PSL UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École normale supérieure

Vérification automatisée de code système à l'aide d'abstractions mémoire basées sur le typage

Automated Verification of Systems Code using Type-Based Memory Abstractions

Soutenue par
Olivier Nicole
Le 19 avril 2022

Ecole doctorale n° 386
Sciences Mathématiques de Paris centre

Spécialité
Informatique

Composition du jury :

Jean-Christophe Filliâtre Directeur de recherche, CNRS	*Président*
Alan Schmitt Directeur de recherche, Inria	*Rapporteur*
Mihaela Sighireanu Pr. des universités, ENS Paris-Saclay	*Rapportrice*
Timothy Bourke Chargé de recherche, Inria	*Examinateur*
Julia Lawall Directrice de recherche, Inria	*Examinatrice*
Matthieu Lemerre Ingénieur-chercheur, CEA List	*Co-directeur*
Xavier Rival Directeur de recherche, ENS	*Directeur de thèse*

ENS | PSL list ceatech

Automated Verification of Systems Code using Type-Based Memory Abstractions

Abstract

As software is an essential component of many embedded systems or online information systems, its malfunction can cause harm or security vulnerabilities. New bugs and vulnerabilities keep being discovered in existing software; many of those bugs and vulnerabilities are caused by violations of memory safety. In particular, low-level code, written in languages that offer few safety guarantees, is the most prone to this kind of bug. However, writing low-level code is sometimes necessary for performance or direct access to hardware features. Formal methods can be used to verify the safety of low-level programs, but automated analysis techniques to verify memory-related properties, such as shape analyses, still require important human effort, preventing wide adoption.

In this thesis, we propose a practical automated analysis based on types that express structural invariants on memory down to the byte level. This analysis, which we formalize in the framework of abstract interpretation, offers a trade-off between precise, flow-sensitive shape analyses and scalable, flow-insensitive pointer analyses. It can be applied to low-level code with only a small amount of manual annotations. We show how the type-based abstraction can be complemented with *retained* and *staged* points-to predicates to handle precisely common low-level code patterns, such as data structure initialization. We demonstrate the effectiveness and practicality of the analysis by verifying the preservation of structural invariants (implying spatial memory safety) on C and machine code programs, showing that it can be helpful in eliminating an entire class of security vulnerabilities.

We then apply our analysis to executables of embedded kernels and show that our type-based invariants allow to verify *absence of runtime errors* and *absence of privilege escalation.* To do this, we introduce the concept of implicit properties, i.e. properties which can be defined without reference to a specific program, and therefore lend themselves well to automated verification; and we prove that absence of privilege escalation is an implicit property. Parameterized verification, i.e. verification of the kernel independently from applicative code and data, poses many challenges, such as the need to summarize memory, or the dependence on a complex precondition on the initial state. We propose a methodology to solve them using our analysis technique. We apply this methodology to verify absence of runtime errors and absence of privilege escalation on a full, unmodified embedded kernel with a high level of automation.

Keywords: static analysis, memory analysis, OS verification

Antique
Paris

Vérification automatisée de code système à l'aide d'abstractions mémoire basées sur le typage

Résumé

Les logiciels étant des composants essentiels de nombreux systèmes embarqués et de nombreux systèmes d'information, un dysfonctionnement logiciel peut entraîner d'importants dommages ou des failles de sécurité. De nouveaux bugs et de nouvelles vulnérabilités sont trouvés régulièrement dans les programmes existants ; une grande partie d'entre eux est causeé par des violations de la sûreté mémoire. En particulier, le code bas niveau, écrit dans des langages de programmation qui offrent peu de garanties de sûreté, est le plus susceptible de contenir ce type de bug. Malgré cela, écrire dans un langage bas niveau reste parfois nécessaire pour des raisons de performance, ou pour accéder directement aux fonctionnalités du matériel. Les méthodes formelles peuvent permettre de vérifier la sûreté des programmes bas niveau, mais les techniques automatisées de vérification de propriétés mémoire, telles que les analyses de forme, nécessitent encore un effort manuel important, ce qui est un obstacle à une adoption large.

Dans cette thèse, nous proposons une analyse automatisée facilement applicable, basée sur un système de types exprimant des invariants structurels sur la mémoire, précis jusqu'au niveau de l'octet. Cette analyse, que nous formalisons dans le cadre de l'interprétation abstraite, offre un compromis entre les analyses de forme, précises et sensibles au flot de contrôle, et les analyses de pointeurs, qui sont insensibles au flot de contrôle mais passent très bien à l'échelle. Elle peut être appliquée à du code bas niveau avec peu d'annotations manuelles. Nous montrons comment cette analyse basée sur les types peut être complémentée par des prédicats de pointeurs *conservés* et *reportés*, afin de supporter précisément des motifs fréquents en code bas niveau tels que l'initialisation de structures de données. Nous démontrons l'efficacité et l'applicabilité de l'analyse en vérifiant la conservation d'invariants structurels (qui impliquent la sûreté mémoire spatiale) sur des programmes C et du code machine, montrant qu'elle peut être utile pour éliminer toute une classe de failles de sécurité.

Nous appliquons ensuite notre analyse à des exécutables de noyaux embarqués, et nous montrons que nos invariants à base de types permettent de vérifier *l'absence d'erreurs à l'exécution* et *l'absence d'escalade de privilèges*. Pour cela, nous introduisons le concept de propriété implicite, c'est-à-dire de propriété qui peut être définie sans référence à un programme en particulier, qui se prêtent bien à la vérification automatique ; et nous montrons que l'absence d'escalade de privilèges est une propriété implicite. La vérification paramétrée, c'est-à-dire la vérification de noyaux indépendamment du code et des données des applications, comporte plusieurs défis, comme le besoin de résumer la mémoire ou bien la dépendance à une précondition complexe sur l'état initial. Nous proposons une méthodologie pour les résoudre à l'aide de notre technique d'analyse. À l'aide de cette méthodologie, nous vérifions l'absence d'erreurs à l'exécution et l'absence d'escalade de privilèges sur un noyau entier sans modification, avec un haut niveau d'automatisation.

