set X *closed* under the equivalence relation π, if $x \in X$ and $\{x, x'\} \in \pi$ implies that also $x' \in X$. For an arbitrary pair $\langle \pi, \rho \rangle$, let $\mathsf{cl}_\pi X$ denote the least set X' with $X \subseteq X'$ which is closed under π, and $\mathsf{cl}_\pi \rho$ the closure of the function ρ by lifting the closure operation pointwise: $(\mathsf{cl}_\pi \rho)(x) = \mathsf{cl}_\pi(\rho(x))$.

We now specify how a pair $T = \langle \pi, \rho \rangle$ describing the program state before an assignment s is transformed into a pair $\langle \pi', \rho' \rangle$ describing the program state after the assignment, i.e., we define the *abstract* meaning $[\![s]\!]^\sharp$ of the statement s. First, consider statements where local pointers are set:

$$\begin{aligned}
[\![\mathbf{y} = \&a]\!]^\sharp\, T &= \langle \pi, \rho[\mathbf{y} : \mathsf{cl}_\pi\{\&a\}] \rangle \\
[\![\mathbf{y} = \mathbf{y}']\!]^\sharp\, T &= [\![\mathbf{y} = \mathbf{y}' \to b]\!]^\sharp\, T = \langle \pi, \rho[\mathbf{y} : \rho(\mathbf{y}')] \rangle \\
[\![\mathbf{y} = \mathsf{null}]\!]^\sharp\, T &= \langle \pi, \rho[\mathbf{y} : \emptyset] \rangle \\
[\![\mathbf{y} = \mathsf{new}(\tau)]\!]^\sharp\, T &= \langle \pi, \rho[\mathbf{y} : \{\bullet\}] \rangle
\end{aligned}$$

where $\rho[\mathbf{y}_i : X_i]_{i \in I}$ is the function obtained from ρ by updating the image of $\mathbf{y}_i$ to X_i for all $i \in I$. Now, consider a write to memory through local pointers. In case either null or a pointer to a fresh memory block is written, the abstract state does not change:

$$[\![\mathbf{y} \to b = \mathsf{null}]\!]^\sharp\, T = [\![\mathbf{y} \to b = \mathsf{new}(\tau)]\!]^\sharp\, T = T$$

Finally, consider a write to memory of the form $\mathbf{y} \to b = \mathbf{y}'$. Let $X = \rho(\mathbf{y}) \cup \rho(\mathbf{y}')$ denote the owners involved in the assignment. If $\bullet \notin X$, we join the equivalence classes of $\mathbf{y}$ and $\mathbf{y}'$:

$$\begin{aligned}
[\![\mathbf{y} \to b = \mathbf{y}']\!]^\sharp\, T = \langle \pi', \mathsf{cl}_{\pi'} \rho \rangle \quad &\text{where} \\
\pi' = \pi \sqcup \{\{x, x'\} \mid x \neq x',\ x, x' \in \rho(\mathbf{y}) \cup \rho(\mathbf{y}')\}&
\end{aligned}$$

If the bullet is involved but no globals, $X \subseteq \{\bullet\}$, then simply $[\![\mathbf{y} \to b = \mathbf{y}']\!]^\sharp\, T = T$; however, when $\{\bullet\} \subsetneq X$, we additionally must consider all pointers that may point into the thread-local region denoted by the bullet. Letting $Y = \{\mathbf{y}, \mathbf{y}'\} \cup$

$\{\mathbf{y}'' \in L \mid \bullet \in \rho(\mathbf{y}'')\}$, we join all globals from X into one equivalence class into which all variables from Y may now additionally point:

$$\begin{aligned}[\![\mathbf{y} \to b = \mathbf{y}']\!]^\sharp\, T = \langle \pi', \mathsf{cl}_{\pi'} \rho' \rangle \quad \text{where} \\ \pi' = \pi \sqcup \{\{x, x'\} \mid x \neq x',\ x, x' \in X\} \\ \rho' = \rho[\mathbf{y}'' : (\rho(\mathbf{y}'') \setminus \{\bullet\}) \cup X]_{\mathbf{y}'' \in Y}\end{aligned}$$

Since we have associated global owners with the freshly allocated objects, we remove the bullet from ρ.

For proving the soundness of the analysis, we rely on a small-step operational semantics of heap-manipulating programs. Since we have currently ruled out procedures, the concrete program state when reaching a program point u consists of a pair $\sigma = \langle \mu, \lambda \rangle$ where λ maps the local pointers to the start addresses of blocks and μ describes the current global memory. We represent the memory μ by a map which assigns a value to every address-field pair (l, b). Type-safety requires that l is the address of a block in the global memory of struct type τ which has a field b. For convenience, we assume that every field of pointer type which has not yet been initialized, holds the value null.

In μ, the address l_1 is *reachable* from the address l_2 iff l_2 can be obtained from l_1 by repeated field selection and dereferencing. A *region* in μ is a set R of addresses in μ such that every $l_1 \in R$ satisfies the condition: $l_2 \in R$ whenever $\mu(l_1, b) = l_2$ for some field name b of the struct at address l_1. This definition implies that the set of regions of μ form a *partition* of the addresses in μ. In particular, no address in the region R is reachable from any address outside the region R.

Assume that the concrete program state $\sigma = \langle \mu, \lambda \rangle$ induces a partition $\Pi = \{R_1, \ldots, R_m\}$ of the addresses in μ. Then σ is in the *concretization* of the abstract state $T = \langle \pi, \rho \rangle$, i.e., $\sigma \in \gamma(T)$, iff the following holds:

1. If $\{x, x'\} \notin \pi$ for global static addresses $x \neq x'$, then x and x' are not in the same region of μ.

2. If $x \notin \rho(\mathbf{y})$, then x and $\lambda(\mathbf{y})$ are not in the same region of μ.

3. If $\rho(\mathbf{y}) \cap \rho(\mathbf{y}') = \emptyset$, for local variables $\mathbf{y} \neq \mathbf{y}'$, then $\lambda(\mathbf{y})$ and $\lambda(\mathbf{y}')$ are not in the same region of μ.[1]

4. If $\rho(\mathbf{y}) = \emptyset$, then $\lambda(\mathbf{y})$ equals null.

It follows that if $\rho(\mathbf{y}) = \{\bullet\}$, then all memory cells reachable from $\lambda(\mathbf{y})$ are definitely not reachable from globals and thus not accessible from other threads.

[1] The third condition does not follow from the previous two when $\rho(\mathbf{y}) \cap \rho(\mathbf{y}') = \{\bullet\}$.

Accordingly, write accesses through $\mathbf{y}$ need not be protected. If on the other hand, $\rho(\mathbf{y})$ contains a global static address, the address of $\mathbf{y}$ must be considered as *published*, i.e., possibly accessible for other threads. The set of static global addresses occurring in ρ (and π) can be considered as the set of *possible owners* of a region for which locks should be provided. The following theorem states that our definitions of the abstract transformers for basic program statements are sound.

Theorem 5 (Soundness of Transfer Functions)**.** *Let s denote a program statement and T denote an abstract state. If $\sigma \in \gamma(T)$ and σ' denotes the concrete program state obtained from σ by the execution of s, then $\sigma' \in \gamma([\![s]\!]^\sharp T)$.*

Proof. We consider only the assignment $\mathbf{y} \to b = \mathbf{y}'$ where the region structure changes. The concrete semantics for this assignment updates the heap mapping as follows:

$$[\![\mathbf{y} \to b = \mathbf{y}']\!]\ \langle \mu, \lambda \rangle = \langle \mu', \lambda \rangle \quad \text{where} \quad \mu' = \mu[\langle \lambda(\mathbf{y}), b \rangle : \lambda(\mathbf{y}')]$$

In terms of the heap graph, this introduces an edge between the block pointed to by $\mathbf{y}$ to the block pointed to by $\mathbf{y}'$. We verify that the four conditions for our abstraction are preserved by the abstract transfer function $[\![\mathbf{y} \to b = \mathbf{y}']\!]^\sharp\ \langle \pi, \rho \rangle = \langle \pi', \rho' \rangle$.

First, assume distinct globals x and x' are in the same region of the updated heap μ'. If there was an (undirected) path in μ which connected x and x', then $\{x, x'\} \in \pi \subseteq \pi'$. If they were not weakly connected in π, but are so in π', the field $\lambda(\mathbf{y})\,.\,b$ must have connected them, hence there must have been a path in μ from one of these globals, say x, to $\lambda(\mathbf{y})$ and a path from the other, x', to $\lambda(\mathbf{y}')$. Then, from the second concretization condition, we have $x \in \rho(\mathbf{y})$ and $x' \in \rho(\mathbf{y}')$, and as the transfer function collapses the region for all pairs of elements in $\rho(\mathbf{y}) \cup \rho(\mathbf{y}')$, we have $\{x, x'\} \in \pi'$.

Second, assume a global x and local variable $\mathbf{y}_0$ are in the same region of μ'. Again, either $x \in \rho(\mathbf{y}_0) \setminus \{\bullet\} \subseteq \rho'(\mathbf{y}_0)$, or x and $\lambda(\mathbf{y}_0)$ are connected through the field $\lambda(\mathbf{y})\,.\,b$, hence we have the paths in μ from say x to $\lambda(\mathbf{y})$ and from $\lambda(\mathbf{y}_0)$ to $\lambda(\mathbf{y}')$. We conclude $x \in \rho'(\mathbf{y})$ as before, and consider the pointer $\mathbf{y}_0$. Assume first that $\mathbf{y}_0 \neq \mathbf{y}'$, then from the third concretization condition, we have $\rho(\mathbf{y}') \cap \rho(\mathbf{y}_0) \neq \emptyset$. Therefore, $\rho(\mathbf{y}_0)$ either contains the bullet or one of the globals in $\rho(\mathbf{y}')$. Since all globals in $\rho(\mathbf{y}')$ are joined together with $\rho(\mathbf{y})$ into an equivalence class containing x and all instances of the bullet in ρ' are replaced by this class, we conclude $x \in \rho'(\mathbf{y}_0)$. Now, if $\mathbf{y}_0 = \mathbf{y}'$, the same argument applies, but we instead rely on the fourth concretization condition to ensure that $\rho(\mathbf{y}_0)$ contains something which is merged with the equivalence class of x.

Third, assume local variables $\mathbf{y}_1$ and $\mathbf{y}_2$ are in the same region of μ'. Then, as before, the condition is either trivially preserved, or through the field $\lambda(\mathbf{y})\,.\,b$,

hence we have $\lambda(\mathbf{y}_1) \cap \lambda(\mathbf{y}) \neq \emptyset$ and $\lambda(\mathbf{y}_2) \cap \lambda(\mathbf{y}') \neq \emptyset$. Then, $\lambda(\mathbf{y}_1) \cap \lambda(\mathbf{y}_2) \neq \emptyset$ again follows because all globals in $\lambda(y) \cup \lambda(\mathbf{y}')$ are in the same equivalence class in π' and all bullets in ρ' are substituted by this class.

Finally, assume $\lambda(\mathbf{y}_0) \neq \mathsf{null}$ for some $\mathbf{y}_0$. Then, $\rho(\mathbf{y}_0) \neq \emptyset$ and since $\forall \mathbf{y} : \rho(\mathbf{y}) \subseteq \rho'(\mathbf{y})$, we have $\rho'(\mathbf{y}_0) \neq \emptyset$. □

4.3 Interprocedural analysis

In this section we present an interprocedural formulation of the region analysis. We model communication between procedures by assuming that every function has the same set L of local variables and that all locals of the caller are passed by value to the callee; however, in our simplified setting, we only pass locals into procedures but do not return them back. Thus, the effect of a procedure call is limited to possible collapses within the partition of globals and the possible joining of thread-local data structures with some global regions. As the state of the heap is shared between all threads, the partitioning is part of the global invariant and accumulated flow-insensitively as described in Section 2.4. In order to deal with the thread-local fresh region, we extend the points-into map ρ for local pointer variables with an extra variable $\diamond$ representing the thread-local data structures before the call. The abstract transformer $\mathsf{enter}^\sharp$ initializes the abstract state at procedure entry based on the abstract state before the call:

$$\mathsf{enter}^\sharp(\rho) = \rho[\diamond : \bullet]$$

While analyzing a procedure q, updates through pointers into thread-local memory may result in globals being added to the region tracked by $\diamond$ (just as for any other variable with $\bullet$ in its points-into set). At procedure exit, the local variables of the called procedure q are removed, while the points-into information accumulated by $\diamond$ are added to every local $\mathbf{y}$ of the caller which before the call may have pointed into the thread-local region. Assume that $T_1 = \langle \pi_1, \rho_1 \rangle, T_2 = \langle \pi_2, \rho_2 \rangle$ are the abstract states before the call and at procedure exit, respectively. Then this combination is achieved by the function $\mathsf{combine}^\sharp$:

$$\begin{aligned} \mathsf{combine}^\sharp(\rho_1, \rho_2) &= \mathsf{cl}_{\pi_2}\rho \quad \text{where} \\ &\rho = \rho_1[z : \rho_1(z) \cup \rho_2(\diamond)]_{z \in L \cup \{\diamond\}, \bullet \in \rho_1(z)} \end{aligned}$$

The abstract functions $\mathsf{enter}^\sharp$ and $\mathsf{combine}^\sharp$ allow us to the general frameworks for interprocedural analysis from Section 2.3. To recapitulate briefly, the analyzer solves a constraint system for the abstract values returned by the summary function for f when called on abstract values a. Given a complete lattice $\mathcal{L}$ of abstract

values, abstract transformers $[\![s]\!]^\sharp$ for basic statements, and abstract transformers $\mathsf{enter}^\sharp$ and $\mathsf{combine}^\sharp$ for parameter passing and function return, the constraint system was set up as follows:

$$\begin{aligned}
\langle e_f, a\rangle &\sqsupseteq a && \text{for an entry node } e_f\\
\langle \pi, \langle v, a\rangle\rangle &\sqsupseteq [\![s]\!]^\sharp\ \langle \pi, \langle u, a\rangle\rangle && \text{for edge } (u, s, v)\\
\langle v, a\rangle &\sqsupseteq \mathsf{combine}^\sharp(\langle u, a\rangle, \langle r_f, \mathsf{enter}^\sharp\langle u, a\rangle\rangle) && \text{for edge } (u, f(), v)
\end{aligned}$$

where $a \in \mathcal{L}$, f denotes functions with return point r_f, and u, v are program points. For a program point v of a function g, the variable $\langle v, a\rangle$ of the constraint system represents the abstract value attained at v in a call to g where evaluation of the body of g starts with the abstract value a. The region partition is here a global invariant, i.e., $\Psi = \pi$, and handled through the side-effecting framework described in Section 2.4. The soundness of the least solution of this constraint system instantiated to our region analysis follows from Theorem 5 and the soundness results for the inter-procedural multi-threaded analysis framework [22, 51, 76]:

Theorem 6 (Soundness of Region Analysis)**.** *Assume that $\varphi\langle v, a\rangle$, for program point v of a procedure f and abstract state a, is the least solution of the constraint system over the complete lattice $\mathcal{L}$. Let $\varphi\langle v, a_e\rangle = \langle \pi, \rho\rangle$, and assume that the pair $\sigma_e = \langle \mu_e, \lambda_e\rangle$ of a heap μ_e and assignment λ_e of locals is in the concretization of a_e, i.e., $\sigma_e \in \gamma(a_e)$. Moreover, assume that R_e is the set of thread-local memory cells at procedure entry, i.e., the set of addresses which can only be reached from the locals in σ_e.*

Then every same-level execution starting in σ_e at the entry point of f and reaching program point v in state $\sigma = \langle \mu, \lambda\rangle$ satisfies the following properties:

- *$\sigma \in \gamma(\langle \pi, \rho\rangle)$;*
- *For every global x, if x is reachable from an address in R_e (w.r.t. μ), or an address in R_e is reachable from x (w.r.t. μ), then $\&x \in \rho(\diamond)$.* □

The given constraint system may be huge depending on the complete lattice of the analysis. *Local fixpoint iteration* is a general technique to partially explore large (or possibly infinite) systems of constraints [29]. Starting from a subset Y of *interesting* unknowns, local fixpoint iteration explores only those other unknowns which may *contribute* to the values of unknowns from Y. This technique is well-suited if the interesting values can be computed by consulting only a small (though possibly unknown) fraction of the constraint variables. This is the case in our application. Here, fixpoint iteration starts with the set $Y = \{\langle r_{\mathsf{main}}, \mathsf{enter}^\sharp\, a\rangle\}$ if main is the start function of the thread currently under consideration, and the abstract value a describes the program state before program execution [29]. Local

fixpoint iteration then will trigger the evaluation of all pairs $\langle v, \mathsf{enter}^\sharp a' \rangle$ where v is the program point of a procedure which (during fixpoint iteration) is called for the abstract program state a'. In our experiments with the analyzer Goblint, we found that the number of different calls of the same procedure is for most programs quite small.

4.4 Relating locks and regions

In order to relate accessed regions of the global memory with acquired locks, we rely the analysis providing *must-alias* information for static global addresses from the previous chapter. For clarity of presentation, however, we will not use the full Herbrand and affine equalities, but consider the simplest instance of such an analysis, which tracks conjunctions of equalities of the form $\mathbf{y} \doteq x$ where $\mathbf{y} \in L$ is a local pointer variable and $x \in L \cup G$ is either a local pointer variable or a global static address. Such a domain has been suggested in [59] where efficient algorithms for the basic operations have been presented.

Let $\mathbf{E}$ denote the lattice of equalities. Technically, each element $\phi \in \mathbf{E}$ either is equivalent to false or is equivalent to a satisfiable finite conjunction of equalities. We write $\phi \models (x \doteq x')$ if the equality $x \doteq x'$ is logically implied by ϕ. The ordering on $\mathbf{E}$ is given by logical implication, i.e., $\phi \sqsubseteq \phi'$ iff either $\phi = \mathsf{false}$ or both ϕ and ϕ' are different from false, and $\phi \models (x \doteq x')$ for every equality $x \doteq x'$ in ϕ'. Thus, the greatest lower bound of ϕ_1, ϕ_2 is given by their conjunction $\phi_1 \wedge \phi_2$, whereas the least upper bound of two satisfiable conjunctions ϕ_1, ϕ_2 is equivalent to the conjunction of all equalities $x \doteq x'$ which are both implied by ϕ_1 and ϕ_2. Here, we consider the abstract functions for procedure calls. According to our assumption, all locals are passed as actual parameters to called procedures. The locals of the caller, on the other hand, are not affected by the changes to locals of the callee. This means that the abstract functions $\mathsf{enter}^\sharp_E, \mathsf{combine}^\sharp_E$ for procedure calls are defined by:

$$\mathsf{enter}^\sharp_E \, \phi = \phi \qquad \mathsf{combine}^\sharp_E(\phi_1, \_) = \phi_1$$

As a third component, our analysis requires information about the set of locks which are definitely held when reaching a program point. For the moment, every lock is identified by static addresses or addresses pointed at by local pointers. For every reachable program point u (in every analyzed invocation of a procedure), our analysis therefore identifies a finite subset S of descriptions of locks which are definitely held when reaching u (in the given invocation). Let $\mathcal{M}$ denote the set of finite subsets of global static addresses of locks. Since we are interested in *definite* information, finite sets of lock address expressions are ordered by the *superset* relation.