Mots clés : analyse statique, analyse mémoire, vérification d'OS

Contents

Chapter 1

Introduction

Software runs thousands of critical infrastructures. Yet, most of it potentially contains severe security flaws that are yet to be discovered. In 2014, a vulnerability in the OpenSSL cryptography software, today known as Heartbleed, was disclosed. It allowed an attacker to steal a web server's private keys, as well as users' passwords [Mit14]. About half a million servers were affected, representing about 17 % of the Internet's "secure" web servers. Only the prompt reaction of server administrators to upgrade their software limited the consequences of the security flaw.

As recently as 2021, two privilege-escalation vulnerabilities have been found in the Linux kernel [Mit21a] and in the `sudo` system utility program [Mit21b].

These three vulnerabilities have in common of exploiting *memory safety* violations. The precise definition of memory safety heavily depends on the programming language, and possibly even on the compiler [AHP18]. However, a reasonable and intuitive account of it can be given by saying that memory safety is the conjunction of two properties:

- *Spatial memory safety* specifies that every memory access is performed on a valid pointer to an object, and into the bounds of the referenced object;
- *Temporal memory safety* specifies that an object is no longer accessed after it has been deleted (freed), and in some languages like C, that only a valid pointer to an object should be freed.

1.1 The need for formal verification of low-level code

In this thesis, we call "low-level" programs that utilize the capabilities of the hardware architecture, such as memory or system instructions, without safety checks, often for performance reasons.

According to the 2020 ranking of the MITRE corporation [Mit20], memory corruption is the second most dangerous attack vector. In 2019, Microsoft reported that 70 % of vulnerabilities reported to them had been caused by a memory safety bug [Mil19], a proportion that has been stable from 2006 to 2018. Such bugs can only appear in languages that do not enforce memory safety, such as C or assembly code.

Despite its lack of safety, C has been extremely popular for decades and remains the most used language for writing performance-critical code, or code that performs low-level operations. This historical success is probably due to its relative simplicity, its wide portability across hardware platforms, and the control it provides on hardware resources.

One approach to avoid many software vulnerabilities would be to write programs in so-called high-level languages, which offer more safety guarantees (such as memory safety), and then interpret them or compile them to lower-level languages for execution. In recent years, the rise in popularity of the Rust language [KN19], a language with a memory-safe and thread-safe subset with a fine-grained control over resources similar to C, shows a trend in that direction.

However, Rust programmers are often forced to use the `unsafe` keyword, which steps out from the safe subset of the language [ECS20]. In addition, this approach does not account for the billions of lines of C that are still in active use, and probably contain many undiscovered vulnerabilities. What's more, foundational programs such as OS kernels will in any case need to use intrinsically unsafe assembly languages, and kernels are the most critical software components of most information systems. Methods to verify the correctness of low-level code are therefore needed.

1.2 The need for automated analyses

Several techniques exist to verify the correctness of programs. The first task consists in precisely defining the *specification*, that is, the set of acceptable behaviors for the program. Then, one must prove with a high level of confidence that the program respects its specification. However, we know by Rice's theorem that all non-trivial, semantic properties of programs are undecidable [Ric53]. In other words, it is impossible to create a program analysis with *all* following properties:

- **Sound.** If the analysis decides that the program complies with the specification, it is indeed the case.
- **Complete.** If the analysis decides that the program does not comply with the specification, it is indeed the case: the program exhibits at least one behavior defined as unacceptable.
- **Automated.** The analysis is a standalone algorithm whose sole input is the program to verify, and does not require further human intervention.
- **Generic.** The analysis should be applicable to arbitrary programs.

Designing a verification method requires to sacrifice at least one of those four properties. Each choice results in a different technique:

- *Testing*, i.e. running the program on a wide variety of inputs, can help to eliminate a large number of bugs. However, it is not a *sound* method: the range of possible entries is usually too vast to perform exhaustive testing, and therefore testing does not allow to prove the absence of bugs.
- Giving up completeness leads to analyses that may fail to verify the desired specification even on programs that respect it; a situation known as false alarms. *Abstract interpretation* [CC77] is a theoretical frameworks for constructing sound, automated program analyses.
- *Assisted program proof* consists in encoding the semantics of the programming language into the language of a proof assistant or an SMT solver, and then using these tools to verify the properties of interest on the program. However, this process is not automated in general: it may require human input, typically to specify loop invariants, or to assist the proving tool.
- Finally, one can opt to give up the genericity of verification. When program states describe a finite space, *model checking* [CE82; QS82] can allow to verify arbitrary properties, with various techniques to deal with combinatorial explosion.

While model checking is relevant in specific domains, we are interested in this thesis in a verification method for low-level code that is widely applicable, which mandates genericity. Similarly, while program proof is extremely powerful, it can be very costly in manual work: proving the correctness of the seL4 kernel with respect to a functional specification required about 200,000 lines of proof script [Kle+09]. In addition, verifying low-level code requires expertise in both low-level computing and program proof, which deters wide applicability.

In this thesis, we will use the abstract interpretation framework to design generic, automated and sound analyses to verify safety properties on low-level programs, such as unrestricted C code, assembly code or even binary programs.

However, analyzing low-level code is particularly challenging. It often has a dynamic control flow —due to computed jumps in assembly, or function pointers in C— whose resolution requires an analysis of possible runtime values, and contains low-level memory manipulations such as unrestricted pointer arithmetic, which can lead to accessing arbitrary addresses in memory. To make matters worse, in binary code, there is no *a priori* separation between code and data, or between different values on the stack; therefore a small imprecision on a memory address can prevent the analysis from determining the effect of reading or writing at that address.