While region and must-alias analysis are independent, the analysis of sets of definitely held locks may profit from the results of both. The must-alias analysis is applied to identify all address expressions which denote the acquired lock, the may-alias information which we infer from the region information, helps to narrow down the set of locks which may no longer be held after releasing a lock. More precisely, assume that $T = \langle \pi, \rho \rangle$ is an abstract description of memory regions. We infer non-equality information as follows. If $\{x, x'\} \notin \pi$ for two pointer expressions x, x', then $x \neq x'$ for every program state $\langle \mu, \lambda \rangle$ in the concretization of π. Likewise, if $x \notin \rho(\mathbf{y})$, then also $\lambda(\mathbf{y}) \neq x$. Finally, if $\rho(\mathbf{y}) \cap \rho(\mathbf{y}') = \emptyset$ while $\rho(\mathbf{y}) \cup \rho(\mathbf{y}') \neq \emptyset$, then also $\lambda(\mathbf{y}) \neq \lambda(\mathbf{y}')$. We denote these facts by $T \models (x \neq x')$, $T \models (\mathbf{y} \neq x)$ and $T \models (\mathbf{y} \neq \mathbf{y}')$, respectively.

Assume that the current program state $T = \langle \pi, \rho, \phi, S \rangle$ consists of the partition of globals π, the points-into information ρ, the conjunction of must-equalities ϕ, and the lock set S. Then the sets of definitely held locks after operations lock and unlock for locks inside static structs are defined by:

$$[\![\mathsf{lock}(\&(z \to b)]\!]^\sharp_S T = S \cup \{\&(x \to b) \mid x \in G, \phi \models z \doteq x\}$$
$$[\![\mathsf{unlock}(\&(z \to b))]\!]^\sharp_S T = S \setminus \{\&(x \to b) \mid \neg(\pi \models z \neq x)\}$$

for $z \in L \cup G$, respectively. When entering or leaving a procedure, the set of definitely held locks does not change. Therefore, we have:

$$\mathsf{enter}^\sharp_L\, S = S \qquad\qquad \mathsf{combine}^\sharp_L(\_, S_2) = S_2$$

4.5 Extension with arrays

So far, our analysis is able to deal with dynamic data structures and a fixed finite set of mutexes. In the next step, we extend this base approach to global data structures which may contain arrays and thus also arrays of mutexes.

Example 15. Figure 4.3 shows a simplified version of the insert-function from the access vector cache of Security Enhanced Linux.[2] At every program point, at most one lock is held which is taken from a possibly large set of locks contained in the array `locks`. For a sound data-race analysis of the function `insert`, it does not suffice to verify that *some* lock from this array is held when the hash map is modified. Instead, it also must check the (statically unknown) *index* of the lock coincides with the index of the list in `slots`. □

[2]The most notable simplification is the use of singly linked lists instead of the doubly linked lists from the Linux kernel; however, since our technique is based on a conservative partitioning of the heap into disjoint regions, dealing with doubly linked lists and even structured use of pointer arithmetic poses no significant further challenge.

```
struct list { int key; int data; struct list *next; };
struct list *slots[512];
spinlock_t   locks[512];

struct list *insert(int key, int data) {
    struct list *t; int hv = hash(key);
    spin_lock(&locks[hv]);
    t = slots[hv];
    if (t == NULL) {
        slots[hv] = new_list(key, data); goto fd; }
    while(1) {
        if (t→key == key) {
            t→data = data; goto fd; }
        if (t→next == NULL) {
            t→next = new_list(key, data); goto fd; }
        t = t→next; }
fd: spin_unlock(&locks[hv]);
    return t; }
```

Figure 4.3: Simplified `insert`-function.

We now extend our core language by additionally allowing arrays within global shared data structures. Here, we consider non-nested arrays only. The address of a memory cell from a *static* global data structure with arrays is identified by $\&a[i]$ where i is an index. Accordingly, we consider address expressions of the form $\&a[e]$ where e is a side-effect free index expression depending on **int**-variables only. We assume each thread has a unique set of variables. Furthermore, we extend our notion of abstract heap partitions π and points-into maps ρ. Besides sets of two-element sets, we now also allow *singleton* sets $\{\&a\}$ in partitions. Such a singleton indicates that different entries of the array $\&a$ may belong to the same memory region. We thus consider the set $\mathbf{P}$ of abstract heap partitions π with the following properties:

1. If $\{x, y\}, \{y, z\} \in \pi$ for $x \neq z$, then $\{x, z\} \in \pi$.
2. If $\{\&a, x\} \in \pi$, then also $\{\&a\} \in \pi$.
3. If $\{\&a\} \in \pi$, then $\&a[e]$ does not occur in π.
4. For the same array $\&a$, π may have at most one address expression e with $\&a[e]$ occurring in π.

We could have allowed multiple index expressions e_i referring to the same array $\&a$ as long as all e_i definitely evaluate to distinct values. In our experiments, the restriction to a single expression, however, has always been sufficient. The partial ordering on $\mathbf{P}$ is given by $\pi_1 \sqsubseteq \pi_2$ iff the following holds:

1. If $\{\&a[e], x\} \in \pi_1$ then $\{\&a[e], x\} \in \pi_2$ or $\{\&a\}, \{\&a, x\} \in \pi_2$.
2. If $\{x, y\} \in \pi_1$ where neither x nor y contains an index expression, then also $\{x, y\} \in \pi_2$.

Thus, e.g., for $\pi_1 = \emptyset$, $\pi_2 = \{\{\mathbf{p}, \&a[\mathbf{i}]\}\}$, $\pi_3 = \{\{\&a\}, \{\mathbf{p}, \&a\}\}$, $\pi_1 \sqsubseteq \pi_2 \sqsubseteq \pi_3$.

Accordingly, we now consider points-into maps ρ where a set X occurring as the image of a local (or $\diamond$) satisfies the following additional restrictions:

1. If $\&a[e], \&a[e'] \in X$, then $e \equiv e'$;
2. If $\&a \in X$ then for every e, $\&a[e] \notin X$

where the ordering on two such sets is the natural extension of $\emptyset \sqsubseteq \{x\}$ for all x, and $\{\&a[e]\} \sqsubseteq \{\&a\}$.

Also, we extend the closure operation cl_π such that $\mathsf{cl}_\pi X$ for a set X of global static address expressions or $\bullet$, now additionally replaces an indexed expression $\&a[e]$ with $\&a$ whenever $\{\&a\} \in \pi$. Likewise, we extend the domain of must equalities and finite lock sets to address expressions containing indexing. The occurring index expressions may depend on $\mathbf{int}$-variables; however, we here ignore definite equalities between $\mathbf{int}$-variables. Thus, we consider two index expressions e_1, e_2 as *definitely* equal only if they are *syntactically* equal. Technically, this allows us to use a similar domain for must equalities and lock sets as in section 4.4 — only that we now additionally consider indexed static addresses $\&a[e]$ instead of static addresses $\&a$ alone.

This simplistic setting is still able to deal with increments or decrements of $\mathbf{int}$-variables. Accordingly, our analysis will track assignments to $\mathbf{int}$-variables $\mathbf{i}$ of the form $\mathbf{i} = \mathbf{i} + c$ for $c \in \mathbf{Z}$ whereas all other assignments to $\mathbf{i}$ are approximated by the *non-deterministic* assignment $\mathbf{i} = ?$ which is meant to assign to $\mathbf{i}$ an *unknown* value. The effect of the assignment $\mathbf{i} = \mathbf{i} + c$ on a triple $T = \langle \pi, \rho, \phi, S \rangle$ consists in substituting $\mathbf{i}$ in all index expressions occurring in T with $\mathbf{i} - c$. The effect of the assignment $\mathbf{i} = ?$ on the other hand, assigns an unknown value to $\mathbf{i}$ and thus must remove all occurrences of $\mathbf{i}$ from T. For a partition π, $\mathsf{delete}(\pi, \mathbf{i})$ replaces all expressions $\&a[e]$ where $\mathbf{i}$ occurs in e with $\&a$ (if there are any) and adds the set $\{\&a\}$ (given that there are any). For a points-into map ρ, $\mathsf{delete}(\pi, \mathbf{i})$ replaces in every image $\rho(z)$ elements $\&a[e]$ where $\mathbf{i}$ occurs in e with $\&a$. For

component ϕ, $\mathsf{delete}(\phi, \mathbf{i})$ removes all equalities involving $\mathbf{i}$. Likewise for S, $\mathsf{delete}(S, \mathbf{i})$ removes all lock expressions $\&a[e].b$ where $\mathbf{i}$ occurs in e.

$$[\![\mathbf{i} = \mathbf{i} + c]\!]^\sharp T = T[\mathbf{i} - c/\mathbf{i}]$$
$$[\![\mathbf{i} = ?]\!]^\sharp T = \langle \mathsf{delete}(\pi, \mathbf{i}), \mathsf{delete}(\rho, \mathbf{i}), \mathsf{delete}(\phi, \mathbf{i}), \mathsf{delete}(S, \mathbf{i}) \rangle$$

The effects of assignments involving local pointers and global memory, are defined componentwise on the first three components, while the set of definitely held locks remains unchanged. We omit the details but instead apply the technique to a typical example.

Example 16. Assume we start the execution of the insert-function from Figure 4.3 with the abstract value $T_0 = \langle \emptyset, \{\diamond \mapsto \{\bullet\}, \mathbf{t} \mapsto \emptyset\}, \mathsf{true}, \emptyset \rangle$. After having called `spin_lock()` and reaching the *while*-loop, we have:

$$\begin{aligned} T_1 &= \langle \emptyset, \rho_1, \phi_1, S_1 \rangle \qquad \text{where} \\ \rho_1 &= \{\diamond \mapsto \{\bullet\}, \mathbf{t} \mapsto \{\&\mathtt{slots}[\mathbf{hv}]\}\} \\ \phi_1 &= \mathbf{t} \doteq \&\mathtt{slots}[\mathbf{hv}] \\ S_1 &= \{\&\mathtt{locks}[\mathbf{hv}]\} \end{aligned}$$

although the precise value of **hv** is unknown. Inside the loop the must-equality $\mathbf{t} \doteq \&\mathtt{slots}[\mathbf{hv}]$ is lost, while the region information as well as the lock set are preserved. Unlocking resets the set of held locks to $\emptyset$. □

Our analysis can be enhanced by jointly performing constant propagation or, more generally, any analysis of **int** variables which provides us with more precise information about how index expressions are related. Such information is provided by analysis from the previous chapter.

While the complete lattice for the combined analysis of regions, must equalities and abstract lock sets in presence of arrays is no longer finite, it still satisfies the ascending chain condition. In order to apply the interprocedural framework from Section 4.3, we generalize the functions $\mathsf{enter}^\sharp$ and $\mathsf{combine}^\sharp$ for abstract parameter passing and procedure return from the last sections. Additionally, we now must track the values of local **int** variables. We could do so by additionally maintaining, e.g., affine must equalities between these, and apply the same technique bullet trick from Section 3.7. Here, for simplicity and to avoid an over-reliance on bullets, we prefer a simpler analysis which just tracks the set of local **int** variables which may have changed their values since procedure entry.

Assume that before the call, we have the abstract state $T = \langle \pi, \rho, I, \phi, S \rangle$ where π, ρ, ϕ, and S are as before and I now denotes a set of **int** variables whose values have possibly changed since procedure entry. When entering a newly called

procedure, we initialize this set to $\emptyset$. Recall that the π-component is handled through side-effecting, so for the other components, we define

$$\mathsf{enter}^\sharp\langle\rho, I, \phi, S\rangle = \langle\rho_1, \emptyset, \phi, S\rangle \quad \text{where}$$
$$\rho_1 = \rho[\diamond : \bullet]$$

Likewise, at procedure exit, the local variables of the called procedure q must be removed. Also all equivalences $\{x, \&a[e]\}$ in the returned must be collapsed to $\{x, \&a\}$ for index expressions e depending on **int**-variables which have changed their values. This is achieved by:

$$\mathsf{combine}^\sharp(\langle\rho_1, I_1, \phi_1, S_1\rangle, \langle\rho_2, I_2, \_, S_2\rangle) = \langle\mathsf{cl}_\pi\rho, I_1, \phi_1, S\rangle \text{ where}$$
$$\rho = \rho_1[z : \rho_1(z) \cup \rho_2(\diamond)]_{z \in L \cup \{\diamond\}, \bullet \in \rho_1(z)}$$
$$S = \mathsf{delete}(S_2, I_2)$$

Here, the calls to $\mathsf{delete}()$ for a set I of **int** variables abbreviate repeated application of $\mathsf{delete}()$ for each element $\mathbf{i} \in I$.

Example 17. Consider the insert-function from Figure 4.3. Assume that at the program point before the call to this function we have the abstract state: $T_0 = \langle\{\diamond, \mathbf{t} \mapsto \{\bullet\}\}, \emptyset, \mathsf{true}, \emptyset\rangle$, where we omit the π-component since it does not change in this example. Then $\mathsf{enter}^\sharp(T_0) = T_1$ is the abstract value for the start point of the corresponding abstract call to the function `insert()` where:

$$T_1 = \langle\rho_1, \emptyset, \phi_1, \emptyset\rangle$$
$$\rho_1 = \{\diamond \mapsto \{\bullet\}, \mathbf{t} \mapsto \{\&\mathtt{slots}[\mathbf{hv}]\}\}$$
$$\phi_1 = \mathbf{t} \doteq \&\mathtt{slots}[\mathbf{hv}]$$

At the program point before the lock operation, we have $T_2 = \langle\rho_1, \{\mathbf{hv}\}, \phi_1, \emptyset\rangle$. After locking, we thus have $T_3 = \langle\rho_1, \{\mathbf{hv}\}, \phi_1, \{\&\mathtt{locks}[\mathbf{hv}]\}\rangle$ — implying that the elements accessed through the pointer $\mathbf{t}$ belong to the region $\mathtt{slots}[\mathbf{hv}]$ and that these accesses are protected by the corresponding lock $\mathtt{locks}[\mathbf{hv}]$. At function exit, we finally arrive at $T_4 = \langle\rho_1, \{\mathbf{hv}\}, \phi_1, \emptyset\rangle$. Combining this state with the state T_0 before the call will recover the set of possibly modified **int** variables as well as the must equalities before the call. In the example, we just recover the abstract state T_0. □

4.6 Analyzing the Linux kernel

We have implemented our analysis in the Goblint analyser and applied it to Linux kernel modules such as device drivers. One challenge in analyzing device drivers

File	Size (merged)	Time	Verified	Warnings
`atmel_tclib`	1317 lines	0,07 s	1	0
`hwmon`	1434 lines	0,23 s	1	0
`enclosure`	1510 lines	0,19 s	1	1
`scsi_dh`	4370 lines	0,57 s	2	0
`dmaengine`	4449 lines	0,83 s	3	0
`scsi_rdac`	4744 lines	0,81 s	1	0
`usb_hcd`	7340 lines	3,32 s	3	2
`avc`	7466 lines	1,68 s	2	1
`ppp_generic`	10818 lines	4,70 s	4	1

Table 4.1: Result of analysing kernel modules

is how to model the rest of the kernel. Goblint uses a driver *harness* that assumes the worst possible interleavings of the device's file operations and interrupt handlers. Starting from the module initialization code, we track function pointers that are held in structs. Pointers passed to library functions are assumed to be potential call-backs and are analyzed as separate threads. These may interleave with each other as well as with the rest of the initialization code.

In the implementation, we also extended the basic approach to deal with nested static global data-structures such as structs containing arrays as well as well-behaved pointer arithmetic within structs. This is necessary for the analysis of the Linux API for doubly linked lists. This API provides macros which, e.g., calculate the start address of a struct from the address of a component. While these macros have a clean semantics, their implementation makes extensive use of type casts, and addition and subtraction of pointers. Therefore, our implementation allows application of the address operator to arbitrary expressions evaluating to global addresses. Thus, pointers may no longer point to the beginnings of blocks. Moreover, a pointer variable whose value is obtained from the value of the pointer variable q by means of such kind of pointer arithmetic is put into the same region as q.

The results of running our analyzer on a number of different modules from the kernel is summarized in Table 4.1. We use the CIL analysis framework [63] as a front-end to parse and process these files. The sizes of the files in the table are the sizes of CIL's outputs after merging the modules with included headers and removing unused definitions. We ran these experiments on an Athlon 64 X2 3800+ machine under Kubuntu.[3]

For all these benchmarks, we are successful in automatically inferring the correlations between elements of lists and their corresponding locks and to verify that

[3]The goblint website, `http://goblint.at.mt.ut.ee`, has detailed instructions on reproducing these benchmarks.

all accesses are protected. The numbers of shared variables for which we could verify a consistent locking scheme as well as those for which conflicting accesses were found are listed in the table. The analyzer registers accesses to each element in a region separately; thus, if k linked lists have collapsed into a single region and there is a conflicting access through a pointer into this region, the number of warnings would be k and not one. The false alarms for these benchmarks are mostly due to our imprecise harness. We will comment here only on two interesting benchmarks. The file `avc` is the access vector cache code of Security Enhanced Linux which served as the inspiration for the examples in this chapter. The analyzer's output is the following:

```
Found correlation: avc_cache.latest_notif is guarded by
  lockset {notif_lock}
Found correlation: avc_cache.slots is guarded by
  lockset {avc_cache.slots_lock[*]}
Datarace over avc_callbacks:
  write in some thread with lockset: {} (avc.c:6953)
```

The asterisk in the second lockset is the analyzer's modest way of indicating that it has verified the correlation between the index expressions used when accessing list elements in the array of slots and the index expressions used to acquire a mutex from the array of locks. The analyzer warns about a "race" for `avc_callbacks`. While this is indeed a race in the context of this module alone, the function for registering callbacks are only used in the initialization code by the files using this module.

The file `dmaengine` is part of the hardware-neutral interface to the DMA subsystem. The programmers have commented in the source file: "The subsystem keeps two global lists, `dma_device_list` and `dma_client_list`. Both of these are protected by a mutex, `dma_list_mutex`." Our analyzer succeeds in verifying this.

4.7 Related work

Regions and ownership types have been used for compile-time garbage collection [85] or to ensure encapsulation in object-oriented languages [17]. More recently, analyzers have been developed for checking correct usage of region-based memory management APIs [9, 90]. Note, however, that the regions there need not be closed under reachability. For analyzing pointers, Gulwani and Tiwari [38] present a domain of quantified may- and must-equality pairs which can express similar invariants to ours. This analysis, while being extremely precise, has problems with dealing with doubly linked lists. Reachability in the presence of pointer

Test	Goblint	Locksmith	Coverity	DDVerify
static	+/+	+/+	+/+	−/+
single list	+/+	+/−	−/+	
shared lists	+/+	+/−	−/+	
simple array	+/+	+/−	−/+	
shared array	+/+	+/−	−/+	

Table 4.2: Summary of comparison. For each idiom, "+" indicates success, while "–" indicates the existence of a False Negative / False Positive.

arithmetic has been studied by Chatterjee et al. [15] who provide an annotation language for reasoning about the linked list API of Windows device drivers.

Precise abstractions of the heap have been provided by separation logic [71] and shape analysis [75]. Gopan et al. [33] present a shape analysis which allows reasoning about dynamic memory and the values of array elements, Gulwani et al. [35] present a *set cardinality analysis* which combines shape and numeric abstractions to reason about sizes of data-structures. Hackett and Rugina [40] present a shape analysis which is built on top of a partitioning of the heap into disjoint regions. These regions are derived from a standard points-to analysis and are again not necessarily closed under reachability. Recent work has also provided methods for making shape analysis scale better [11, 55, 93] — at a certain loss in precision, e.g., by no longer tracking arrays.

Our main interest has been to provide efficient methods which are precise enough for analyzing data races in presence of dynamic data-structures and arrays. Rugina and Rinard [74] present techniques to avoid races by analyzing disjointness of accessed memory blocks. Naik and Aiken [62] propose *conditional must-not aliasing* to deal with locking schemes of various levels of granularity in Java. They introduce *disjoint reachability* analysis for dealing with medium-grained locking; however, their notion of disjointness is based on allocation sites, which is not helpful in cases such as Figure 4.2. We have experimented with some analyzers that perform race detection for C. We compared the following analyzers: Locksmith, a sound race detection tool based on type-based label-flow [68]; Coverity Prevent, a commercial bug-detection tool based on meta-compilation techniques [27]; and DDVerify, a device driver model-checker that checks for proper use of the kernel API [92].

We compared the tools on small test programs. For each test, there is a version with a race and one without races. The test *static* is the simplest possible race example, has a static global variable that should be protected by a static lock; *single list* contains a linked list where access to its nodes are protected by a single lock; *shared lists* has two lists that are protected by their own locks; still, there

might be races due to sharing between elements in the lists; *simple array* contains an array of locks and an array of linked lists where the accesses should be properly correlated as in the examples of this chapter; *shared array* is like the previous test, except there might be sharing between the linked lists of different array elements, hence there may be a race although the correct lock is acquired. The summary of this comparison is shown in Table 4.2.

It seems that DDVerify checks other properties related to mutexes, e.g., double-acquisition of locks, but not whether accesses to globals are protected by the same locks. Locksmith and Coverity Prevent pass the first test, but already the simple linked list example is beyond their current capabilities. Locksmith complains on all tests, even when the program is perfectly safe; Coverity remains completely silent, even in the presence of races. Naturally, these analyzers have their advantages: Coverity checks a host of other properties, Locksmith deals with per-element locking, and DDVerify has an extremely precise automatic device driver harness mechanism; nevertheless, for medium-grained locking, Goblint is the clear winner.

4.8 Conclusion

We have presented a general approach to certify absence of data-races in C. In order to deal with dynamic data-structures, we provided a simple region analysis which allows to analyze reachability through field selection and dereferencing. We also indicated how this method can be extended to deal with arrays of regions and (well-behaved) pointer arithmetic. Our methods have been implemented in the efficient interprocedural data-race analyzer Goblint allowing us to verify locking schemes for dynamic data structures and arrays in the Linux kernel.

While we have analyzed benchmarks without modifying the original kernel code, in four of the benchmarks we only considered conflicts between write accesses. Read accesses are often protected by reader/writer locks, or more recently, the Read-Copy-Update mechanism. This poses a problem when the read accesses are protected at a coarser level of granularity than that of the write accesses. Thus, our failure to distinguish these would generate false alarms. Another challenge is to combine our technique here with methods dealing with per-element locking [67] in order to verify programs where some dynamically allocated structures, such as the per-device structure, contain linked lists and associated mutexes.

CHAPTER 5

HEAP INVARIANTS THAT ARE NOT

Elements migrating from a region of the heap protected by one mutex to another region is frequent in low-level C code. So far we have presented abstractions of the heap required to hold globally during the entire program execution. These techniques break down when elements migrate, since the invariant is temporarily violated. Therefore, we now consider a shape analysis which allows to jointly infer the shape of regions together with the locks by which they are protected, and we show that this analysis is able to deal with element migration. In addition, we propose a scheme to combine this expensive analysis with the light-weight region analysis to identify the sections of code where the full precision of shape analysis is required.

5.1 Introduction

In many practical applications, the global invariant required for proving absence of data-races is *temporarily violated*. This leads to false alarms for our analyzer because we rely on a single heap abstraction to over-approximate the state of the shared memory throughout the concurrent program execution.

We identified *migration* as a particularly common source of such temporary violations. Migration happens when an object is extracted from one protected resource and afterwards either accessed without locking or added to another protected resource. Probably the most common instance of migration is moving objects to and from a free-list. Migration is related to the problem of unsharing, which was identified as source of false alarms for the Relay analyzer by Voung et al. [89].

In order to remove these false alarms, one could apply a full-fledged *concurrent* shape-analysis, like [6, 34, 55]. Such analyses are designed to deal with local violations, but are at the same time prohibitively expensive for larger programs.

```
1  void generate() {
2    p = malloc();
3    lock(A_lock);
4    list_add(p,&A);
5    unlock(A_lock);
6  }
```

```
7  void process() {
8    lock(A_lock);
9    q = A→next;
10   lock(B_lock);
11   list_move(q,&B);
12   unlock(B_lock);
13   unlock(A_lock);
14 }
```

```
15 void dispose() {
16   lock(B_lock);
17   r = B→next;
18   list_del(r);
19   unlock(B_lock);
20   free(r);
21 }
```

Figure 5.1: Moving an element from one list to the other.

In this work, we present a light-weight techniques to prove race freedom for programs with migration between resources which is significantly more scalable than concurrent shape analyses for applications such as Linux device drivers written in C.

Challenging example. The example in Fig 5.1 illustrates migration issues. The program contains a generator thread, which adds elements into a shared linked list A. The processing thread takes elements out of the work list, processes them in private, and inserts them into another list. The portion of code shown for each thread should be thought of as part of a loop. We have also omitted checks of emptiness from the code. The list macros used are taken from `list.h` of a standard Linux distribution.

The program is perfectly safe and race-free: An element is produced by the generator thread while acquiring a lock for list A. The processing thread acquires both locks and moves an element to the free-list B for disposal. The clean-up thread acquires the lock for B and removes and frees elements.

The challenge is to prove race freedom for similar programs with migration. Complications arise when a thread maintains a persistent pointer into the global structures, i.e., a pointer variable which is live at the end of an atomic section. For example, if the generator thread after line 5 would re-acquire `lock_A` and undo the addition of the element with `list_del(p)`. The program would be unsafe because the element pointed to by p may have already been moved to list B. Another challenge is sharing which is accidentally introduced between the two lists, e.g., when line 11 is replaced with `list_add(q,&B)`. Then, one can no longer be certain that traversing list A will only lead to accesses of elements in A. Acquiring the lock for A is therefore not enough to ensure mutual exclusion, as the same elements may be accessed by a thread traversing list B.

We want to stress, that, for this work, we are not concerned with properties like memory safety, but focus on race detection only.

Contributions. When programs communicate through global variables, one efficient approach to thread-modular analysis is the *side-effecting* discussed in Section 2.4. Here, we refine the side-effecting scheme to take into account mutually exclusive accesses. When a thread has ensured exclusive access to a variable, the entire sequence of changes to the variable can be considered as a single effect rather than having to accumulate and join each update with other threads individually. To that end, in Section 5.2, we refine the side-effecting framework to allow non-relational value domains to violate the global invariant within an atomic section. A key feature of the framework is that it *simultaneously* infers invariants on shared variables as well as the locks that definitely protect them. This is crucial since race-freedom may depend on an invariant to hold, while at the same time the invariant only holds if the program is free from races (see example in Section 5.2). Simultaneous inference sets us apart from approaches like [16, 34] that exploit a-priori computed lock sets.

In Section 5.3, we adapt the underlying idea to allow side-effecting of disjoint portions of the heap. The technique is parametric on the actual shape analysis used to describe each weakly connected component in the heap graphs. In Section 5.5 we present a suitable shape domain, which allows us to prove race freedom for the motivating example, and for migration in general. However, this instance is rather close to concurrent shape analysis and has the well-known scalability issues. Therefore, in Section 5.6, we identify two methods – cheap heap domains and on-demand shape analysis – that make our approach more scalable for the programs we consider. The second method relies on the observation that code sections, where cheap race detection tools may fail to detect distinct locks guaranteeing mutual exclusion, e.g., due to migration, may be significantly smaller than the actual programs of interest. Only there, expensive shape domains need be applied.

Section 5.7 provides experimental evidence of this claim; the remaining sections describe related work and conclude.

5.2 Side-effecting with atomic sections

Before we discuss migration in the heap, we work through a much simpler but analogous instance of side-effecting with atomic sections where we only track the values of integer variables. Consider the program in Fig. 5.2. The main thread sets the global variable x to one and starts two threads. Globally, the shared variable x has value 1. This invariant, however, is locally violated by the first thread while it holds the lock. The other thread relies on the invariant to hold whenever it acquires the lock. Otherwise, it sets x to an error value. Verifying that this program is free from races requires the inference of the invariant, but the invariant can only be

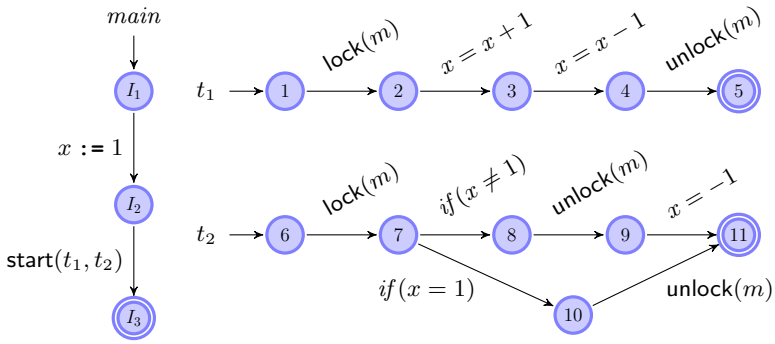

Figure 5.2: Invariant for global variable x is broken within atomic sections.

soundly deduced if the program can be shown to be free from races.

We now refine the global invariant approach from Section 2.4 to take atomic sections into account. Recall the original constraints:

$$\begin{aligned}
\Lambda(x) &\sqsupseteq (x \in [\![s]\!]_{acc}\, \langle \rho_u, \Psi \rangle)\ ?\ \lambda_u : \bot && \text{for } (u,s,v) \in E \text{ and } x \in V \\
\langle \lambda_v, \rho_v, \psi \rangle &\sqsupseteq [\![s]\!]^\sharp\, \langle \lambda_u, \rho_u, \psi \rangle && \text{for } (u,s,v) \in E \\
\langle \lambda_n, \rho_n, \psi \rangle &\sqsupseteq \langle \top, \top, \psi_0 \rangle && \text{for start nodes } n \in \vec{st}
\end{aligned}$$

where we have added a constraint for the start nodes, since we intend to here prove that this refined semantics is still sound w.r.t. the interleaving semantics of Section 2.1. In contrast to the interleaving semantics, however, we have the global lockset mapping Λ and the global invariant ψ, and for each program point, only a single lockset and mapping of the locals. Also recall that the lockset ordering is reversed, since we track must-sets of locks, hence $\top = \emptyset$ and $\bot = \mathcal{M}$.

The refinement required for atomic sections is allowing the value of globals to be temporarily stored within a localized version of the global state ψ during the atomic section. For this, we need an operator that determines the set of globals protected by some lockset λ according to the global lockset map Λ:

$$\mathsf{prot}_\Lambda(\lambda) = \{x \in G \mid \lambda \cap \Lambda(x) \neq \emptyset\}$$

The constraint system for atomic section also maintains for a program point u, the localized version of the global state ψ_u.

$$\begin{aligned}
\Lambda(x) &\sqsupseteq (x \in [\![s]\!]_{acc}\, \langle \rho_u, \psi_u \rangle)\ ?\ \lambda_u : \bot && \text{for } (u,s,v) \in E \text{ and } x \in V \\
\langle \lambda_v, \rho_v, \psi_v \rangle &\sqsupseteq [\![s]\!]^\sharp\, \langle \lambda_u, \rho_u, \psi_u \rangle && \text{for } (u,s,v) \in E \\
\langle \lambda_n, \rho_n, \psi_n \rangle &\sqsupseteq \langle \top, \top, \psi_0 \rangle && \text{for start nodes } n \in \vec{st} \\
\Psi(x) &\sqsupseteq (x \notin \mathsf{prot}_\Lambda(\lambda_u))\ ?\ \psi_u(x) : \bot && \text{for } u \in N, x \in G \\
\psi_u(x) &\sqsupseteq (x \notin \mathsf{prot}_\Lambda(\lambda_u))\ ?\ \Psi(x) : \bot && \text{for } u \in N, x \in G
\end{aligned}$$

u	$\psi_u(x)$	λ_u	$\Lambda(x)$	$\Psi(x)$	comment
1	1	$\emptyset$	M	1	Starting from ψ_0 and $\lambda(X) = \mathcal{M}$
2	1	$\{m\}$	M	1	lock is acquired
3	2	$\{m\}$	$\{m\}$	1	protected access to x
4	1	$\{m\}$	$\{m\}$	1	x is still protected
5	1	$\emptyset$	$\{m\}$	1	only now $\psi_5(x) \sqsubseteq \psi(x)$
6	1	$\emptyset$	$\{m\}$	1	Start from ψ_0, but also $\psi(x) \sqsubseteq \psi_6(x)$
7	1	$\{m\}$	$\{m\}$	1	lock is acquired
8	$\bot$	M	$\{m\}$	1	condition $x \neq 1$ is false
9	$\bot$	M	$\{m\}$	1	dead code: not a race!
10	1	$\{m\}$	$\{m\}$	1	condition $x = 1$ may be true
11	1	$\emptyset$	$\{m\}$	1	lock is released
1	1	$\emptyset$	$\{m\}$	1	re-compute first thread
2	1	$\{m\}$	$\{m\}$	1	$\lambda_2(x) \cap \lambda(x) \neq \emptyset$ still holds
3	2	$\{m\}$	$\{m\}$	1	Fixpoint!

Figure 5.3: Fix-point computation of the program in Fig. 5.2.

The two additional constraints synchronize the localized information with the global invariant ψ on the portion of the shared state that may not be protected at a given program point.

Note that for locksets $\lambda \sqsubseteq \lambda'$, we have $\lambda \supseteq \lambda'$, hence these constraints will be distinct from bottom for more variables in the case of λ' than for λ. As the same holds for Λ, the right-hand sides of these constraints are monotonic. Hence, we can solve the system by iterating from the least element as shown in the following example.

Example 18. The constraint solving would proceed as follows on the example from Fig. 5.2. We begin analyzing the main thread in single-threaded mode, which we could simulate as having all locks at all program points. The fix-point computation proceeds as shown in Fig. 5.3.

In order to relate the invariant-based semantics with the interleaving semantics, let $D[u] = \langle \lambda_u, \rho_u, \psi_u \rangle$ and let $S = (D, \Lambda, \Psi)$ be the least solution to the above constraint system. And let for all concurrent program points $\vec{u}$, the set $\mathbf{C}(\vec{u})$ denote all reachable configurations according to the interleaving semantics. For a given concurrent program location $\vec{u}$, we define the concretization $\gamma_{\vec{u}}(S)$. This set should include all the configuration reaching the concurrent point $\vec{u}$. A concurrent configuration $\langle \vec{\mu}, \vec{\sigma}, \varphi \rangle \in \gamma_{\vec{u}}(S)$ iff the following conditions hold. First, the locksets for these program points must be valid and over-approximate the locksets in $\vec{\mu}$,

i.e., $\forall i \in \mathcal{T} : \lambda_{u_i} \subseteq \mu_i$. Also, all local states must satisfy the concretization for maps defined above: $\forall i \in \mathcal{T} : \sigma_i \in \gamma(\rho_{u_i})$.

The global state must be represented by the global invariant together with the localized states. Each thread i has a localized version of the global variables for which it has acquired the associated mutex, hence the values for these variables should be taken from the localized mappings, while the unprotected variables are taken from Ψ. We define the auxiliary function $\psi_{\vec{u}}$:

$$\psi_{\vec{u}}(x) = \begin{cases} \psi_{u_i}(x) & \text{if } \exists i \in \mathcal{T} : x \in \mathsf{prot}_\Lambda(\lambda_{u_i}) \\ \Psi(x) & \text{otherwise} \end{cases}$$

We then require $\varphi \in \gamma(\psi_{\vec{u}})$. Note that this function is ambiguous if there exist more than a single i which satisfies the condition. This can realistically happen if the Λ sets are non-singleton. For now, let us assume these sets are singleton. We believe the following result hold in general because holding partial locksets would not allow a thread to update the invariant. However, the formal argument is not entirely trivial, hence we leave it as future work to generalize this result to non-singleton locksets.

Theorem 7 (Soundness)**.** *Let S and $\mathbf{C}$ be the invariant-based and the interleaving semantics, respectively, of a program. Then, for each concurrent program location $\vec{u}$, we have $\gamma_{\vec{u}}(S) \supseteq \mathbf{C}(\vec{u})$.*

Proof. Let $\mathbf{C}^*$ be the concretization of the invariant solution, i.e., $\mathbf{C}^*(\vec{u}) = \gamma_{\vec{u}}(S)$. We show that $\mathbf{C}^*$ satisfies the constraint system for the interleaving semantics. For this, we show that when a transition can be made from $\vec{u}$ to $\vec{v}$, using some edge (u_i, s, v_i), if a concurrent configuration $\langle \vec{\mu}, \vec{\sigma}, \varphi \rangle \in \mathbf{C}^*[\vec{u}]$, then $[\![s]\!]_i \langle \vec{\mu}, \vec{\sigma}, \varphi \rangle = \langle \vec{\mu}', \vec{\sigma}', \varphi' \rangle \in \mathbf{C}^*[\vec{v}]$.