For all these reasons, low-level code analysis is prone to imprecisions, which can easily snowball up to a point where the analysis cannot prove any property.

1.3 The case for a type-based memory abstraction

Among the hard points mentioned above, low-level memory manipulation is arguably the hardest.

There is no type system in assembly or binary, and in C, typechecking does not guarantee memory safety, due to unchecked array accesses, type casts or pointer arithmetic.

Like many languages, C allows to manipulate data structures of unbounded size —i.e. their size cannot be determined by a static analysis, like when traversing a linked list whose size is a parameter, for instance. These data structures may exhibit complex sharing patterns, i.e. the graph of points-to relations between values can be arbitrary. And unlike in higher-level languages, the semantics of C and assembly make it very easy to cause crashes or undefined behaviors when manipulating such structures. In other words, even the *memory safety* of programs depends on intricate memory properties. The *analysis of memory* is therefore a crucial point of analyses of low-level code.

On the lower end of the precision spectrum, *pointer analyses*, pioneered by Andersen [And94] and Steensgaard [Ste96], consist in abstracting pointer values by the set of program variables they may point to. Such analyses are very efficient and therefore widely implemented for optimization purposes. However, they are unable to infer any information about the organization of data structures in memory, beyond properties of pointer variables and expressions in the program. This makes them often unable to prove even base properties like memory safety.

On the other end of the spectrum are *shape analyses*, able to infer complex properties about data structures in memory, including separation or connectivity properties between memory regions and well-formedness of data structures (such as doubly-linked lists or red-black trees), including unbounded data structures. Such unbounded structures are represented in a summarized form in the analysis's internal state, and the analysis procedures are able to *refine* that summarized form when needed to account precisely for memory locations that are read or written. Conversely, they are also able to *generalize* a set of memory states into a summarized form, which is necessary in order to ensure termination of the analysis. Such analyses include abstract domains based on separation logic [DOY06; CR08] and analyses based on three-valued logic [SRW99]. Shape analyses have been used successfully to prove memory safety, as well as the preservation of structural invariants, in various data structure libraries in C or Java [Li+17; DPV11; Hab+12; Cal+11].

On the negative side, shape analyses have yet to scale to large codebases. Possible reasons are the necessity to record *disjunctions* of possible states in the analysis, which can make the analysis state grow exponentially and hinders scalability; or the fact that using shape analysis requires in-depth knowledge of static analysis to encode data structure invariants into their annotation language, to understand the origin of false alarms, and to provides hints to the analysis tool in order to eliminate those alarms.

However, useful properties can be proved by verifying simpler invariants than those abstracted by shape analyses. In this thesis, we propose an analysis that occupies a new spot on the precision spectrum, between pointer analyses and existing shape analyses. Instead of requiring that all objects be separated in memory, which requires fine-grained tracking, our approach only verifies that objects of different types are separated, allowing for efficient verification of the preservation of expressive typing invariants —which implies, notably, spatial memory safety— without having to specify sharing patterns.

To achieve such a verification on low-level programs, we define a type system, which we call *physical types*, that expresses the layout of values in memory down to the byte level. It has some common traits with Chandra and Reps's physical type checking algorithm [CR99], that our analysis generalizes. This approach allows to re-use the types used in the C code, improving automation.

1.4 Overview of the method and illustration on an operating system kernel

We will illustrate our method on the case of OS kernels verification, for which it is a good fit as we explain below.

1.4.1 Motivation

Operating systems, and more specifically *operating system kernels*, hold a particular position in the software world. They are the cornerstone of the security of most computer systems, making it particularly critical for them to be devoid of bugs.

Manually proving a kernel using a proof assistant or deductive verification can ensure that the kernel complies with a formal specification, thus reaching a high level of safety and security. But such a verification is out of reach of most actors, particularly in *embedded systems* development, because of the tremendous proof efforts involved (e.g. 200,000 lines of proof script for seL4 [Kle+09] or 100,000 for CertiKOS [Gu+16]) and of the difficulty to find experts in both systems and formal methods. For such actors, the ideal method would be an automated verification where the system developers only provide their code and, with very little configuration or none at all, run the tool to verify the properties of interest. Automation is all the more necessary for embedded kernels that are typically produced in many variants, depending on the target hardware and the applications to run.

Automated verification of complex, high-level specifications of kernels is currently out of reach of verification techniques. But two fundamental properties can be verified with a high level of automation using our analysis:

1. The absence of *crashes*, such as those caused by divisions by zero, illegal opcodes or illegal memory access; we call this property the *absence of runtime errors (ARTE).*

2. The *absence of privilege escalation (APE).* Privilege escalation occurs when a malicious application bypasses kernel protections and obtains kernel privileges. In the case of hypervisors, this corresponds to virtual machine escape. In fact, no safety or security property can be proved on the kernel unless APE holds.

Both properties will be defined more precisely in Chapter 8 where we detail the application of our analysis method to OS kernels.

In addition, kernel code is very low-level and heterogeneous: it is a mix of low-level C, assembly code and linker scripts (in the case of embedded kernels). To perform a comprehensive verification, a solution is to analyze the kernel *executable.*

Finally, as we illustrate below, analyzing kernels involves to deal with complex memory layouts, and in particular with unbounded structures containing the characteristics and data of applications.

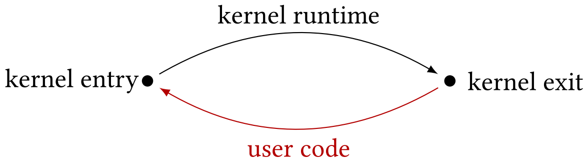


Figure 1.1: Alternated execution of kernel and user code.

```
Task *cur; Context *ctx;

runtime() {
  save_context();
  /* Schedule next task */
  cur = cur->next;
  ctx = &cur->ctx;
  load_protection();
  load_context();
}
```

```
typedef struct { Int8 pc, sp, flags; } Context;

typedef struct Task {
  Memory_Table *mt;
  Context ctx;
  struct Task *next;
} Task;
```

Figure 1.2: Code of a simple embedded OS kernel and associated type of task data.

1.4.2 Kernel description

In essence, a kernel is a program whose execution alternates with the execution of the application code, and sets up the appropriate *hardware protections* before giving hand to applicative code again (Figure 1.1). We shall term *user code* the code of applications in general. The *kernel runtime* starts executing whenever an *interrupt* occurs, either a software interrupt (e.g. an application performed a system call) or a hardware interrupt (e.g. due to a timer firing, or an illegal memory access).

In the context of kernel verification, the user code is unknown.

Consider the extremely simplified kernel whose code is in Figure 1.2. Each task is assigned a *memory table* describing the memory region it should have access to, a context in which program counter (`pc`), stack pointer (`sp`) and hardware-related flags (`flags`) are stored. In addition, each `Task` structure contains a pointer to the next task to be scheduled, the whole forming a circular list.

Every time the kernel runtime executes, it saves the context of the current task, switches to the next task, and configures memory protection appropriately for that task, by setting a special register to instruct the Memory Protection Unit (MPU) to enforce the access policy described in the task's memory table. `Memory_Table` structures (not detailed here) describe intervals of authorized addresses.

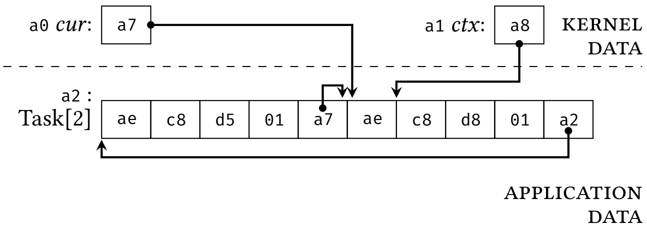


Figure 1.3: Example state of the memory reachable by the kernel.

Memory layout and parameterization Let us now look at an example memory layout of this system (Figure 1.3). For simplicity, we represent it as it would be on a fictitious 8-bit hardware. It represents a system composed of two tasks sharing the same memory table. However, the kernel is independent from the tasks: the same kernel code could manage a system composed of thousands of tasks, each with different memory rights. We say that the kernel is *parameterized* by the application data. A consequence of this parameterization is that the addresses of many system objects (e.g. of type `Task` or `Memory_Table`) vary and are not statically known in the kernel.

1.4.3 Verification method

The static analysis will compute an over-approximation of the possible system states at each program location. A *state* of the system consists in the value of each variable, register, and the content of each memory cell.

For instance, let us suppose that, at the beginning of the function `runtime()`, the variable `cur` contains either the value `0xa2` or `0xa7`. Then it is quite straightforward to verify that the two dereferences of `cur` at the second and third line of the function are safe. The expression `cur->next` dereferences one of the values `0xa6` and `0xab`, which are both valid memory address. The result is either `0xa7` or `0xa2`, therefore the instruction `cur = cur->next` leaves the set of possible values for `cur`, namely {`0xa2`, `0xa7`}, unchanged. From that we deduce that the dereference `cur->ctx` that follows is also safe, since the dereferenced address is either `0xa3` or `0xa8`.

More generally, the absence of runtime errors is a *state property*: it can be proved by finding a state invariant on the system. We show in Chapter 8 that the absence of privilege escalation is also a state property, and therefore can be verified by the same technique.

In order to compute such an invariant, a static analysis requires an abstraction of memory. A very simple abstraction would be to use a *flat model*: to view memory as an array of bytes, and represent memory addresses by their numeric values. Abstract memory states computed by the analysis would then resemble Figure 1.3. This approach has been taken by other works in kernel verification [Dam+13; Nel+19; Nor20]. However, it prevents *parameterized* verification: the kernel cannot be analyzed independently from the tasks. In addition, it poses scalability issues: when analyzing a system with 1,000 tasks, all corresponding structures have to be enumerated. It is also fragile: tiny imprecisions on the numeric addresses can cause huge precision loss. This would happen for instance if, in our example kernel, the set of possible values for `cur`, {`0xa2`, `0xa7`}, was over-approximated by the interval [`0xa2`, `0xa7`].

Lifting those limitations requires an analysis capable of abstracting the numeric values of addresses, and manage structures of unbounded size, while conserving the structure of objects and the links between them. The abstract domains presented in this thesis meet these requirements. We show in Part II that it is able to represent data structures with sufficient precision to prove ARTE and APE on real OS kernels automatically, with a very low annotation burden.

Our method could be summarized by the following steps:

1. Define a set of *physical types* that describe the structure of task data.

2. When analyzing the kernel runtime code, abstract values by both a numeric component and a physical type. Our analysis both *infers* types of values and *verifies* that all operations are well typed. As we show, this is sufficient to verify spatial memory safety.

3. Our abstraction is also able to express numerical properties on values within data structures, which can be used to eliminate non memory-related runtime errors (such as divisions by zero), and to verify that the kernel sets up memory protections appropriately, thus verifying absence of privilege escalation. For example, the types should express the fact that the `Memory_Table` of each task gives access to a memory region disjoint from kernel memory.

1.5 Contributions and outline of the thesis

The objective of the present thesis is to define an efficient memory analysis to verify structural invariants expressed as type constraints, and to demonstrate the interest of such invariants. We evaluate the effectiveness and efficiency of the analysis by verifying spatial memory safety on a set of low-level C and binary programs, and the interest of type-based invariants by verifying ARTE and APE on embedded OS kernels as exemplified in Section 1.4.