By definition $\langle \vec{\mu}, \vec{\sigma}, \varphi \rangle \in \mathbf{C}^*[\vec{u}]$ implies $\langle \mu_i, \sigma_i, \varphi \rangle \in \gamma \langle \lambda_{u_i}, \rho_{u_i}, \psi_{\vec{u}} \rangle$. If only $\forall x \in G : \psi_{\vec{u}}(x) \subseteq \psi_{u_i}(x)$, we could conclude from the soundness of the transfer functions that $\langle \mu_i', \sigma_i', \varphi' \rangle \in \gamma([\![s]\!]^\sharp \langle \lambda_{u_i}, \rho_{u_i}, \psi_{u_i} \rangle)$ and since this constrains the triple $\langle \lambda_{v_i}, \rho_{v_i}, \varphi_{v_i} \rangle$, we would be done if $\forall x \in G : \varphi_{v_i}(x) \subseteq \varphi_{\vec{v}}$. Due to the localization by other threads, the heap at program point u_i may not contain values which are exclusively held by other threads. For any other global x, i.e., those protected by the current thread i and those not protected at all, we do have $\psi_{\vec{u}}(x) \subseteq \psi_{u_i}$.

The first constraint limits the lockset of all variables accessed by the statement s to the currently held set of locks λ_{u_i}. Since our lockset must be valid, λ_{u_i} does not overlap with any other lockset. If a thread were to access a variable protected by another thread, we would have disjoint locksets. Therefore, the current statement cannot access any variable protected by other threads, so the effect of the

statement does not depend on the values which are different. Thus, at least the lockset and local state are safe.

For the global state, we verify for each $x \in G$, whether $\varphi(x) \in \psi_{\vec{u}}(x)$ implies that $\varphi'(x) \in \psi_{\vec{v}}(x)$. We consider all cases:

1. $\exists j \neq i : x \in \mathsf{prot}_\Lambda(\lambda_{u_j})$. Since $u_j = v_j$, the lockset does not change, hence $\psi_{\vec{u}}$ and $\psi_{\vec{v}}$ both take their value from ψ_{u_j}. By the above reasoning $\varphi(x) = \varphi'(x)$, hence the implication holds.

2. $x \in \mathsf{prot}_\Lambda(\lambda_{u_i}) \cap \mathsf{prot}_\Lambda(\lambda_{v_i})$. Then, the $\psi_{\vec{u}}$ and $\psi_{\vec{v}}$ both take the value of x from the ψ_{u_i} and ψ_{v_i}, respectively. The implication holds based on the soundness of the transfer function.

3. $x \notin \mathsf{prot}_\Lambda(\lambda_{u_i}) \cup \mathsf{prot}_\Lambda(\lambda_{v_i})$. Then, the $\psi_{\vec{u}}$ and $\psi_{\vec{v}}$ both take the value of x from the global invariant Ψ. The last two constraints guarantee that this works correctly, as in the next two cases.

4. $x \in \mathsf{prot}_\Lambda(\lambda_{u_i})$, but $x \notin \mathsf{prot}_\Lambda(\lambda_{v_i})$. If a lock has been released, then we rely on the information being propagated from the local state to the global invariant to infer $\varphi_{v_i}(x) \subseteq \varphi_{\vec{v}}$.

5. $x \notin \mathsf{prot}_\Lambda(\lambda_{u_i})$, but $x \in \mathsf{prot}_\Lambda(\lambda_{v_i})$. When a lock is acquired we rely on the last constraint to infer $\psi_{\vec{u}}(x) \subseteq \psi_{u_i}(x)$.

This concludes the proof. □

Corollary 2 (Data Race freedom)**.** *Let $S = (g, \vec{d}, \Lambda)$ be the invariant-based semantics of a program. For a resource $r \in R$, if $\Lambda(r) \neq \emptyset$, then there is no data race on r at any configuration.*

Proof. For there to be a race at variable x for concurrent program point $\vec{u}$, there would need to be an admissible lockset $\vec{\mu}$ at $\vec{u}$ and two different threads i and j having the transfer functions from points u_i and u_j which access the same resource. From theorem 7 $\lambda_{u_i} \subseteq \mu_i$ and $\lambda_{u_j} \subseteq \mu_j$. For the lockset at $\vec{u}$ to be admissible μ_i and μ_j cannot overlap, hence also λ_{u_i}) and λ_{u_j} have no locks in common. Assuming the correctness of acc, these both constrain the lockset $\Lambda(x)$ resulting in $\Lambda(x) = \emptyset$. □

5.3 Privatization of heap regions

In this section, we adapt the underlying method to shape domains in order to deal with migration. In this case, the domain tracks local variables as part of the shape graph. For the concrete semantics, we assume threads execute their dedicated

code, hence all local variables are distinct and the heap and local variables of all threads can be represented by a huge shape graph. In the invariant-based semantics, we join portions of the localized heap with the global invariant. For these constraints to be monotonic, the domain of shape graphs must be closed with respect to Cartesian products of disjoint sub-graphs. Thus, we cannot express in our abstract domain that two linked lists are either both empty or both non-empty. Such properties may be critical for proving memory safety, but it is hard to imagine such correlations between different data structures playing any significant role in race detection.

Formally, let h be a shape graph. As for the region analysis, we are interested in disjoint portions of the heap associated with global owners. For a subgraph $h' \subseteq h$, we let $\mathsf{globs}(h')$ denote the set of global variables in h'. For two disjoint graphs h_1 and h_2 with $\mathsf{globs}(h_1) \cap \mathsf{globs}(h_2) = \emptyset$, we write $h_1 \star h_2$ for the juxtaposition of the heap graphs, i.e., the (disjoint) union of the graph nodes and edges. For a graph h, let $\mathsf{wcc}(h) = \{h_1, \cdots, h_p\}$ denote its weakly connected components, and similarly for a set of graphs g, we define $\mathsf{wcc}(g) = \bigcup\{\mathsf{wcc}(h) \mid h \in g\}$. Then, the closure of a set of graphs g is defined as all compatible and complete juxtapositions of the weakly connected components of g:

$$\mathsf{cl}(g) = \{h_1 \star \cdots \star h_p \mid h_i \in \mathsf{wcc}(g),\ \mathsf{partition}\{h_1, \ldots, h_p\}\}$$

where the predicate $\mathsf{partition}(g)$ holds for the set of (sub-)graphs g, if there exists an equivalence relation π on G such that the owners partition the globals according to π, i.e., $\{\mathsf{globs}(h) \mid h \in g\} = G/\pi$.

Our abstract domain $\mathbb{D}_L$ is the set of all closed sets of graphs ordered by subset inclusion. The least upper bound is the closure of the union. The juxtaposition operator $\star$ can be lifted to sets of heap graphs with disjoint owners. For $\mathsf{globs}(g) = \bigcup\{\mathsf{globs}(h) \mid h \in g\}$ and $\mathsf{globs}(g_1) \cap \mathsf{globs}(g_2) = \emptyset$, we can compute $g_1 \star g_2 = \{h_1 \star h_2 \mid h_1 \in g_1, h_2 \in g_2\}$. The operation is only defined for sets where all pairwise juxtapositions are possible. Furthermore, require that our shape domain supports the following operations.

- $\mathsf{restrict}\colon 2^G \times \mathbb{D}_L \to \mathbb{D}_L$. The restriction of a graph to the portion of the graph weakly connected to the variables in r. The local variables pointing outside the preserved region is set to a poison value.

- $\mathsf{extract}\colon \mathbb{D}_L \to \mathbb{D}_G$. The elimination of local pointer variables from the shape graph. The domain $\mathbb{D}_G \subseteq \mathbb{D}_L$ is the set of shape graphs without local pointers. We use this for the global invariant.

Making the implicit assumption that we always apply the closure operation to each right-hand side, the constraint system for the invariant-based semantics can

be defined similar to previous system:

$$\begin{aligned}
\Lambda(x) &\sqsupseteq (x \in [\![s]\!]_{acc}\, \langle \psi_u \rangle)\ ?\ \lambda_u : \bot && \text{for } (u,s,v) \in E \text{ and } x \in V \\
\langle \lambda_v, \psi_v \rangle &\sqsupseteq [\![s]\!]^\sharp\, \langle \lambda_u, \psi_u \rangle && \text{for } (u,s,v) \in E \\
\langle \lambda_n, \psi_n \rangle &\sqsupseteq \langle \top, \psi_0 \rangle && \text{for start nodes } n \in \vec{st} \\
\Psi &\sqsupseteq \mathsf{restrict}(r, \Psi) \star \mathsf{extract}_{\bar{r}}(\psi_u) && \text{for } u \in N,\ r = \mathsf{prot}_{\Lambda_\Psi}(\lambda_u) \\
\psi_u &\sqsupseteq \mathsf{restrict}(r, \psi_u) \star \mathsf{restrict}(\bar{r}, \Psi) && \text{for } u \in N,\ r = \mathsf{prot}_{\Lambda_\Psi}(\lambda_u)
\end{aligned}$$

where we use the abbreviation $\mathsf{extract}_r(\psi) = \mathsf{restrict}(r, \mathsf{extract}(\psi))$. The lockset mapping Λ tracks accesses for individual region owners. When a region collapses it will associate an access with all associated owners; however, the current locksets based on previous accesses should be intersected immediately.

The right-hand sides are monotonic because the unprotected portion of the heap increases when we lose information about the locksets. Assume locksets $\lambda \sqsubseteq \lambda'$, and as ordering is inversed λ' protects either the same or a smaller portion of the heap. If we need to propagate a larger portion of the heap for λ', we know it must be disjoint, e.g., $\psi_{prev} \star \psi_{now}$, and since we keep only closed sets of shapes, ψ_{prev} will be juxtaposed with elements in Ψ producing all the heaps that would be produced when only ψ_{prev} is propagated.

For the soundness proof, the critical observation is that the lockset map ensures that regions are checked in and out in such a way the juxtaposition will always succeed. Then, the same concretization strategy can be applied as in the previous chapter. To more clearly see the parallel, one could regard the closed sets of heap graphs as consisting of the generator sets of the weakly connected components which form the closure. Then, the constraints for propagating unprotected sections of the heap can be expressed more similarly to the integer constraints:

$$\begin{aligned}
\Psi &\supseteq (x \notin \mathsf{prot}_\Lambda(\lambda_u))\ ?\ \mathsf{region}(\psi_u, x) : \emptyset && \text{for } u \in N, x \in G \\
\psi_u &\supseteq (x \notin \mathsf{prot}_\Lambda(\lambda_u))\ ?\ \mathsf{region}(\Psi, x) : \emptyset && \text{for } u \in N, x \in G
\end{aligned}$$

where $\mathsf{region}(\psi, x)$ returns all the weakly connected components containing the global x. The formalization of these intuitions remains as future work.

5.4 Brief introduction to TVLA

We express a proof-of-concept implementation of these ideas in the Three-valued Logic Based Shape Analysis (TVLA) formalism [75]. This setting is particularly suitable to our work, since TVLA naturally supports reasoning about reachability and disjointness, which will be the basis of our $\star$-separation. TVLA builds on the notion of logical structures over a certain signature $\mathcal{P}$. A *logical structure*

$S = (U, \iota)$ *over* $\mathcal{P}$ is a pair of a set of individuals U ranged over by u and an interpretation, ι. Each predicate symbol $p/k \in \mathcal{P}$ of arity k is mapped by ι to a boolean-valued function $\iota(p/k) : U^k \to \mathbb{B}$. The set of all structures over a $\mathcal{P}$ is written $\mathcal{S}[\mathcal{P}]$. We evaluate formulas of first-order logic with transitive closure, FO(TC), on logical structures. Formulas are defined by:

$$\varphi = 0 \mid p(v_1, \ldots, v_k) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists v : \varphi \mid TC(v_1, v_2 : \varphi)(v_1', v_2')$$

where $v \in \mathsf{Var}$ is a logical variable.

The transitive closure operator, $TC(v_1, v_2 : \varphi)$ defines a binary relation by taking a formula φ using free variables v_1 and v_2. The transitive closure of this relation may then be applied to v_1' and v_2'. The evaluation of a formula φ in structure S and assignment Z (of free variables to individuals) is written $[\![\varphi]\!]^S(Z)$.

Logical structures are used to encode heap graphs. Traditionally, an individual corresponds to a record and a binary predicate s holds of individuals u and u', if there is a pointer-valued component s of the record modeled by u which points to (the head of) the record modeled by u'. The predicates used to encode a heap are essentially the binary selectors, $\mathbf{Sel}$, and the unary program variables, $\mathbf{Id}$, that hold of records pointed to by the corresponding variables. These predicates are called *core predicates*, the set of which is denoted by $\mathcal{C}$.

The semantics of an assignment *st* is a mapping $[\![st]\!] : \mathcal{S}[\mathcal{C}] \to \mathcal{S}[\mathcal{C}]$. This is given in terms of *predicate update formulas* that update the value of predicates affected by the statement. Given for each k-ary predicate $p \in \mathcal{C}$ an update formula φ_p^{st} with free variables v_1, …, v_k, the semantics of st is defined as $[\![st]\!](S) = (U', \lambda p.\lambda u_1, \ldots, u_k.[\![\varphi_p^{st}]\!]^S(Z))$ where $Z = [v_1 \mapsto u_1, \ldots, v_k \mapsto u_k]$ and the universe U' is either the same as before or (in the case of memory allocation) extended with fresh individuals.

Abstract states in the TVLA framework are *three-valued logical structures* based on Kleene's three-valued logic. Abstract states are obtained by *canonical abstraction*, an abstraction that summarizes individuals that are indistinguishable under a set of abstraction predicates to *summary nodes*. Due to summarization, knowledge about certain predicates may become indefinite, in which case the logical value 1/2 is introduced. To avoid serious loss of precision, *instrumentation predicates* are employed; these are additional predicates defined through formulas of FO(TC) using the core predicates $\mathcal{C}$. Instrumentation predicates allow to better distinguish abstract nodes by annotating logical structures with additional information such as reachability, sharing, or cyclicity. Update formulas for instrumentation predicates can be automatically inferred using differencing [70].

As abstract states are still logical structures, the concrete semantics in terms of predicate update formulas is easily lifted to three-valued logical structures, too. As a consequence, it is sufficient to specify predicate update formulas and a set of

instrumentation predicates to define a program analysis in the TVLA framework. Soundness then is immediate.

5.5 Dealing with migration

To deal with our running example we focus on cyclic doubly-linked lists, which are ubiquitous in the Linux kernel and device drivers. Therefore, we present the instance of $\mathbb{D}_L$ able to deal with these lists. $\mathbb{D}_L$ is the powerset of the set of *shape graphs*, $\mathbb{D}_L = 2^{3-\mathsf{STRUCT}(\mathcal{P})}$, where each graph S is a three-valued logical structure (U, ι) over predicates $\mathcal{P}$. Here, U is the set of individuals (each representing a kernel list struct), ι maps predicates to their *three-valued interpretation*.

To deal with kernel lists we provide a new, concrete instance of the TVLA framework. Despite being novel, it employs mostly standard predicates found in almost all TVLA instances. Concretely, we employ predicates $\mathcal{P}$:

- unary predicates $\mathsf{X} = \{z/1 \mid z \in G \cup L\}$ that hold for memory cells pointed to by a local or global variable; and
- binary predicates $\mathsf{Sel} = \{next/2, prev/2\}$ to encode dereferenced forward and backward pointers.

The sets X and Sel make up the set of core predicates, which are needed to encode heaps consisting of kernel lists. In addition, we employ the following instrumentation predicates needed to maintain more precise information in abstract structures. Instrumentation predicates are defined in terms of core predicates using first-order logic with *transitive closure*:

- Unary predicates $\mathsf{R} = \{r_{s,z}/1 \mid s \in \mathsf{Sel}, z \in \mathsf{X}\}$ to hold for heap cells reachable from a variable z following s selectors only. They are defined as $r_{s,z}(v) = \exists v'.z(v') \wedge s^*(v', v)$ where $*$ denotes reflexive, transitive closure.
- Unary predicates $\mathsf{C} = \{c_{next}/1, c_{prev}/1\}$ to hold for heap cells lying on a $next$ ($prev$) cycle, defined as $c_s(v) = \exists v'.s(v, v') \wedge s^*(v', v)$. This is a standard instrumentation in the treatment of possibly cyclic singly-linked lists.
- Unary predicates $\mathsf{C}' = \{c_{s_1,s_2}/1 \mid s_1 \neq s_2 \in \mathsf{Sel}\}$ to require that an element v has an s_1 neighbor v' with an s_2 edge back to v: $c_{s_1,s_2}(v) = \exists v'.s_1(v, v') \wedge s_2(v', v)$. This deviates from a similar predicate used in TVLA analyses of doubly-linked lists, which is defined by $\forall v'.s_1(v, v') \rightarrow s_2(v', v)$. Here we need a stronger property, which, due to cyclicity holds for the last element of a kernel list as well.

Finally, there are a few more auxiliary instrumentation predicates to capture sharing, binary transitive reachability and summary information (a summary element in an abstract shape represents one or more than one concrete elements). Figure 5.4 shows the two-element set $d^A_{\mathsf{ker}} = \{S^0_A, S^1_A\}$ of shape graphs obtained from running the *generate* thread of our running example in isolation. These shape graphs represent an empty and a non-empty kernel lists, respectively, pointed to by A. The instrumentation predicates that hold for a node are written inside a node. Due to these predicates, we can infer that we are actually facing a proper, cyclic, doubly-linked kernel list.

Defining the operations required for the side-effecting framework of the previous section is straightforward provided the reachability information is correctly computed. Extracting global information $g \in \mathbb{D}_G$ from local information $d \in \mathbb{D}_L$ is described in terms of a single shape graph $h \in d$ and lifted elementwise to d. Let h be (U, ι), then $\mathsf{extract}(h)$ is obtained by setting $\iota(p)$ to $\lambda x.0$ for each $p \in \mathcal{P}_L$. This is effectively the composed effect of the transfer functions nullifying all local variables. See Figure 5.4 for an example of extraction. Furthermore, restricting a shape graph to only those elements reachable from a set of globals is straightforward computation on the graphs.

A major benefit of the TVLA framework is the uniform treatment of concrete and abstract transformers. This is possible because both concrete and abstract states are logical structures over the same set of predicates. The semantics of a statement is a set of predicate transformers. Also, in most cases it suffices to provide update formulas for core predicates only, while the update formulas for instrumentation predicates can be derived automatically [70]. The predicate update formulas needed for our instance are mostly standard, hence we omit their details here; in the next chapter, where we consider shape analysis for system code, we provide update formulas.

We now briefly describe how the shape instance of our framework handles the running example and the list migration problem. We assume that the initialization phase simply initializes A and B to empty kernel lists, that is, $g_0 = \{S^0_A, S^0_B\}$, where S^0_A is given in Figure 5.4(a) and S^0_B is analogous for B. Moreover, we have $G = \{A, B\}$, $\mathcal{M} = \{l_A, l_B\}$, and $L = \{p, q, r\}$.

Running the generator thread accesses only A with lock l_A (called `A_lock` in the program). So due to the first constraint when A is accessed at line 4, we constrain $\Lambda(A)$ to be contained in $\{l_A\}$, the lock set held at line 4. At that line, our shape analysis runs completely sequentially, adding one element at a time to A. The last two constraints simply update information on the disjoint B at this point. We will obtain $d^A_{\mathsf{ker}} \star \{S^0_B\}$ after iterating over `generate`.

Let us now iterate over the *processor* thread. The critical part is program line 11. Here we introduce sharing between A and B, however, both resources are

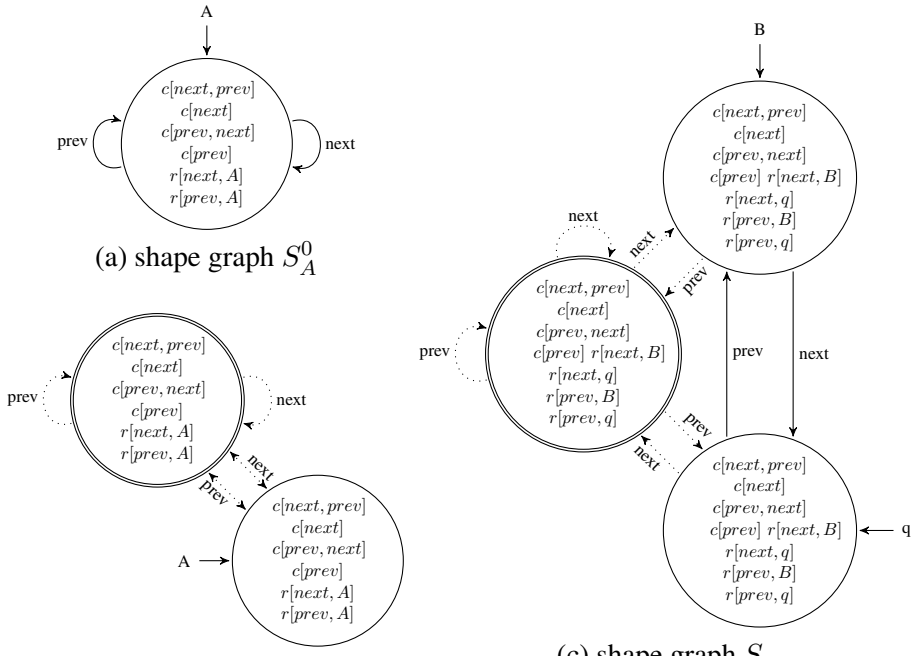

Figure 5.4: The element $d^A_{\mathsf{ker}} = \{S^0_A, S^1_A\} \in \mathbb{D}_L$ consisting of shape graphs representing empty and non-empty kernel lists pointed to by A, respectively. Nodes with double lines represent summary nodes. Shape graph (c) is obtained after `list_move` at program line 11 of the running example.

protected since we hold both locks, $\lambda_{11} = \{l_A, l_B\}$. Note that conservative race detection tools are now lost and will not be able to re-establish separation between A and B. Using shape analysis, however, we obtain $d_{12} = \{S_q\} \star d^A_{\mathsf{ker}}$ where S_q is given in Figure 5.4(c). It shows that q has been correctly inserted immediately after B, as well as that A and B are disjoint. Only now do we release the lock on B, hence we need to compute $\mathsf{extract}_{\{B\}}(d_{12})$. We obtain S^1_B, which is exactly S_q without local variable q.

At this point $\Lambda(B) = \{l_A, l_B\}$, because B has only been accessed under protection of both locks. It will be constrained to $\{l_B\}$ at line 17, when B is accessed under protection of l_B only. This is essentially all we need to do, and the final global result will be $g = \{d^A_{\mathsf{ker}}, d^B_{\mathsf{ker}}\}$, $\Lambda(A) = \{l_A\}$, and $\Lambda(B) = \{l_B\}$. According to Corollary 2, this proves race-freedom for this migration example.

5.6 Going practical

The shape instance of the previous section is fairly general. If we decide on a more precise shape domain, for example, one that can reason about trees or any other data structure of interest, we obtain a thread-modular shape analysis; however, the simultaneous inference of lock sets, requires for the sake of monotonicity that we lose precision. Nevertheless, the *cost* of this analysis is still comparable to other thread-modular shape analyses [34].

Here, we have a particular interest in proving absence of races for device drivers. In this section, we exploit two key observations for that application domain that facilitate scalability in practice. Numbers supporting this claim are presented in Section 5.7.

1. Kernel lists are the ubiquitous data structure in device drivers. The only information that we really need is whether at non-protected sections of code, these are well-formed. This allows for a very *cheap heap domain* $\mathcal{G}$.

2. While migration is a common phenomenon, it happens only on small portions of the analyzed code. Without these portions efficient race detection techniques used in Goblint could prove race-freedom. We propose *on-demand* shape analysis that employs expensive shape domains only on these exact portions and cheap domains otherwise.

Cheap heap domains. In our tentative implementation described in Section 5.7, we use $\mathbb{D}_L$ as provided in Section 5.5. However, as invariant capturing domain, we use $\mathbb{D}_G = 2^\Pi$. $\Pi = 2^G \times \mathbb{B}$ is a partition of the globals, like the one used in the region analysis, except each element has an extra bit. This extra bit is simply information about whether or not we are dealing with a well-formed kernel

list. For example, $\{(\{A, B\}, 1), (\{C\}, 0)\}$ represents a heap, where A and B are aliases and point to a well-formed kernel list. C points to an unspecified (but separate) portion of the heap. It is effectively an additional layer of abstraction.

However, it is practically important that as soon as we set the extra bit to 0, no more shape analyses will be carried out for this portion of heap. Most likely this corresponds to race condition, because we have an unstable, non-well-formed, non-protected shared state. So we conservatively give up at this point potentially saving a lot of extra shape analysis computations.

This cheap domain approach can be taken even further and generalized to other data structures. Instead of $\mathbb{B}$ we take a flat lattice, where 0 is replaced with $\top$ and 1 with a set of incomparable elements such as tree, list-of-list, or other well-formed structures.

This lattice could even be inferred automatically based on initialization code which build up internal data structures. One could then perform a standard sequential shape analysis of the initialization and take the resulting shapes as *target graphs* to be put into the flat lattice. The implementation and evaluation of the latter method is ongoing work.

On-demand shape analysis. The region analysis of Chapter 4, can be used as an initial analysis. It infers a preliminary analysis result $\Lambda : G \to \mathcal{M}$ based upon which we can define

$$G_0 = \bigcup\{x \in G \mid \Lambda(x) = \emptyset\} \qquad G_1 = \bigcup\{x \in G \mid \Lambda(x) \neq \emptyset\}$$

Then we know that all accesses to heap elements reachable from globals G_1 are protected by some lock, while accesses through globals from G_0 are endangered to be non-exclusive. We can use this information to start the more refined analysis using $\mathbb{D}_L$.

In this case, it can be accelerated by starting the fixpoint iteration with

$$g = g_1 \star \{(x, 0) \mid \Lambda(x) = \emptyset\}$$

where g_1 with $\mathsf{globs}(g_1) = G_1$ is the G_1 reachable part of the heap obtained from initialization.

This analysis will be as imprecise on the heap cells reachable from globals G_1 as the region analysis and, hopefully, more precise on the heap cells reachable from G_0. In the next section, we give experimental evidence, that, in practice, G_0 tends to be much smaller than G_1, effectively leading to a much faster analysis.

5.7 Empirical evaluation

Our implementation builds upon the region analysis of Goblint combined with the precise shape domain $\mathbb{D}_L$ of Section 5.5 implemented in the TVLA analyzer [75].

Name	LoC	Slice	warnings before	warnings after
char-ipmi	3113	440	16	8
ieee1394-raw	3938	294	7	5
infiniband-ucma	980	78	5	3
sound-core-timer	1689	318	9	5
ssb-main	1635	82	8	4
usb-gadget-serial	776	106	10	8
infiniband-ulp	1754	137	15	3
move	79	26	2	0
direct	76	23	2	0

Table 5.1: Experimental results.

We are currently implementing a shape domain tailored for cyclic doubly linked lists in Goblint, but for our preliminary experiments we needed to interface semi-automatically. This is why we do not state any running times. We can, however, say that the combined running time of the tools was well under a minute for all but the first of the examples of Table 5.1. There it was a few minutes. These small running times are caused by rather small and easy shape analysis problems (due to the on-demand approach) as well as by the fact that we give up early within the cheap heap domain.

We followed the on-demand approach outlined in Section 5.6; that is, running region analysis first and then extracting a slice of the program: a set of global variables where region analysis produced warnings. The table indicates that the section of code, where we needed a precise shape analysis amounts to 10 percent on average.

The warnings columns indicate how many warnings could be dismissed due to precise handling of migration. The larger values are the number of warnings produced by region analysis, the smaller number results from using on-demand shape analysis. On average, the number of warnings could be reduced by 50 percent. In particular, we could prove race-freedom for the running example, move.

The best results could be obtained for the InfiniBand SCSI RDMA Protocol initiator module, called infiniband-ulp in the table. It features a constant number of request structures. Requests may reside in a work queue or in a free-list. New requests are taken from the free-list, populated with data and added to the work queue. Requests move from the work queue to the free-list if they have been processed and their substructures have been freed. These two lists use different protection. The corresponding locking scheme perfectly matches the algorithmic

patterns whose correctness can be verified by our method.

The remaining warnings are either actual races or due to intricate features like overlapping structures, where data is organized in a number of structures at the same time. We shall present a shape domain capable of reasoning about overlap in the next chapter; however, it is not yet implemented within this framework.

5.8 Related work

Static analysis of heap-manipulating programs, sequential or concurrent, is a very active field of research. In this discussion, we focus on a few representative analyses based on either separation logic [71] or three-valued logic based shape analysis [75].

In sequential settings, such analyses can be remarkably scalable [11, 93]. The situation is obviously different in a concurrent setting. The un-optimized shape instance of our framework presented in Section 5.5 resembles thread-modular shape analysis by Gotsman et. al [34]. They designed concurrent shape analysis based on resource invariants, where portions of the heap protected by a lock are separated. However, they rely on a-priori race-detection and lock set computation to enable sequential shape analysis of concurrent programs, whereas we use on-demand shape-analysis to enable race detection of heap-manipulating programs.

More recent concurrent shape analyses algorithms have been introduced, e.g. [3, 12, 55, 86], but they focus mainly on proving properties such as memory safety, data structure invariants, and linearizability for small but complex concurrent data structure implementations. Data-structure wise, our problem is much simpler, since the ubiquitous data-structure in drivers and kernel code are cyclic, doubly-linked lists.

In general, one can observe that different notions of heap decomposition and separation [54, 71, 73] are key to achieve thread- and heap-modularity for shape analyses. Our work is no different, except we require a stronger notion of separation, more similar to that used in type-based approaches to enforce isolation [82].

While such shape analysis techniques are potentially capable of dealing with race-free migration, they have difficulties in scaling to thousands of lines of code. We have proven that the heavy machinery of full-fledged concurrent shape analysis is not required to verify race-freedom of many list manipulating programs, even including migration. Indeed, we view our work more in line with cheap, scalable static race detection methods, such as [47, 61, 68, 89]. None of these methods, however, is able to deal with migration in a sufficiently precise way.

5.9 Conclusion

We identified the problem of migration to be a common source of imprecision for scalable race detection. Instead of resorting to expensive concurrent shape analyses methods, we solved the problem by integrating as little shape reasoning as possible into existing and scalable methods in a demand-driven fashion. To that end we provided a general framework of side-effecting with atomic sections, which is parametric in the notions of resources and resource separation. A unique feature of the framework is the *simultaneous* inference of global invariants, which may be temporarily violated, and the set of locks that protect them. Suitably instantiated the framework allowed to prove race-freedom for a prototypical migration example, list move, which is frequently found in device drivers and kernel code. In addition, we illustrated two methods, cheap heap domains and on-demand shape analysis, that make the approach scale to thousands of LoC. Experimental evidence supports our key observation: shape analysis can be used parsimoniously to discover data-races in realistic code at high precision and little cost. Clearly, in the future, we need a tighter tool integration or even a new tool, to fully put our technique to work. Also we are working on automatic inference of cheap heap domains by careful (shape) analysis of initialization code to maximize automation and applicability.

CHAPTER 6

SHAPE ANALYSIS OF LOW-LEVEL C

In the previous chapter, we relied on a shape analysis to detect whether a node has been completely removed from a data structure. Unfortunately, low-level C programs require more fine-grained view of the heap than used in off-the-shelf shape analyzers. Device drivers often keep data in multiple data structures simultaneously while embedding list or tree related records into the records containing the actual data; this results in *overlapping* structures. Shape analyses have traditionally relied on a graph-based representation of memory where a node corresponds to a whole record and edges to pointers. As this is ill-suited for encoding overlapping structures, we propose and formally relate two refined memory models. We demonstrate the appropriateness of these models by implementing shape analyses based on them within the *Three-valued Logic Based Shape Analysis (TVLA)* framework [75]. The implementation is exemplified using code extracted from cache managing kernel modules.

6.1 Introduction

Shape analysis of heap-manipulating programs is a very active field of research; however, the focus of most work has been devoted to Java-like data-structures, where pointers are not as heavily manipulated and computed with as in low-level C. While shape analyses addressing pointer arithmetic in a broad sense have recently been designed, e.g., [5, 10, 14, 15, 25, 35, 38], we address a related and particularly difficult problem: *overlapping structures*. The term was coined in [5], where the shape analysis of such structures was stated as an open problem.

Overlap is often found in device drivers where data is kept in several data-structures at the same time by means of embedding list or tree related records into the records containing the actual data. An example of such code is shown in Figure 6.1, where a `node` record (we shall consequently use the more general

```
struct hlist_head { struct hlist_node *first; };
struct hlist_node { struct hlist_node *next, **pprev; };
struct  list_head { struct  list_head *next, *prev; };

struct node { int data; struct hlist_node list;
                        struct  list_head queue; };

struct hlist_head ht[512]; mutex hlock;
struct  list_head cq;

void cleanup_task(void *arg) {
    struct hlist_head garbage; struct node *pos;
    lock(&hlock);
    list_for_each(pos, &cq) {
        hlist_del(&pos→list);
        list_del(&pos→queue);
        hlist_add(&pos→list, &garbage); }
    unlock(&hlock);
    hlist_for_each(pos, &garbage, list) {
        access(pos→data);
        hlist_del(&pos→list); } }
```

Figure 6.1: Overlapping data-structures from the Linux kernel.

term record to denote C structs) contains data as well as two list-related components. The first, `hlist_node`, is the record type which embeds the forward and backward pointers of an hlist (see below) into a node; the second, `list_head`, is a record type which serves both as the list head and as the record that embeds standard cyclic doubly-linked lists into nodes.

Hlists (or pprev lists) are in themselves quite tricky data-structures. In order to save memory while maintaining efficient implementation of insertion and deletion, Linux developers use these doubly linked lists with a pointer to the `next` component of the previous element rather than to the element itself. (This is visualized in a memory snapshot in Figure 6.2, where the edges from pprev boxes end at the smallest boxes rather than at the medium-sized ones as is the case for edges originating in prev boxes.) Hlists are used in hash-tables where having only a single pointer in the list head can be a significant gain.

The code given in Figure 6.1 shall serve as a basis for our case study. We use a syntax close to the original code, but abbreviate function names and eliminate some of the parameters to the list-traversal macros. These macros expand into for-

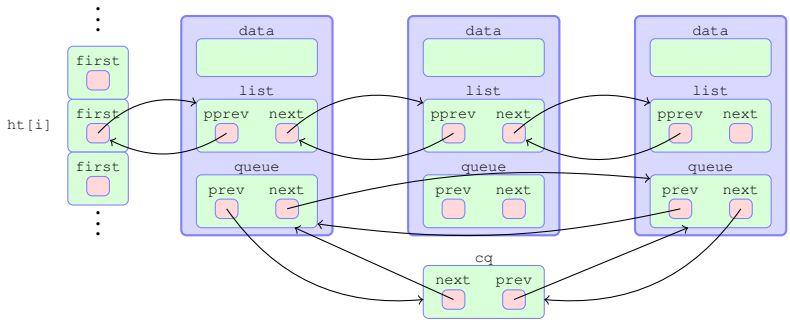

Figure 6.2: Overlapping Structures.

loops and use pointer-arithmetic to move from a record embedded within a node to the containing record. The example is based on code for maintaining a cache where the least recently used items are tagged and added to the cleanup queue. This queue is processed asynchronously by a cleanup task whose code is given in the figure. As this task may be executing concurrently with code that accesses the cache, elements in the queue are moved to the thread-local list `garbage` for statistical processing before being deallocated. This minimizes the time that the cleanup task must keep the lock on the cache.

When an object is removed from *all* thread-shared data-structures, the subsequent post-processing of the *privatized* object no longer requires protection through the acquisition of locks. However, if an element resides in two lists simultaneously, traversing these distinct lists may cause a race when accessing the data of the shared element. To prove absence of races in the example, we must infer that an element is in the queue but no longer in the list, although both queue- and list-related records are embedded into the same `node`.

Shape analyses often rely on graph-based representations of memory where a node corresponds to a whole record and edges to pointers. For these, it is not immediate how to encode pointers between components. Therefore, we propose two refined memory models which exhibit the low-level details required to reason about overlapping structures. Both memory models are formulated in terms of the TVLA framework. Using the TVLA framework is not necessary but allows for a quick prototype implementation. Since our refinement is conservative, we can also benefit from knowledge and developments in the TVLA realm.

The model of Section 6.2 uses a one-node-per-component approach, i.e., each box of Figure 6.2 becomes a single node in the shape graph, and the hierarchy induced by the box nesting is translated into a tree structure. We design a program analysis based on this model and demonstrate its potential on a list element

deletion procedure that uses unorthodox pointer manipulations.

Alternatively in Section 6.3, we propose a coarser and possibly more efficient model that employs a one-node-per-outermost-record paradigm. This approach annotates edge dereferences with access paths into the finer structure of the records, making sources and targets of dereferencing explicit. We exemplify shape analyses based on this coarser semantics by verifying deletion from an hlist. In Section 6.4, we characterize the relationship between the two models.

In Section 6.5, we revisit our motivating example of Figure 6.1. The analysis is conducted w.r.t. the coarse-grained semantics and enables us to verify race detection properties for it. Sections 6.6 and 6.7 present related work and conclude.

6.2 Fine-grained semantics

We begin by introducing the syntax of the C subset under consideration. Our aim is to cover the part of the C language crucial to most low-level programs like kernel code and drivers. We support arbitrarily nested named records and pointers to named types only; integers are not considered. This implies that we have both records and pointers as values. We use the domain $\mathbf{Id}$ of *variables* ranged over by x, y, and z, and the domain $\mathbf{Sel}$ of component *selectors* ranged over by s. Type names are ranged over by t. We consider the following languages of types τ and pointer expressions e:

$$\begin{array}{lcl} \tau & ::= & \mathbf{struct}\ t\ \{\tau_1\ s_1, \ldots, \tau_k\ s_k\} \mid \tau * \\ e & ::= & \mathsf{null} \mid x \mid *e \mid \&e \mid e \to s \mid \mathbf{malloc}(t) \mid \mathbf{up}(x, t, s) \end{array}$$

We omit arbitrary pointer arithmetic, unions, and type-casts; rather, we restrict pointer manipulation to component selection and the expression $\mathbf{up}(x, t, s)$ used as a primitive to model the `container_of` macro which computes the address of a record of type t from a pointer x to its s component. Since recent versions of the Linux kernel rely on *built-in* support by the compiler to implement this macro, a primitive treatment of the container-of idiom is sensible.