We make the following contributions:

- We define a new abstraction able to summarize memory by means of a type invariant, as well as the algorithms to use that abstraction in a program analysis, formalized in the framework of abstract interpretation (Chapters 3 to 5).
- Given the insufficient precision of this analysis on some ubiquitous programming patterns, we introduce two independent abstractions (Chapter 6) to make the precision suitable for proving the properties of interest: the abstract domain of *retained points-to predicates* refines information about the memory regions most recently accessed, allowing strong updates without introducing disjunctions; the abstract domain of *staged points-to predicates* allows the analysis of programs that temporarily violate typing invariants, which is crucial for precision, in particular on programs that perform initialization of data structures.
- Based on these abstract domains, we construct two analysis tools: one working on C source code, the other on binary executables. We use them to evaluate the applicability of our abstractions to verify spatial memory safety and basic structural invariants on a set of benchmark programs (Chapter 7).
- We formalize the notions of absence of runtime errors and absence of privilege escalation in OS kernels (Chapter 8), and use our binary analysis to prove them on embedded OS kernels, including a kernel used in the industry (Chapters 9 and 10). Finally, we review the existing works on kernel verification (Chapter 11) and discuss the positioning of this work relative to them.

The work presented in this thesis was published in two articles:

- The work presented in Chapters 4 to 7 has been published in a summarized version in the following paper: Olivier Nicole, Matthieu Lemerre, and Xavier Rival. “Lightweight Shape Analysis Based on Physical Types”. In: *23rd International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI ’22)*. 2022.
- The work presented in Chapters 8 to 11 has been published in the following paper: Olivier Nicole, Matthieu Lemerre, Sébastien Bardin, and Xavier Rival. “No Crash, No Exploit: Automated Verification of Embedded Kernels”. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS ’21). 2021. This paper was attributed the *RTAS 2021 Best Paper Award.*

Part I

Type-based Shape Analysis

Chapter 2

Related work on static analysis of memory

Outline of the current chapter

Chapter 1 gave a brief account of pointer and shape analyses and our positioning between them. This chapter aims at giving a more comprehensive picture of static analyses that are concerned with memory properties. We first describe pointer analyses (Section 2.1) and shape analyses (Section 2.2), which constitute the two opposite ends of the expressiveness spectrum. We then survey a third line of work: memory analyses based on type systems, to which our work is related (Section 2.3). We present here a survey of existing works; a comparison with our work can be found later in Section 5.6.

This chapter is focused on memory analysis, but a survey of operating system verification can be found in Chapter 11.

2.1 Pointer analysis

Pointer analyses attempt to determine properties of pointer values. The term *alias analysis* is also often used in similar contexts; although the terminology is not entirely unified across literature, pointer analysis usually seeks to determine to which objects a pointer variable may point, whereas alias analysis focuses on the question of whether two pointers may be equal, or point to overlapping objects (*alias*). An alias analysis can be derived from the result of a pointer analysis.

Unlike shape analyses, pointer analyses do not attempt to infer properties of memory objects that are not directly pointed to by variables of the program, or —for some analyses— by expressions in the program. Therefore, while they are useful to compute aliasing relations, they are unable to infer many properties, such as validity of pointers, reachability, or preservation of structural invariants, especially when the data structures involved are of unbounded size.

However, the simplicity of the static heap representation makes pointer analyses much more tractable than shape analyses in terms of complexity, allowing most of them to be applied as whole-program analyses to millions of lines of code.

2.1.1 Original works

The best-known pointer analysis technique has been introduced by Andersen [And94]. This analysis computes *points-to sets* for all pointer variables, i.e. sets of memory objects to which the variable may point. Points-to sets are inferred in a flow-insensitive way. To do that, it infers subset constraints from statements of the program, then runs a constraint solving algorithm to obtain a superset of the points-to set for each variable. The whole process runs in polynomial time in the size of the program.

As in all pointer analyses, the computed points-to sets are in fact sets of *abstract memory objects*. Indeed, since the number of memory objects referenced by a pointer during program execution is not bounded *a priori*, some form of abstraction is necessary for the analysis to terminate. In case of Andersen's analysis, memory objects are abstracted by their allocation site, i.e., labels of instructions where the object is allocated.

Steensgaard [Ste96] computes similar points-to sets, but using a unification-based approach: every assignment to a pointer value gives lieu to the unification of two points-to sets. Like in Andersen's analysis, the computed points-to sets are flow-insensitive. This approach is strictly less precise than Andersen's, but permits better performance, as it can be implemented using union-find algorithms to run in almost-linear time. However, Andersen-style analyses are now more popular, because their higher precision is considered worth the slight performance cost [SB15]. However, we can also mention that Das [Das00] proposes a middle ground between the two algorithms, by representing directional points-to constraints *à la* Andersen between memory objects that are directly pointed by program variables, but unifies points-to sets *à la* Steensgard for objects further down the points-to chains. By this trade-off, Das achieves a worst-case quadratic time complexity in the size of the program.

2.1.2 Enhancing precision with richer pointer abstractions

In order to improve the precision of pointer analyses without too much degradation of performance, *context-sensitivity* has emerged as an effective variation of Andersen-style analyses. Context-sensitive analyses add *context* information to local variables and abstract memory objects, and the analysis merges information over all executions that result in the same context, but keeps information separate for different contexts. The definition of "context" depends on the analysis flavour. Two main kinds of context-sensitivity have been explored: *call-site sensitivity* and *object sensitivity*.

Call-site sensitivity qualifies memory objects with a sequence of call sites (i.e., labels of instructions where a function is called) corresponding to the call stack at the time of their allocation. Points-to information is then merged or kept separate depending on the current call stack.