In the presence of overlap the $\mathbf{up}()$-operator is crucial to *change views*. For instance, in the example of Figure 6.1 one could traverse the cleanup queue, use the operator $\mathbf{up}()$ to jump to the data component of a node, and then continue traversing the `list` components.

As for statements we only consider assignments between pointer expressions. We assume that programs are compiled into a control-flow graph where assignments are attached to edges and where pointer comparisons may serve as guards. We now present our first refinement of the one-node-per-record paradigm by adopting a *one-node-per-component* representation. More precisely, a record of type $\mathbf{struct}\ t\ \{\tau_1\ *\ s_1, \ldots, \tau_k\ *\ s_k\}$ is represented by $k + 1$ nodes, one being

the head and one for each pointer component. If the component types are records again, additional nodes for the subcomponents are introduced, until finally pointer types are reached. This corresponds to transforming the hierarchy of boxes in Figure 6.2 into a tree.

This memory model is both more explicit and more abstract than that of real C. While in C the address of a record and the address of its first component coincide, they are considered as different here. On the other hand, we do not model the order of components or padding between each two of them. In our model, the operation $\mathbf{up}()$ amounts to moving from a component node to the head of its enclosing record.

Since we rely on the TVLA framework, as outlined in Section 5.4, we aim at encoding a state as a logical structure. Here we use the signature

$$\mathcal{C} = \{x/1 \mid x \in \mathbf{Id}\} \cup \{s/2 \mid s \in \mathbf{Sel}\} \cup \{*/2\}$$

In order to reason about expressions of the form $\&x$, the corresponding predicate x holds of an individual representing the stack location where x's value is stored. In standard TVLA the predicate x holds of the element *pointed to* by x.

The key predicate in our formulation is the binary $*$ for dereferencing. In particular, it holds between the location of a pointer variable and the value the variable points to. An interesting feature of our model is that the only predicate that is affected by assignments is the $*$ predicate: once allocated, the nodes representing a record and its components do not change and neither does the location of a variable.

We now formalize the semantics of our programming language. Each basic statement gives rise to an *update formula*. The update formulas, except for allocation, are given in Table 6.1. As is often done, we consider only one pointer operation per assignment, and we assume that pointers which are assigned to are always explicitly nullified before-hand, so that updates for assignments only introduce a single new points-to relationship. For example, in the case of $x = \mathbf{up}(y, t, s)$, we have that $*(u_1, u_2)$ holds after the assignment if it either held before, or if u_1 is the individual hosting x and u_2 has an s-component which is pointed to by the individual hosting y.

As for memory allocation, we expand the universe by a set of new individuals depending on the type of the record to be allocated. Recall that we require one individual per (sub)component of each non-pointer type. To this end, we introduce the notion of an *access path*. Such paths are not to be confused with access paths found in storeless semantics [24, 43]. Here, they merely reflect the static structure of a (possibly nested) record. Intuitively, there is a path for each component of a record. Formally, we define the set Π of access paths to be a union over all record

st	$\varphi^{st}_*(v_1, v_2) =$
$x = \mathsf{null}$	$*(v_1, v_2) \wedge \neg x(v_1)$
$*x = \mathsf{null}$	$*(v_1, v_2) \wedge \neg\exists\, v' :\ x(v') \wedge *(v', v_1)$
$x \to s = \mathsf{null}$	$*(v_1, v_2) \wedge \neg\exists\, v', v'' :\ x(v') \wedge *(v', v'') \wedge s(v'', v_1)$
$x = y$	$*(v_1, v_2) \vee x(v_1) \wedge \exists\, v' :\ y(v') \wedge *(v', v_2)$
$x = *y$	$*(v_1, v_2) \vee x(v_1) \wedge \exists\, v', v'' :\ y(v') \wedge *(v', v'') \wedge *(v'', v_2)$
$x = \&y$	$*(v_1, v_2) \vee x(v_1) \wedge y(v_2)$
$x = \&y \to s$	$*(v_1, v_2) \vee x(v_1) \wedge \exists\, v', v'' :\ y(v') \wedge *(v', v'') \wedge s(v'', v_2)$
$x = y \to s$	$*(v_1, v_2) \vee x(v_1) \wedge \exists\, v', v'', v''' :$ $y(v') \wedge *(v', v'') \wedge s(v'', v''') \wedge *(v''', v_2)$
$x = \mathbf{up}(y, t, s)$	$*(v_1, v_2) \vee x(v_1) \wedge \exists\, v', v'' :\ y(v') \wedge *(v', v'') \wedge s(v_2, v'')$
$*x = y$	$*(v_1, v_2) \vee \exists\, v', v'' :\ x(v') \wedge *(v', v_1) \wedge y(v'') \wedge *(v'', v_2)$
$x \to s = y$	$*(v_1, v_2) \vee \exists\, v', v'', v''' :$ $x(v') \wedge *(v', v'') \wedge s(v'', v_1) \wedge y(v''') \wedge *(v''', v_2)$

Table 6.1: Predicate update formulas for nullification and assignments. The latter assume that $*x$, x, and $x \to s$ have been nullified.

types t occurring in the program, $\Pi = \bigcup_t \Pi(t)$, where

$$\Pi(\tau\, *) = \{\varepsilon\}$$
$$\Pi(\mathbf{struct}\ t\ \{\tau_1\ s_1, \ldots, \tau_k\ s_k\}) = \textstyle\bigcup_{i=1}^{k} \{s_i\}.\Pi(\tau_i) \cup \{s_i\}$$

As t ranges over record types, $\varepsilon \notin \Pi$; we write $\Pi_\varepsilon = \Pi \cup \{\varepsilon\}$.

Using the access paths from Π, we define the semantics of malloc as $[\![x = \mathbf{malloc}(t)]\!](U, \iota) = (U', \iota')$ where $U' = U \uplus \{u_\pi \mid \pi \in \Pi_\varepsilon(t)\}$ and

$$\iota'(s)(u_1, u_2) = \begin{cases} \iota(s)(u_1, u_2) & \text{if } u_1, u_2 \in U \\ 1 & \text{if } u_1 = u_\pi \wedge u_2 = u_{\pi.s} \\ 0 & \text{otherwise} \end{cases}$$

$$\iota'(*)(u_1, u_2) = \begin{cases} \iota(*)(u_1, u_2) & \text{if } u_1, u_2 \in U \\ 1 & \text{if } \iota(x)(u_1) \wedge u_2 = u_\varepsilon \\ 0 & \text{otherwise} \end{cases}$$

Analysis. As first example which goes beyond the one-node-per-record memory model, we consider a program which iterates over a singly-linked list pointed to by `x` using a pointer, `lpp`, to the `next` *component* of list elements, rather than to the elements themselves. The iteration is driven by the loop

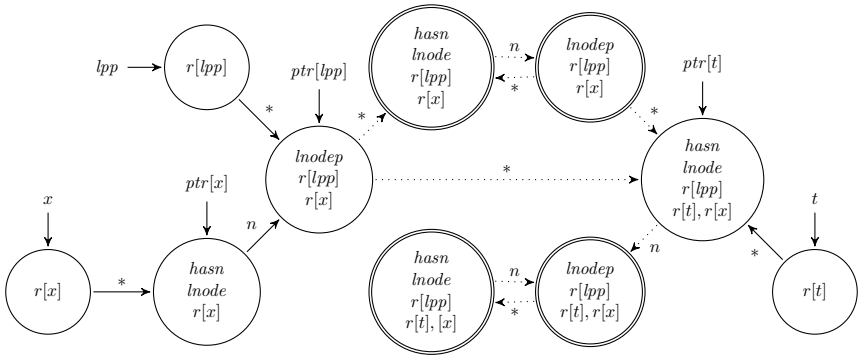

Figure 6.3: Shape graph during list traversal with indirect pointer. Double lines indicate summary nodes, solid arrows indicate definite edges, and dotted ones 1/2-edges. Predicates within a node do hold for this node.

```
for (lpp = &x; *lpp != NULL; lpp = &(*lpp)→next)
```

In the beginning `lpp` points to the *address* of `x`. It is advanced by dereferencing and taking the address of the `next` component of the next element. Once an element to be deleted is found, the assignment `*lpp = (*lpp)→next` removes it from the list. This routine is quite elegant in that it needs only one iterator and no check whether the iterator points to the first element or not. Also it uses pointers to components of records.

We implemented the creation of a fine-grained singly-linked list, the iteration over it, and the deletion of an element from it as outlined above in TVLA. The encoding of the fine-grained model into TVLA amounted in representing the $*$ predicate together with its update formulas for the basic statements. Through this encoding, we could re-use instrumentation predicates like sharing and reachability (`r[z]`: reachability from program variable `z`) to make the analysis go through and prove memory safety and well-formedness. Essentially, these come for free from TVLA. Additional instrumentation that we had to provide concerned type information (`lnode`, `lnodep`), the location to which pointer variables point (`ptr[z]`), and the fact that each record always has a `next` component (`hasn`).

In Figure 6.3, we show a sample shape graph, where summary nodes are denoted by double lines, definite edges by solid arrows, and 1/2 edges, which may or may not be there, by dotted arrows. Variable `t` points to an element in the middle of the list and is to be deleted using the code above. The snapshot is taken after the first iteration of the loop, where `lpp` was advanced once. The two *pairs* of summary nodes (double circles) represent any number (at least 1) of list elements before and after `t`. Each pair would be a single node in the standard TVLA memory model. Also observe, that `lpp` indeed points to the `n` *component* of the list

element pointed to by x.

6.3 Coarse-grained semantics and analysis

The fine-grained model from the last section has a very explicit view of the heap and allows for very detailed modelling. The number of individuals, though, in a logical structure is a multiple of the number occurring in standard TVLA based analyses because each component of a record is modeled by a separate individual. On top of that, care must be taken that individuals belonging to the same record — encoded as the outermost boxes in Figure 6.2 — are kept together, something not supported by standard TVLA.

Fortunately, we can atone for these drawbacks by exploiting the fact that the structure of a record is completely static. Once allocated, the interpretation of predicates in $\mathbf{Sel}$ never changes, only the $*$ predicate does. This observation suggests an encoding of records as single nodes after all, rather than representing them explicitly through a linked set of nodes — as in the one-node-per-record paradigm. Still, pointers to the head of a record need be distinguished from pointers to components. We do so by *parameterizing* the $*$ predicate. For example, if $*[p, n]$ is true of two individuals u and u', it means that the p-component of the record modeled by u holds a pointer to the n-component of the record modeled by u'. Analogously, we parameterize the unary predicates encoding pointer variables: if $x[n]$ holds of individual u, it means that x holds a pointer to the n-component of the record modeled by u. In the special case (which in practice is the most common) of a pointer to the head of a record, we write $x[\varepsilon]$.

Addresses of variables can be handled by adding one individual per variable exactly like in the fine-grained model. In order to simplify the presentation, though, we here omit addresses of pointer variables. Unlike in the fine-grained semantics, the unary predicate $x[\varepsilon]$ now holds for the individual pointed to by the pointer x, rather than for the location of x itself. Hence, taking address of pointers local pointers is not directly supported in this model, but can be simulated if temporary variables are introduced to denote the locations of pointer variables. Thus, the standard TVLA model is obtained from this version of the coarse-grained model by restricting predicates to the forms $*[s, \varepsilon]$ and $x[\varepsilon]$, i.e., all pointers point to the *heads* of records.

Recall the notion of an access path of the previous section. Using access paths, we define coarse-grained states as logical structures over the following signature, $\mathcal{D}$, serving as our set of core predicates.

$$\mathcal{D} = \{x[\pi]/1 \mid x \in \mathbf{Id}, \pi \in \Pi_\varepsilon\} \cup \{*[\pi_1, \pi_2]/2 \mid \pi_1 \in \Pi, \pi_2 \in \Pi_\varepsilon\}$$

st	φ_p^{st}
$x = \mathsf{null}$	$\varphi^{st}_{x[\pi]}(v) = 0$
$*x = \mathsf{null}$	$\varphi^{st}_{*[\pi,\pi']}(v_1, v_2) = *[\pi, \pi'](v_1, v_2) \wedge \neg x[\pi](v_1)$
$x \to s = \mathsf{null}$	$\varphi^{st}_{*[\pi.s,\pi']}(v_1, v_2) = *[\pi.s, \pi'](v_1, v_2) \wedge \neg x[\pi](v_1)$
$x = y$	$\varphi^{st}_{x[\pi]}(v) = y[\pi](v)$
$x = *y$	$\varphi^{st}_{x[\pi]}(v) = \exists\, v' :\ \bigvee_{\pi' \in \Pi} y[\pi'](v') \wedge *[\pi', \pi](v', v)$
$x = \&y$	not supported
$x = \&y \to s$	$\varphi^{st}_{x[\pi.s]}(v) = y[\pi](v)$
$x = y \to s$	$\varphi^{st}_{x[\pi]}(v) = \exists\, v' :\ \bigvee_{\pi' \in \Pi} y[\pi'](v') \wedge *[\pi'.s, \pi](v', v)$
$x = \mathbf{up}(y, t, s)$	$\varphi^{st}_{x[\pi]}(v) = y[\pi.s](v)$
$*x = y$	$\varphi^{st}_{*[\pi,\pi']}(v_1, v_2) = *[\pi, \pi'](v_1, v_2) \vee x[\pi](v_1) \wedge y[\pi'](v_2)$
$x \to s = y$	$\varphi^{st}_{*[\pi.s,\pi']}(v_1, v_2) = *[\pi.s, \pi'](v_1, v_2) \vee x[\pi](v_1) \wedge y[\pi'](v_2)$

Table 6.2: Predicate update formulas. Here, $\varphi_{x[\pi]}$ and $\varphi_{*[\pi,\pi']}$ denote rule schemes and stand for one rule per instance of π.

In order to complete the coarse-grained semantics, we provide the predicate update formulas for the predicates in $\mathcal{D}$. The update formulas shown in Table 6.2 constitute the state transformers both for the concrete and for the abstract semantics. These formulas are more concise than those of the fine-grained model. The update for the $\mathbf{up}()$ operation, e.g., only requires updating the predicates $x[\pi]$ to be true whenever the corresponding $y[\pi.s]$ used to be true. In the case of memory allocation, the effect of $x = \mathbf{malloc}(t)$ is to extend the universe with one fresh individual for which only the predicate $x[\varepsilon]$ holds.

Analysis. As for the fine-grained semantics, we implemented the coarse-grained transformers inside TVLA. As an example, we analyzed a program that first generates an hlist using the expanded `hlist_add` macro, which was already used in Figure 6.1, then iterates to some arbitrary point, and then deletes the element there using the `hlist_del` macro. The concrete C code of these macros is available from the `list.h` file of the current Linux distribution.

Being able to handle hlists is mandatory for verifying absence of races in programs such as in Figure 6.1. In our TVLA implementation, we parameterized the $*$ predicate with source and target components as described in the semantics. Other than that, we could migrate existing analysis specifications for doubly-linked lists to hlists. The analysis of doubly linked lists uses, e.g., the instrumentation predicate which says that first following the pointers n and then

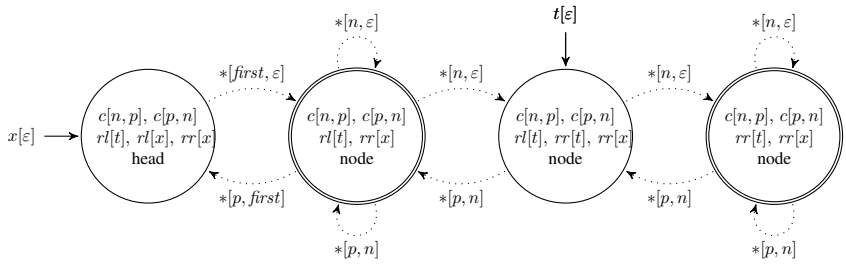

Figure 6.4: Shape graph obtained during hlist traversal.

p yields the same element. This predicate now is migrated to a predicate $c[n, p]$ stating that following $*[n, \varepsilon]$ and then $*[p, n]$ results in the same individual.

The sample shape of Figure 6.4 shows a situation where `t` points to the middle of an hlist. Note that there are two sorts of reachability: forward (`tr`, `rr[z]`) and backward (`tl` and `rl[z]`). This shows that `t` is indeed in the middle, well-formedness follows from the `c[]` predicates. Finally, observe that the back pointers either point to the `first` component of the head `x`, `*[p,first]`, or to the `n` component of a predecessor element (`*[p,n]`). In contrast, the forward pointer always points to the head of a record (`*[n,ε]`).

We successfully verified well-formedness and memory safety for the hlist example. Before we proceed to the example program of Figure 6.1, we investigate the formal relation between the fine-grained and the coarse-grained model in terms of expressiveness.

6.4 Fine-grained versus coarse-grained

Since the fine-grained model is more detailed, it is able to simulate the coarser one in a sense to be made explicit now.

We start by defining a mapping g from a coarse-grained structure $S_c = (U_c, \iota_c)$ into a fine-grained structure $g(S_c) = (U_f, \iota_f)$. The set of individuals of $g(S_c)$ is given by

$$U_f = \mathbf{Id} \cup \{u_\pi \mid u \in U_c, \pi \in \Pi_\varepsilon(\mathbf{type}(u))\}$$

The interpretation function ι_f then is given by:

$$\begin{array}{rll} \iota_f(*)(u_\pi, u'_{\pi'}) & \text{iff} & \iota_c(*[\pi, \pi'])(u, u') \\ \iota_f(*)(x, u_\pi) & \text{iff} & \iota_c(x[\pi])(u) \\ \iota_f(x)(v) & \text{iff} & v = x \\ \iota_f(s)(v, v') & \text{iff} & \exists u_{\pi.s} \in U_f.\ v = u_\pi \wedge v' = u_{\pi.s} \end{array}$$

where $\mathbf{type}(u) = t$ if u was created by $\mathbf{malloc}(t)$. Also we assume that S_c *respects types*, i.e., there are no pointers from or to a π component of node u if $\pi \notin \Pi(\mathbf{type}(u))$.