Object sensitivity only applies to object-oriented languages. It uses as context the allocation site of the object on which the current method is called (in object-oriented languages such as Java, all function calls are method calls), or possibly sequences of allocation sites, corresponding to allocation site of the receiver object O of the current method, the allocation site of the allocator object (O′) of O, etc.

As an object-sensitive analysis can sometimes suffer from combinatorial explosion, Smaragdakis, Bravenboer, and Lhoták [SBL11] propose *type-sensitive analysis* as a similar but more scalable approach.

Type sensitivity is identical to object sensitivity, except that types are used as context elements instead of allocation sites.

2.1.3 Algorithmic improvements over Andersen-style analyses

Andersen's analysis can be reformulated as the computation of the transitive closure of a points-to graph. For this reason, cycle detection methods can be used to optimize them, because all nodes in the same cycle have identical points-to sets and can be collapsed together. In particular, Hardekopf and Lin [HL07] introduce hybrid cycle detection (HCD), a method that partitions a cycle detection algorithm into an off-line and on-line analysis, and show that it improves the performance of cycle detection algorithms. Applying HCD to the other algorithm that they introduce, namely lazy cycle detection, improves significantly over the state of the art in terms of analysis time.

All analyses mentioned up to now are flow-insensitive, meaning that the computed points-to relations hold globally, independently of the program point. Flow-sensitive analyses are generally more precise than flow-insensitive ones, but are also less scalable. In a later paper, Hardekopf and Lin [HL11] note that flow-sensitive analyses can be made more scalable by running them on a so-called "sparse" representation of the program, such as the static single assignment (SSA) form [Cyt+91], which relates variables to their uses. However, constructing this sparse representation requires the very points-to information that the analysis is meant to compute. Hardekopf and Lin propose a staged analysis, that first computes this "def-use" information conservatively, by running a flow-insensitive (and therefore fast) pointer analysis—which they call the "auxiliary" analysis— enabling a flow-sensitive analysis to be performed sparsely. Using Hardekopf and Lin [HL07] as their auxiliary flow-insensitive analysis, they are able to perform a flow-sensitive, Andersen-style analysis of a program of 1.9 million lines of code in 14 minutes.

Other directions to improve scalability include *modular* [WL95; CH00; CRL99] and *demand-driven* [SXX12] analyses. Modular analyses compute for each function a *summary* of its effect, which can be used in the rest of the analysis when these functions are called, instead of re-analyzing the function for every different context. Cheng and Hwu [CH00] propose function summaries consisting in sets of pairs of access paths, representing possible aliasing relations created by the function. Demand-driven pointer analyses compute only the results necessary to answer a query at a given program location. Shang, Xie, and Xue [SXX12] combine demand-driven analysis with function summaries to answer to queries in a context-sensitive way.

2.2 Shape analysis

Unlike pointer analysis, shape analysis techniques attempt to retain information about the heap structures manipulated by the program, notably on the links between heap objects. This enables shape analysis methods to verify properties stronger than mere memory safety or aliasing relations, such as the *reachability* of some memory cells, or the preservation of some *data structure invariants* (e.g. the invariants on pointers and values that characterize a doubly-linked list or a red-black tree).

In order for the analysis to terminate, or simply to prevent the static heap representation from growing too large, shape analyses must be able to *summarize* regions of arbitrary size in a finite way. In addition to summarization, in order to precisely take into account the effect of memory writes, shape analyses are able to temporarily refine a summary into a more precise representation, a process often called *materialization* or *focus*. Conversely, it must be able to turn a non-summarized heap representation into a summary, possibly at the cost of some precision. To sum up, shape analyses are able to represent precise properties of data structures in a summarized way, and are able to go back and forth between summarized and non-summarized representations.

2.2.1 Shape analysis based on three-valued logic

Shape analysis based on three-valued logic [SRW99], and the associated TVLA tool, view each heap object as a node. Object fields, as well as variables, are represented using logical predicates called the *core predicates*. Besides core predicates, TVLA also allows the user to define a set of *instrumentation predicates*, defined in a language of first-order logic with transitive closure. Instrumentation predicates can be simple points-to predicates, or global ones, specifying reachability or acyclicity properties. These predicates give a complete picture of the information stored in a TVLA heap: a set of nodes and the predicates that hold among them. The predicate facts are simply a conjunction of atomic literals, which have the form $\mathrm{P}(n_1, n_2, \ldots)$ or $\neg\mathrm{P}(n_1, n_2, \ldots)$. The combination of nodes and predicate information is called a *structure* in TVLA.

Abstraction is introduced with the concepts of *summary node*, which represent one or more concrete nodes, and by allowing predicates to take the value ½, meaning "don't know", in addition to "true" and "false". A notion of *embedding* (denoted $\sqsubseteq$) can be derived: $\mathrm{S}_1 \sqsubseteq \mathrm{S}_2$ expresses that S_1 is more precise than S_2 in the sense that its predicates refine the predicates of S_2. The concretization of a structure S can be defined as:

$$\gamma(\mathrm{S}) = \{\mathrm{S}_0 \ ; \ \mathrm{S}_0 \text{ is concrete} \wedge \mathrm{S}_0 \sqsubseteq \mathrm{S}\}$$

However, the embedding relation $\sqsubseteq$ does not allow to define a *join* operation between any two structures. To perform program analysis, some form of join is necessary; therefore, the abstract memory representation in TVLA is a *finite disjunction* of structures. To ensure that the number of disjunctions remains bounded, canonicality constraints are enforced on the structures at selected program points (including loop heads), weakening some structures (i.e. losing precision) if necessary. The shape analysis can then be performed by abstract interpretation (see Chapter 3 for a definition of the abstract interpretation framework).

The TVLA tool can express and verify an extremely rich range of properties. It has been used in practice to verify memory safety and other higher-level properties of libraries [Dor+08]. It can be, however, expensive in terms of resources [Dor+08; McC10]. This may be due to the global nature of TVLA predicates, which can depend on nodes arbitrarily far away from one another.

On the other hand, the heap representation of TVLA allows it to natively handle structures with unstructured sharing, which are challenging for separation logic analyses.

2.2.2 Shape analysis based on separation logic

This family of shape analyses is based on the fact that reasoning over memory operations is much easier when the modified memory cells can be precisely localized. This led to favor local reasoning by using *separation logic* [Rey02; DOY06; CR08]. A separation logic formula describes a set of memory states using *points-to predicates* and *separating conjunction*. Unbounded regions can be summarized by *inductive predicates* [CR08].

In order to precisely calculate the effect of memory writes, the analysis can refine the representation of a summarized region. This operation is known as the *unfolding* of a summary predicate. Conversely, in order for the analysis to terminate, or simply to prevent the abstract representation from growing too large, a memory region must sometimes be *folded* into an inductive predicate.

Analyses based on separation logic and inductive predicates have been used to verify complex memory properties such as preservation of data structure invariants using the Xisa [CR08], Predator [DPV11; DPV13] and MemCAD [Li+17] tools. Xisa and MemCAD must be configured with user-provided inductive predicates [CR08] that summarize linked data structures, such as linked lists or trees. The Predator tool is more automated and requires no such configuration, however its scope is more limited as it only handles various forms of singly- and doubly-linked lists (possibly nested or cyclic, with data pointers and some sentinel pointers), but not trees or direct acyclic graphs (DAGs) [DPV11]. Predator has been augmented to support low-level C programming patterns such as unrestricted pointer arithmetic, pointers to a non-zero offset in a structure, manipulation of pointers to invalid targets, or block memory operations [DPV13].

However, heap representation by separation logic makes it impossible to analyze structures with "unstructured sharing" using the standard algorithms. An example of such a structure would be the representation of a mathematical graph by adjacency lists, or a linked list containing pointers to the nodes of a binary tree. Li et al. [LRC15] introduce set-valued inductive predicates in order to precisely represent such data with "unstructured" sharing, and devise an analysis able to prove memory safety and preservation of data structure invariants on a graph library.

Separation-logic-based analyses also need to introduce disjunctions of abstract states when unfolding an inductive predicate, which can make the abstract state size increase exponentially. Sophisticated canonicalization-based heuristics [DOY06] or semantic-based methods [Li+17] are required to prevent that combinatorial explosion.

Another solution to scale up is to design a *compositional* shape analysis, that infers pre- and postconditions for each procedure. This has been realized in Facebook Infer [Cal+11], which makes use of bi-abduction techniques to infer the missing pre-conditions on a code fragment to prove its memory safety. This allows to analyze procedures out of context and infer several procedure summaries (i.e. precondition–postcondition pairs) for each procedure; these summaries can be reused when the procedure is called. This allows the tool to scale up impressively, as it is able to analyze a version of the Linux kernel (3 million lines of code) in less than 3 hours, and infer correct pre- and postconditions for 60 % of the procedures (although it does not suffice to verify the memory safety of these procedures, because they are not guaranteed to be called with the inferred preconditions).

Illous, Lemerre, and Rival [ILR20] take another approach to modular analysis and represent function summaries as *abstract transformations* rather than precondition–postcondition pairs. They introduce novel logical connectors over abstract transformations, inspired by separation logic, which enable them to devise a static analysis by abstract interpretation based on those connectors. In addition, to make the analysis modular, they introduce an algorithm to compose abstract transformation, thus enabling to analyze each function only once.

2.2.3 Other shape analysis techniques

Based on the work of Jonkers [Jon81] which characterizes data structures by their access paths and the aliasing relations between them, Deutsch [Deu92; Deu94] proposes a storeless analysis by abstract interpretation whose lattice consists of alias pairs of the form $((p_1, p_2), \mathrm{K})$ where p_1 and p_2 are *symbolic access paths* like $\texttt{X}{\to}(\texttt{tl}{\to})^i\texttt{hd}$ (for a linked list structure `struct l { void *hd; struct l *tl; }`), and K is an element of a numerical abstract domain (such as intervals, linear equalities, etc.) accounting for the free variables in p_1 and p_2 (like i in our example). This way, Deutsch is able to express properties such as "the elements of X and Y are pairwise aliased" or "the i-th element of Y is aliased with the $2i+1$-th element of X". This analysis has the potential to express precise properties even on dynamic and unbounded data structures, although representing data structures by alias pairs may look unintuitive compared to shape graphs [SRW99]. Existing applications of symbolic access paths are limited to type-safe languages [Deu92] or type-safe C programs [Deu94].

Ghiya and Hendren [GH96] proposed a storeless abstraction that retains the shape of the region pointed by every pointer variable, among a limited set of data structure "shapes": tree, acyclic graph or graph with cycles. This shape abstraction is propagated in combination with a record of may-alias relations between pointer variables. Because the heap is not represented deeper than by attributing a shape to variables, this analysis is very conservative in presence of destructive updates.

The Forester tool [Hab+12; Hol+13] also uses a graph-based representation of the heap, but represents possibly unbounded regions using *tree automata*. Indeed, such automata can faithfully abstract memory structures that are shaped like trees, i.e. regions that do not feature sharing. When sharing is present, the heap is split in several sharing-free regions, with "cut points" to represent the links between these regions. Such automata can be recursive —hence the name *forest automata*—, allowing to describe structures such as lists of lists. While forest automata originally had to be provided by the analysis user, [Hol+13] adds the possibility for the tool to infer them automatically.