Since we deal with two different vocabularies, $\mathcal{C}$ and $\mathcal{D}$, on top of the mapping g between structures, a mapping $\mathcal{T}$ is required which translates formulas. Let φ be a FO(TC) formula over $\mathcal{D}$. The translation $\mathcal{T}$ commutes with boolean connectives and additionally is defined by:

$$\begin{aligned}
\mathcal{T}(*[\pi_1, \pi_2](v_1, v_2)) &= \exists v_1', v_2', v : \pi_1(v_1, v_1') \wedge \pi_2(v_2, v_2') \wedge *(v_1', v_2') \\
\mathcal{T}(x[\pi](v)) &= \exists v', v'' : x(v') \wedge \pi(v, v'') \wedge *(v', v'') \\
\mathcal{T}(\exists v : \varphi) &= \exists v : \mathit{head}(v) \wedge \mathcal{T}(\varphi) \\
\mathcal{T}((TC\, v_1, v_2 : \varphi)(v_3, v_4)) &= (TC\, v_1, v_2 : \mathit{head}(v_1) \wedge \mathit{head}(v_2) \wedge \mathcal{T}(\varphi))(v_3, v_4)
\end{aligned}$$

where for $\pi = s_1.\cdots.s_k \in \Pi$, the formula $\pi(v_0, v_k)$ is given by

$$\exists v_1, \ldots, v_{k-1} :\ s_1(v_0, v_1) \wedge \ldots \wedge s_k(v_{k-1}, v_k)$$

and where head holds for heads of records in a fine-grained structure only. A node is a head, if it is not the location of a variable and if it has no incoming $\mathbf{Sel}$ edge. The following theorem states that this translation preserves the valuation of formulas and that it commutes with state transformers, i.e., with predicate update formulas.

Theorem 8. *Let S_c be a type-respecting, coarse-grained logical structure and $S_f = g(S_c)$ the corresponding fine-grained structure. Then we have:*

1. *For every closed FO(TC) formula φ over $\mathcal{D}$, $[\![\varphi]\!]^{S_c} = [\![\mathcal{T}(\varphi)]\!]^{S_f}$.*
2. *For every basic statement st, $g([\![st]\!]_c(S_c)) = [\![st]\!]_f(S_f)$.*

Proof. For an induction argument, we prove the statement for open formulas. Let $Z_c\colon \mathsf{Var} \to U_c$ be an assignment of logical variables to individuals in the coarse-grained universe; we define $Z_f = g(Z_c)\colon \mathsf{Var} \to U_f$ as an assignment selecting the head u_ε for each record u. We show $[\![\varphi]\!]^{S_c}(Z_c) = [\![\mathcal{T}(\varphi)]\!]^{S_f}(Z_f)$ by induction on φ. For the core predicates, we compute for $u = Z_c(v)$:

$$\begin{aligned}
[\![x[\pi](v)]\!]^{S_c}(Z_c) &= \iota_c(x[\pi])(u) = \iota_f(*)(x, u_\pi) \\
&= \exists u' \in U_f :\ \pi(u_\varepsilon, u') \wedge \iota_f(*)(x, u') \\
&= [\![\mathcal{T}(x[\pi](v))]\!]^{S_f}(Z_f)
\end{aligned}$$

And analogously for the binary predicates. We need to further consider cases for $\wedge$, $\exists$, $\neg$, and TC (as the rest follows from DeMorgan's Laws). Conjunction and

negation are obvious, while existential quantification and transitive closure rely on the restriction of quantification to heads of records. We consider existential quantification, for which we observe:

$$\begin{aligned}
[\![\exists v : \varphi]\!]^{S_c}(Z_c) &= \exists u \in U_c : [\![\varphi]\!]^{S_c}(Z_c[v \mapsto u]) \\
&= \exists u \in U_c : [\![\mathcal{T}(\varphi)]\!]^{S_f}(Z_f[v \mapsto u_\varepsilon]) \\
&= \exists u' \in U_f : head(u') \wedge [\![\mathcal{T}(\varphi)]\!]^{S_f}(Z_f[v \mapsto u']) \\
&= [\![\exists v : head(v) \wedge \mathcal{T}(\varphi)]\!]^{S_f}(Z_f) = [\![\mathcal{T}(\exists v : \varphi)]\!]^{S_f}(Z_f)
\end{aligned}$$

This completes the proof of the first statement. For the second statement, let $S_c = (U_c, \iota_c)$ denote a coarse-grained logical structure. We do a case distinction on the form of basic statements.

Consider, e.g., the statement st given by $x = \mathbf{up}(y, t, s)$. If it exists, let $u \in U_c$ denote the unique individual for which $\iota_c(y[\pi.s])$ holds for some access path π. Then $[\![st]\!]_c(S_c) = S'_c = (U'_c, \iota'_c)$ where $U'_c = U_c$ and ι'_c equals ι_c up to the predicate $x[\pi]$, which is updated such that $\iota'_c(x[\pi])(u')$ holds iff $u' = u$. Let $S_f = g(S_c)$ denote the fine-grained structure corresponding to S_c. This generates for $u \in U_c$ the head $u_\varepsilon \in U_f$ as well as its components, including u_π and $u_{\pi.s}$. Since we assumed that $\iota_c(y[\pi.s])(u)$ is true in S_c, we know that $\iota_f(*)(y, u_{\pi.s})$ must hold in S_f. Thus, $[\![st]\!]_f(S_f) = S'_f = (U'_f, \iota'_f)$ where the set of individuals are the same as S_f and ι'_f equals ι_f up to the predicate $*$ which now additionally holds for the pair (x, u_π). Ultimately, the only change to S_f and S_c is that $\iota'_c(x[\pi])(u)$ holds in S'_c and $\iota'_f(*)(u, u_\pi)$ holds in S'_f. As this is in accord with the definition of g, we conclude that $g(S'_c) = S'_f$. This holds also if $\iota_c(y[\pi.s])$ is false everywhere, in which case $S'_c = S_c$ and $S'_f = S_f$. □

The theorem effectively constitutes a simulation result between fine-grained and coarse-grained semantics. Notice that the restriction of quantified variables to heads of records in the translation $\mathcal{T}$ is an important one. It also demonstrates exactly *how* fine-grained structures are finer: they can talk about record components explicitly and quantify over them, while components occur only implicitly in the coarse-grained model.

Part 1 of Theorem 8 can be lifted to abstract states as well. Assume an abstract, three-valued coarse-grained structure S_c^3 and any two-valued coarse-grained structure S_c^2 such that $S_c^2 \sqsubseteq S_c^3$ using the embedding order of [75]. Then any formula ψ of FO(TC) that holds for S_c^3 also holds for S_c^2 by the Embedding Theorem. By Theorem 8, $\mathcal{T}(\psi)$ holds in $S_f = g(S_c^2)$. If S_c^3 was obtained by the set A of abstraction predicates, then ψ will also hold in the canonical abstraction of S_f using $\mathcal{T}(A)$ as abstraction predicates. Lifting part 2 of Theorem 8 is far more involved, because it needs to take materialization strategies into account.

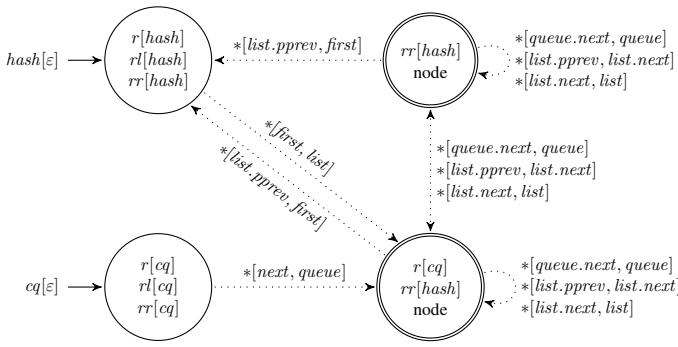

Figure 6.5: Shape graph obtained while analyzing overlapping data-structure.

6.5 Application

Let us finally consider the motivating program from the Introduction. Its code is shown in Figure 6.1 and a typical memory configuration in Figure 6.2. In order to argue about data races in the presence of privatization, reachability information is crucial. In particular, one must reason about reachability along different embedded lists. For instance, in Figure 6.2, only the first and the third node are in the queue, whereas all three are in the list.

In order to find out if the techniques are worth implementing in the Goblint analyzer, we wanted to find out if the techniques work for our motivating example. Jörg Kreiker therefore implemented the analysis in the dedicated shape analysis framework TVLA.[1] However, the TVLA tool does not natively support computations on predicates as necessary to conveniently express the string manipulation on access paths as used in the update formulas of Table 6.2. This makes the implementation cumbersome and look clumsy in places. Also, it introduces a lot of superfluous predicates and coercion constraints greatly slowing down the tool. This, however, is not a principal restriction of our memory model but the lack of tool support. Therefore, we had to settle for a proof of concept implementation where the cleanup queue is actually a singly-linked list.

First, we analyzed a program creating a structure like that of Figure 6.2 from scratch. This amounts to iterating the code

```
n = malloc(sizeof(node));
hlist_add_head(&n→list,&hash);
if (?) list_add(&n→queue, &cq);
```

[1]The precise definitions can be found at `http://www7.in.tum.de/~joba/overlap.tgz`.

After this loop, four shapes are obtained, the most general of which is shown in Figure 6.5. It shows that (i) all nodes are reachable from `hash`, which is the head of the hlist component, a fact indicated by `rr[hash]`; and (ii) only some nodes are reachable from `cq`, indicated by `r[cq]`. This is the arbitrary subset of nodes added to the cleanup queue. Also it shows the $*$ predicates with parameters like `list.pprev`, denoting the pprev component of the hlist component of a node. Recall that `rr[x]` (`rl[x]`) means reachability from `x` along forward (backward) pointers in a doubly-linked list, while `r[x]` is just singly-linked list reachability — which is how we implement the cleanup queue.

Subsequently, the elements of the queue are to be removed from the hlist component using

```
list_for_each_entry(n, &cq, queue) {
    hlist_del(&n→list); }
```

Here, the challenge for the analysis is the *change of views* implied by traversing the queue and then removing from the hlist. A lot of reachability information is lost; in fact, properties like absence of memory leaks cannot be guaranteed by this analysis. Still, we are able to prove that an element is deleted from the queue using the very same routine that was used in a non-embedded record in Section 6.3. Thus, we can infer that the element is no longer reachable from the thread-shared data.

Again, the analysis specifications are available online. Even in this most complicated scenario, the analysis time was just a few seconds.

6.6 Related work

The body of work on shape analysis is too large to do equal justice to all techniques. Approaches based on regular model checking [7], symbolic backwards reachability analysis [1], or decision procedures such as [8] seem not to have dealt with the analysis of low-level system code, much less with overlapping records. There are a number of approaches that make use of numeric reasoning to deal with pointer arithmetic. While pioneered by Deutsch [24], who used numeric domains to constrain access paths, Gulwani and Tiwari [38] provide a C semantics which perhaps is even more explicit about blocks and offsets as ours. However, it is unable to deal with structures such as doubly-linked lists. Recent work [35] combines numeric and shape domains. It is focused on tracking partition sizes to prove memory safety and sometimes even termination in presence of arrays of dynamically allocated structures. So far, however, none of the above treats overlapping records.

More direct approaches to shape analysis are based on either TVLA [75] or separation logic [71]. As stated before, most work in the TVLA setting focuses on

higher-level programming languages; however, Dor's thesis [25] and subsequent work provided a semantics of low-level C similar to our fine-grained semantics and program analyses based on this semantics. These analyses are mostly concerned with string manipulations.

As for separation logic based approaches which deal with a low-level C semantics and with pointer arithmetic, one early work is [10], which however specifically targets the data-structure of multiword lists. Berdine et al. [5] present a shape analysis of composite data-structures which can reason about lists of lists. They explicitly identify shape analysis of overlapping or embedded structures as presented here as a limitation to their approach. Also, Chang and Rival [14] present a shape analysis based on separation logic and user-specified data-structure specifications called checkers. It also treats combinations of numerical and shape domains, but overlapping records are not considered. Despite some impressive improvements recently [11,93], in particular concerning scalable shape analyses of real code, a formal treatment of overlapping records has yet to be reported. In addition, most of that work focuses exclusively on memory safety rather than on subtle reachability problems as we face.

Separation logic is also used in the broader context of modular verification and extended static checking. There, one relies on specifications of components, and the analysis operates under the assumption that other components behave as specified [4, 20, 32]. The fine-grained memory model we use for shape analysis is also used by the VCC C verifier [18]; in particular, it uses an implicit type-system to verify that distinct pointers do not reference overlapping objects [19]. In the HAVOC verifier [15], a particular reachability predicate is employed which also works on a semantics resembling ours, but is much more numeric in nature, focused on pointer arithmetic. Other techniques exist for dealing with the heap in modular verification, including ownership [17], which is used by Spec# and Java/JML; dynamic frames [49, 80], which is used by VeriCool 1 and Dafny; and implicit dynamic frames [81], which are used in VeriCool 3 and Chalice.

Our interest in shape analysis of overlapping records is derived from attempts to verify absence of data races in low-level C. In static race detection, dynamic memory is treated at a fairly superficial level by *blobbing* together objects into static allocation sites. There are techniques for verifying mutually exclusive access to heap objects when each record contains its own dedicated lock [67]; and analyses relying on reachability information, such as *disjoint reachability analysis* [62] and the *region analysis* of Chapter 4, have been employed to ensure correct synchronization of accesses to disjoint regions of dynamically allocated memory. These analyses, however, cannot deal with object privatization and overlapping structures as occur in our example. On the other hand, by virtue of not tracking the state of the heap at each program point, such analyses can be directly used

in a concurrent setting, while our approach requires adaptations of the TVLA approach to handle concurrency [6, 55].

6.7 Conclusion

We presented a shape analysis for overlapping data-structures, which are ubiquitous in low-level systems code. Using our prototype implementation we were able to establish subtle reachability properties as required, e.g., for reasoning about data races in system code with overlapping records.

For that, we introduced two refinements of existing memory models. This enabled us to implement both approaches within the TVLA framework. While this was a proof-of-concept implementation, having the shape analysis performed by a dedicated shape analysis tool would allow us to benefit from any future improvements of the TVLA tool. However, as dynamic manipulation of predicates is not natively supported by the TVLA tool, we would like to directly implement this shape analysis, or a simplified version tailored to cyclic doubly linked lists, directly in Goblint.

The step from fine-grained to coarse-grained semantics is essentially a technique of encoding statically known parts of graph structures like the internal structure of records into syntax. Somehow similar, separation logic based approaches rely on inductively defined predicates capturing data-structures. This connection might be exploited to enable the use of different formalisms for different parts of the heap in a common setting.

CHAPTER 7

CONCLUSIONS

We have provided methods to address the most significant challenges in race detection for heap-manipulating low-level C programs. In order to deal with fine-grained locking idioms, we provided a must-alias analysis which allows reasoning about field and index correlations. We then tackled problems of medium-grained locking by introducing a region analysis. The key idea was to move away from allocation site abstraction and instead associate accesses into the heap with static owners of the region into which an element belongs. Thus, the same techniques can be applied as for fine-grained locking to associate the locks with the owners of regions. Finally, in order to deal with migrating elements and privatization, we refined the global invariant approach to deal with temporary violations of heap invariants within critical sections. Using a low-level shape analysis, we were able to show when some element has been removed from the Linux kernel's embedded list structures.

While these techniques are sufficient to deal with many difficult situations that occur in kernel code, putting these techniques together into a coherent analysis framework is still ongoing work. Many of the experimental results were on fairly small code sections that precisely exhibited the behaviour we were interested in analyzing. Currently, we are targeting a more significant piece of the Linux kernel, the USB subsystem, with the aim of analyzing it fully. In addition to generalizing the current techniques, there are a number of issues we need to solve before we can claim victory.

One important area we have so far neglected is synchronization sensitivity. Consider, as an example [68], the scenario where we have a main thread with k worker threads. The main thread maintains an array A with k elements, one for each thread, such that $A[i]$ is manipulated by thread i. Furthermore, suppose the main thread initializes the array before spawning worker threads and processes the array after all workers have terminated. Although there is no locking, the program is free from races because the main thread may only access the array when

the workers do not and the workers follow a convention that ensures mutually exclusive access.

There are distinct *temporal* phases in a program, such as initialization, processing, and post-processing. An analyzer must, therefore, determine not only which threads may run in parallel, but whether two given accesses may actually conflict, taking into account which threads accesses what data at what time. In the example, the main is still running when the worker threads start, but it no longer touches the array. The conventional approach is to attempt to partially order statements when it is clear that something must *happen before* another operation. A race can then only occur on two accesses that lack ordering constraints. This is exploited by many dynamic analyzers, such as the Intel Thread Checker. For static analysis of C, acquisition histories, proposed by Kahlon et al. [45], is an interesting approach to improving synchronization-sensitivity. Although some progress has been made in this regard [46], much of the practical synchronization in programs are based on deeper properties of the program logic, or rely on various complicated synchronization primitives, like signals, conditional variables, wait-queues, etc.

Once we are satisfied with our race detection tool, we plan on extending the analyzer to deal with additional concurrency problems such as dead-locks and atomicity problems. There is also significant interest from our industrial partners to adapt our race detection techniques for specific concurrency frameworks, such as the Autosaar/OSEK operating system, used in the automotive domain.

Bibliography

[1] Abdulla, P.A., Bouajjani, A., Cederberg, J., Haziza, F., Rezine, A.: Monotonic abstraction for programs with dynamic memory heaps. In: CAV'08. LNCS, vol. 5123, pp. 341–354. Springer (2008)

[2] Artho, C., Havelund, K., Biere, A.: Using block-local atomicity to detect stale-value concurrency errors. In: ATVA'04. LNCS, vol. 3299, pp. 150–164. Springer (2004)

[3] Attiya, H., Ramalingam, G., Rinetzky, N.: Sequential verification of serializability. In: POPL'10. pp. 31–42 (2010)

[4] Barnett, M., Chang, B., DeLine, R., Jacobs, B., Leino, K.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO'06. LNCS, vol. 4111, pp. 364–387. Springer (2006)

[5] Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., Yang, H.: Shape analysis for composite data structures. In: CAV'07. LNCS, vol. 4590, pp. 178–192. Springer (2007)

[6] Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. In: CAV'08. LNCS, vol. 5123, pp. 399–413 (2008)

[7] Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: CAV'00. LNCS, vol. 1855, pp. 403–418. Springer (2000)

[8] Bouillaguet, C., Kuncak, V., Wies, T., Zee, K., Rinard, M.C.: Using first-order theorem provers in the Jahob data structure verification system. In: VMCAI'07. LNCS, vol. 4349, pp. 74–88. Springer (2007)

[9] Boyapati, C., Salcianu, A., Beebee, W., Rinard, M.: Ownership types for safe region-based memory management in real-time Java. In: PLDI'03. pp. 324–337. ACM Press (2003)

[10] Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In: SAS'06. LNCS, vol. 4134, pp. 182–203. Springer (2006)

[11] Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL'09. pp. 289–300. ACM Press (2009)

[12] Calcagno, C., Distefano, D., Vafeiadis, V.: Bi-abductive resource invariant synthesis. In: APLAS'09. pp. 259–274 (2009)

[13] Chakrabarti, A., de Alfaro, L., Henzinger, T., Jurdziński, M., Mang, F.: Interface compatibility checking for software modules. In: CAV'02. pp. 654–663. LNCS, vol. 2404, Springer (2002)

[14] Chang, B.Y.E., Rival, X.: Relational inductive shape analysis. In: POPL'08. pp. 247–260. ACM Press (2008)

[15] Chatterjee, S., Lahiri, S., Qadeer, S., Rakamarić, Z.: A reachability predicate for analyzing low-level software. In: TACAS'07. LNCS, vol. 4424, pp. 19–33. Springer (2007)

[16] Chugh, R., Voung, J.W., Jhala, R., Lerner, S.: Dataflow analysis for concurrent programs using datarace detection. In: PLDI'08. pp. 316–326. ACM Press (2008)

[17] Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA'98. pp. 48–64. ACM Press (1998)

[18] Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: TPHOL'09. LNCS, vol. 5674, pp. 23–42. Springer (2009)

[19] Cohen, E., Moskal, M., Tobies, S., Schulte, W.: A precise yet efficient memory model for C. In: SSV'09. ENTCS, vol. 254, pp. 85–103. Elsevier (2009)

[20] Condit, J., Hackett, B., Lahiri, S.K., Qadeer, S.: Unifying type checking and property checking for low-level code. In: POPL'09. pp. 302–314. ACM Press (2009)

[21] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL'77. pp. 238–252. ACM Press (1977)

[22] Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive programs. In: Neuhold, E. (ed.) Formal Descriptions of Programming Concepts, pp. 237–277. North-Holland Publishing Company (1977)

[23] Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: PLDI'02. pp. 57–68. ACM Press (2002)

[24] Deutsch, A.: Interprocedural may-alias analysis for pointers: beyond k-limiting. In: PLDI'94. pp. 230–241. ACM Press (1994)

[25] Dor, N.: Automatic Verfication of Program Cleanness. Master's thesis, Tel Aviv University (2003)

[26] Engler, D., Ashcraft, K.: RacerX: effective, static detection of race conditions and deadlocks. In: SOSP'03. pp. 237–252. ACM Press (2003)

[27] Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: OSDI'00. pp. 1–16. USENIX Association (2000)

[28] Fähndrich, M., Rehof, J., Das, M.: Scalable context-sensitive flow analysis using instantiation constraints. In: PLDI'00. pp. 253–263. ACM Press (2000)

[29] Fecht, C., Seidl, H.: A faster solver for general systems of equations. Sci. Comput. Program. 35(2), 137–161 (1999)

[30] Flanagan, C., Freund, S., Qadeer, S., Seshia, A.A.: Modular verification of multithreaded programs. Theor. Comp. Science 338, 153–183 (2005)

[31] Flanagan, C., Freund, S.N.: Detecting race conditions in large programs. In: PASTE'01. pp. 90–96. ACM Press (2001)

[32] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI'02. pp. 234–245. ACM Press (2002)

[33] Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: POPL'05. pp. 338–350. ACM Press (2005)

[34] Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: PLDI'07. pp. 266–277. ACM Press (2007)

[35] Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: POPL'09. pp. 239–251. ACM Press (2009)

[36] Gulwani, S., Necula, G.C.: A polynomial-time algorithm for global value numbering. In: SAS'04. pp. 212–227. LNCS, vol. 3148, Springer (2004)

[37] Gulwani, S., Tiwari, A.: Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In: ESOP'06. pp. 279–293. LNCS, vol. 3924, Springer (2006)

[38] Gulwani, S., Tiwari, A.: An abstract domain for analyzing heap-manipulating low-level software. In: CAV'07. LNCS, vol. 4590, pp. 379–392. Springer (2007)

[39] Gulwani, S., Tiwari, A.: Computing procedure summaries for interprocedural analysis. In: ESOP'07. pp. 253–267. LNCS, vol. 4421, Springer (2007)

[40] Hackett, B., Rugina, R.: Region-based shape analysis with tracked locations. In: POPL'05. pp. 310–323. ACM Press (2005)

[41] Hind, M., Burke, M., Carini, P., Choi, J.D.: Interprocedural pointer alias analysis. ACM Trans. Progr. Lang. Syst. 21(4), 848–894 (1999)

[42] Holzmann, G.J.: The power of ten: Rules for developing safety critical code. IEEE Computer 39(6), 95–97 (2006)

[43] Jonkers, H.B.M.: Abstract storage structures. In: Algorithmic Languages. pp. 321–343. IFIP (1981)

[44] Kahlon, V.: Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In: PLDI'08. pp. 249–259. ACM Press (2008)

[45] Kahlon, V., Ivančić, F., Gupta, A.: Reasoning about threads communicating via locks. In: CAV'05. LNCS, vol. 3576, pp. 505–518. Springer (2005)

[46] Kahlon, V., Sinha, N., Kruus, E., Zhang, Y.: Static data race detection for concurrent programs with asynchronous calls. In: ESEC/FSE'09. pp. 13–22. ACM Press (2009)

[47] Kahlon, V., Yang, Y., Sankaranarayanan, S., Gupta, A.: Fast and accurate static data-race detection for concurrent programs. In: CAV'07. LNCS, vol. 4590, pp. 226–239. Springer (2007)

[48] Karr, M.: Affine relationships among variables of a program. Acta Inf. 6(2), 133–151 (1976)

[49] Kassios, I.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: FM'06. LNCS, vol. 4085, pp. 268–283. Springer (2006)

[50] Kildall, G.A.: A unified approach to global program optimization. In: POPL'73. pp. 194–206. ACM Press (1973)

[51] Knoop, J., Steffen, B.: The Interprocedural Coincidence Theorem. In: CC'92. LNCS, vol. 641, pp. 125–140. Springer (1992)

[52] Kreiker, J., Seidl, H., Vojdani, V.: Shape analysis of low-level C with overlapping structures. In: VMCAI'10. LNCS, vol. 5944, pp. 214–230. Springer (2010)

[53] Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R.A., Zhou, Y.: MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In: SOSP'07. pp. 103–116. ACM Press (2007)

[54] Manevich, R., Berdine, J., Cook, B., Ramalingam, G., Sagiv, M.: Shape analysis by graph decomposition. In: TACAS'07. LNCS, vol. 4424. Springer (2007)

[55] Manevich, R., Lev-Ami, T., Sagiv, M., Ramalingam, G., Berdine, J.: Heap decomposition for concurrent shape analysis. In: SAS'08. LNCS, vol. 5079, pp. 363–377 (2008)

[56] Müller-Olm, M., Rüthing, O., Seidl, H.: Checking Herbrand equalities and beyond. In: VMCAI'05. pp. 79–96. LNCS, vol. 3385, Springer (2005)

[57] Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL'04. pp. 330–341. ACM Press (2004)

[58] Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. ACM Trans. Progr. Lang. Syst. 29(5) (2007)

[59] Müller-Olm, M., Seidl, H.: Upper adjoints for fast inter-procedural variable equalities. In: ESOP'08. LNCS, vol. 4960, pp. 178–192. Springer (2008)

[60] Müller-Olm, M., Seidl, H., Steffen, B.: Interprocedural Herbrand equalities. In: ESOP'05. pp. 31–45. LNCS, vol. 3444, Springer (2005)

[61] Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: PLDI'06. pp. 308–319. ACM Press (2006)

[62] Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: POPL'07. pp. 327–338. ACM Press (2007)

[63] Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: Cil: An infrastructure for C program analysis and transformation. In: CC'02. LNCS, vol. 2304, pp. 213–228. Springer (2002)

[64] Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Progr. Lang. Syst. 1(2), 245–257 (1979)

[65] Netzer, R.H.B., Miller, B.P.: What are race conditions? Some issues and formalizations. ACM Lett. Program. Lang. Syst. 1(1), 74–88 (1992)

[66] Paterson, M., Wegman, M.N.: Linear unification. In: STOC'76. pp. 181–186. ACM Press (1976)

[67] Pratikakis, P., Foster, J.S., Hicks, M.: Existential label flow inference via CFL reachability. In: SAS'06. LNCS, vol. 4134, pp. 88–106. Springer (2006)

[68] Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Context-sensitive correlation analysis for detecting races. In: PLDI'06. pp. 320–331. ACM Press (2006)

[69] Rehof, J., Fähndrich, M.: Type-based flow analysis: from polymorphic subtyping to CFL-reachability. In: POPL'01. pp. 54–66. ACM Press (2001)

[70] Reps, T., Sagiv, M., Loginov, A.: Finite differencing of logical formulas for static analysis. In: ESOP'03. LNCS, vol. 2618, pp. 380–398. Springer (2003)

[71] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS'02. pp. 55–74. IEEE Press (2002)

[72] Rinard, M.: Analysis of multithreaded programs. In: SAS '01. LNCS, vol. 2126, pp. 1–19. Springer (2001)

[73] Rinetzky, N., Bauer, J., Reps, T.W., Sagiv, S., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: POPL'05. pp. 296–309 (2005)

[74] Rugina, R., Rinard, M.C.: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. ACM Trans. Progr. Lang. Syst. 27(2), 185–235 (2005)

[75] Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Progr. Lang. Syst. 24(3), 217–298 (2002)

[76] Seidl, H., Vene, V., Müller-Olm, M.: Global invariants for analyzing multithreaded applications. Proc. of the Estonian Academy of Sciences: Phys., Math. 52(4), 413–436 (2003)

[77] Seidl, H., Vojdani, V.: Region analysis for race detection. In: SAS'09. LNCS, vol. 5673, pp. 171–187. Springer (2009)

[78] Seidl, H., Vojdani, V., Vene, V.: A smooth combination of linear and Herbrand equalities for polynomial time must-alias analysis. In: FM'09. LNCS, vol. 5850, pp. 644–659. Springer (2009)

[79] Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Jones, N., Muchnick, S. (eds.) Program Flow Analysis: Theory and Applications, pp. 189–234. Prentice Hall (1981)

[80] Smans, J., Jacobs, B., Piessens, F.: VeriCool: an automatic verifier for a concurrent Object-Oriented language. In: FMOODS'08. LNCS, vol. 5051, pp. 220–239. Springer (2008)

[81] Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: ECOOP'09. LNCS, vol. 5653, pp. 148–172. Springer (2009)

[82] Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for Java. In: ECOOP'08. LNCS, vol. 5142, pp. 104–128. Springer (2008)

[83] Steensgaard, B.: Points-to analysis in almost linear time. In: POPL'96. pp. 32–41. ACM Press (1996)

[84] Steffen, B., Knoop, J., Rüthing, O.: The value flow graph: A program representation for optimal program transformations. In: ESOP'90. pp. 232–247. LNCS, vol. 1694, Springer (1990)

[85] Tofte, M., Birkedal, L.: A region inference algorithm. ACM Trans. Progr. Lang. Syst. 20(4), 724–767 (1998)

[86] Vafeiadis, V.: RGSep action inference. In: VMCAI'10. LNCS, vol. 5944, pp. 345–361. Springer (2010)

[87] Vojdani, V., Vene, V.: Goblint: Path-sensitive data race analysis. In: SPLST'07. pp. 130–141. Eötvös Lorand Univ. (2007)

[88] Vojdani, V., Vene, V.: Goblint: Path-sensitive data race analysis. Annales Univ. Sci. Budapest., Sect. Comp. 30, 141–155 (2009)

[89] Voung, J.W., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: ESEC/FSE'07. pp. 205–214. ACM Press (2007)

[90] Wang, X., Xu, Z., Liu, X., Guo, Z., Wang, X., Zhang, Z.: Conditional correlation analysis for safe region-based memory management. In: PLDI'08. pp. 45–55. ACM Press (2008)

[91] Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Trans. Progr. Lang. Syst. 13(2), 181–210 (1991)

[92] Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent linux device drivers. In: ASE'07. pp. 501–504. ACM Press (2007)

[93] Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable shape analysis for systems code. In: CAV'08. LNCS, vol. 5123, pp. 385–398 (2008)

ACKNOWLEDGMENTS

First and foremost, I would like to thank my wife, Maris, for her magic; indeed, she is a magical creature, a *goblin*, and that is the term of endearment I have used to address her for as long as I can remember. The analyzer which has been my obsession for most of my academic life was named after her, although it had to be subsequently renamed to avoid confusion and jealousy between the two.

The Goblint project would never have gotten anywhere if it were not for my supervisors, Prof. Varmo Vene and Prof. Helmut Seidl. I am very grateful to Varmo for getting me interested in this field, and I sincerely thank Helmut for gently and patiently teaching me how to write papers; that is, for gently butchering my attempted formalisations and patiently killing my flawed proofs. I am heavily indebted to my collaborators: Jörg Kreiker, Martin Schwarz, and Kalmer Apinis. In particular, Kalmer's work has been invaluable, and by now, he has refactored Goblint beyond recognition. I also thank my opponents, Prof. Alan Mycroft and Dr. Keiko Nakata, for finding so many faults in an earlier draft of this work. Finally, and it feels awkward because this is the first time in my life that I put my parents last — I would like to thank my parents and, of course, my dear sister.

This dissertation was financially supported by the Doctoral School of Information and Communication Technology (IKTDK) and the Estonian Science Foundation under grants No. 6713 and 8421. During the last three years of work, I was employed by the Technische Universität München.

DÜNAAMILIST MÄLU TÖÖTLEVATE C PROGRAMMIDE STAATILINE ANDMEJOOKSUDE ANALÜÜS

Mitmelõimeline programmeerimine võimaldab kirjutada programme, mis samaaegselt lahendavad mitut ülesannet. See on vajalik, et programmerida interaktiivset kasutajaliidest või jagada arvutusi mitme protsessori vahel. Paraku on mitmelõimeline programmeerimine äärmiselt vigadealdis, eriti kui mitu lõime manipuleerivad samu ressursse. Erinevalt järjestikulisest programmeerimisest, ei ole programmis kirjeldatud käskude täitmisjärjekord rangelt fikseeritud, vaid sõltub juhuslikest teguritest ja väliskeskkonnast. Seetõttu on traditsioonilise testimise tõhusus mitmelõimeliste programmide puhul küllaltki tagasihoidlik. Mis veelgi olulisem, testimisega on küll võimalik näidata vigade olemasolu, aga mitte anda garantiisid, et süsteemi rohkem vigu ei jäänud.

Kõrget usaldusväärsust nõudvate süsteemide korral on rangelt matemaatiliselt põhjendatud formaalsete meetodide kasutamine hädavajalik. Nagu ehitusinsenerid saavad ligikaudses mudelis simuleerida, kuidas ehitis peab vastu füüsilistele survetele, on võimalik arvutiprogrammi põhjal püstitada andmevoogu kirjeldavad võrrandid ning neid lahendades saada kindlaid garantiisid programmi käitumise kohta. Abstraktsel interpretatsioonil põhinev staatiline analüüs võimaldab seega ilma programmi käivitamata arvutada välja programmi kõigi võimalike käitumiste ülemhulga.

Andmejooks (*data race*) on olukord mitmelõimelises süsteemis, kus mitu lõime üritavad samaaegselt ühele ja samale jagatud ressursile ligi pääseda. Probleemide vältimiseks, kasutatakse sünkroniseerimismehhanisme, mis garanteerivad, et programmi kriitilistesse sektsioonidesse saab siseneda ainult üks lõim korraga. Programmerija ülesandeks on olukorras, kus kaks lõime püüavad samale jagatud ressursile ligi pääseda, tagada, et mõlemad lõimed lukustaksid sama luku. Staatilise andmejooksude analüüsi eesmärgiks on kindlaks teha, kas kõigi jagatud ressursside jaoks on ülaltoodud tingimus korrektselt täidetud, millest järeldub, et programmis andmejookse ei esine.

Käesolev doktoritöö uurib andmejooksude tuvastamist programmides, kus jagatud ressursiks on dünaamiliselt (programmi täitmise ajal) allokeeritud mälu ning ka sünkroniseerimiseks kasutatavad lukud luuakse alles programmi täitmise ajal. Täpse ja efektiivse staatilise analüüsi teeb selliste programmide korral raskeks just see, et kõik analüüsi pidepunktid luuakse dünaamiliselt. Töö põhitulemustena on välja töötatud kolm uudset meetodit, et lahendada dünaamilist mälu manipuleerivate programmide analüüsiga seotud küsimusi.

1. On esitatud viitade analüüsi meetod, mis võimaldab vastata päringutele aadressavaldiste võrduste kohta. Sellega saab korrektselt analüüsida lukustamist peene granulaarsusega lukustamisskeemide puhul, s.t., kui lukud asuvad jagatud ressurssides endas.

2. Välja on töötatud dünaamiliselt allokeeritud mälu regioonabstraktsioon, mille abil saab eelmist meetodit laiendada jämedama lukustusgranulaarsusega programmidele. Meetodi põhiidee on seostada täitmisajal loodud objektid staatiliselt kirjeldatavate omanikega.

3. On esitatud uudne meetod teatud invariantsustingimusi lokaalselt rikkuvate progammide staatiliseks analüüsiks. See võimaldab korrektselt analüüsida näiteks programme, kus üldiselt eeldatakse, et sama ressurssi kaitstakse alati sama lukuga, kuid lühiajaliselt (näit. objektide migreerimisel või eemaldamisel jagatud andmestruktuuridest) võib see tingimus mitte kehtida.

Kõik töös vaadeldud meetodid on realiseeritud mitmelõimeliste C programmide analüsaatorite raamistikus Goblint. Meetodite efektiivsust ja praktilisust testiti edukalt mitmete vabavaraliste programmide ja operatsioonisüsteemi Linux tuumamoodulite analüüsiga.

CURRICULUM VITAE

Personal data

Name	Vesal Vojdani
Birth	June 26, 1980 Huddinge, Sweden
Citizenship	Swedish
Marital Status	Married
Languages	Swedish, English, Estonian, Persian, German
Address	Faberstraße 13 81373 München Germany
Contact	+49 176 80087053 vesal@ut.ee

Education

2005–	University of Tartu, Ph.D. candidate
2003–2005	University of Tartu, M.Sc. in Computer Science
1999–2003	University of Tartu, B.Sc. in Computer Science
1994–1999	Tallinn English College (Estonia), secondary education
1987–1994	Irstaskolan (Västerås, Sweden), primary education

Employment

2007–	Technische Universität München, Wissenschaftlicher Mitarbeiter
2005–2007	University of Tartu, programmer

ELULOOKIRJELDUS

Isikuandmed

Nimi	Vesal Vojdani
Sünniaeg ja -koht	26. juuni 1980 Huddinge, Rootsi
Kodakondsus	rootslane
Perekonnaseis	abielus
Keelteoskus	rootsi, inglise, eesti, pärsia, saksa
Aadress	Faberstraße 13 81373 München Saksamaa
Kontaktandmed	+49 176 80087053 vesal@ut.ee

Haridustee

2005–	Tartu Ülikool, doktorant
2003–2005	Tartu Ülikool, MSc informaatikas
1999–2003	Tartu Ülikool, BSc informaatikas
1994–1999	Tallinna Inglise Kolledž, keskharidus
1987–1994	Irstaskolan (Västerås, Sweden), algharidus

Teenistuskäik

2007–	Technische Universität München, Wissenschaftlicher Mitarbeiter
2005–2007	Tartu Ülikool, programmeerija