Marron et al's Unified Memory Analysis (UMA) [Mar+07] is a shape analysis for a subset of the Java language. It represents the heap by a directed graph whose nodes represent sets of heap objects. It is able to infer various properties, including connectivity between regions and the shape of regions, by marking graph nodes with shape labels in a manner similar to Ghiya and Hendren [GH96] (they add singly-linked lists to the set of possible shapes); having a richer heap abstraction allows them to deal with destructive updates less conservatively. Their analysis uses both summarization and refinement operations; the refinement does not introduce disjunctions, which avoids the scalability issues linked to disjunctions in shape analysis. However, refining a memory region is only possible for some data structures, and if there is exactly one incoming points-to edge into the region.

2.3 Type-based memory analyses

A third line of work, to which this thesis belongs, explores the possibility of leveraging type systems to infer or verify memory properties.

A restriction of refinement types [FP91] called *liquid types* has been introduced in high-level languages such as Haskell. Rondon, Kawaguchi, and Jhala [RKJ10] have proposed methods to use liquid types in low-level code as well. Using liquid types in function signatures is a way to specify function contracts. They are mostly used to verify memory safety.

Rondon et al.'s analysis algorithm has focusing and unfocusing operations, which allows it to perform flow-sensitive, strong updates on a single type instance at a time. The insertion of "fold" and "unfold" directives is conservative, which causes precision loss. This limitation is inherent to the fact that their analysis is primarily designed as a type inference algorithm which cannot make the focusing decision based on the inferred possible states of variables and memory.

Chandra et Reps's physical types for C [CR99], like ours, describe the byte-level layout of types in memory. For this reason, we reuse the name *physical types* for our type system (Chapter 4). Chandra and Reps's physical types support a restricted form of pointer arithmetic, namely expressions of the form `p + 1` where `p` is a pointer. Their analysis aims at verifying the correctness of programs manipulating standard C types and consists in a type inference algorithm.

Diwan et al.'s type-based alias analysis [DMM98] is an alias analysis which works on type-safe programs, implemented for Modula-3 programs. It consists in performing a Steensgaard-style points-to analysis, but using types instead of allocation sites as abstraction of memory objects. In addition, points-to sets are refined using subtyping relations: if `S1` is a subtype of `T`, then an expression of type `T` may refer to an object of `S1`, but the converse is not true.

Balatsouras and Smaragdakis [BS16] propose a *structure-sensitive* analysis for C and C++, which adds type information to abstract memory objects (in addition to allocation site information) to increase precision. Since type information is not always readily available—e.g. a call to `malloc()` returns a pointer to an array of bytes, which is cast to the correct type at a later point—they record the cast instructions these objects may flow to using a simpler points-to analysis. They show that distinguishing abstract memory objects by type improves precision significantly, and is crucial in resolving virtual calls in the case of C++.

Lattner [Lat05] proposed Data Structure Analysis (DSA), which takes as input LLVM code. It makes use of LLVM's type system to infer for each function a flow-insensitive points-to graph using a unification-based algorithm. Unlike other analyses, DSA infers possible points-to relations between abstract objects, including objects not directly pointed to by program variables. Connected instances of the same type are collapsed together. As a consequence, it can e.g. verify that two linked lists are disjoint, but not that they are acyclic. In addition, since it relies on LLVM's typed access paths, it cannot handle arbitrary pointer arithmetic. Despite being fully context-sensitive (each function call is analyzed in-context), it remains scalable enough to analyze hundreds of thousands of lines of C in a matter of seconds. The data structure graphs computed can be used to perform a pointer or alias analysis.

Finally, safer dialects of C have been proposed, such as Cyclone [Mor+02], CCured [Nec+05], or CheckedC [Ell+18]. Compilers for these languages use the programmer's annotations to check as many memory accesses as possible using a type inference algorithm, and inserts runtime checks when that is not possible. These tools focus on practicality and easy interfacing with unchecked code.

Chapter 3

Abstract interpretation framework

Outline of the current chapter

In this chapter, we define While-memory, a simple Turing-complete language with memory reading and writing. Then, after a brief reminder of the abstract interpretation results that we use, we lay out the general definition scheme of an abstract domain and analysis on this language. The static analyses defined throughout Part I take While-memory as input. In Chapter 7, we will detail how to port this analysis to C code and binary executables.

3.1 General mathematical notations

Given two integers $a, b \in \mathbb{Z}$, we define the following integer interval notations:

$$\begin{aligned}
[a, b] &\text{ means } \{x \in \mathbb{Z} \mid a \leq x \leq b\}\\
]a, b] &\text{ means } \{x \in \mathbb{Z} \mid a < x \leq b\}\\
[a, b[&\text{ means } \{x \in \mathbb{Z} \mid a \leq x < b\}\\
]a, b[&\text{ means } \{x \in \mathbb{Z} \mid a < x < b\}.
\end{aligned}$$

We let $\mathrm{X} \rightarrow \mathrm{Y}$ denote the set of total functions from X to Y, and $\mathrm{X} \rightharpoonup \mathrm{Y}$ the set of partial functions. Given a partial function $f : \mathrm{X} \rightharpoonup \mathrm{Y}$, we let $\mathrm{dom}(f)$ denote the domain of f and $\mathrm{codom}(f)$ its codomain,

$$
\begin{array}{rcll}
\mathit{stmt} & ::= & x := \mathit{expr} & (\mathit{assignment},\ x \in \mathbb{X}) \\
 & \mid & *_{\ell}\mathit{expr} := \mathit{expr} & (\mathit{memory\ write\ of\ size}\ \ell \in \mathbb{N}) \\
 & \mid & \mathit{stmt};\mathit{stmt} & (\mathit{sequence}) \\
 & \mid & \mathbf{skip} & (\mathit{no\text{-}op}) \\
 & \mid & \mathbf{while}\ \mathit{expr}\ \mathbf{do}\ \mathit{stmt}\ \mathbf{done} & (\mathit{loop}) \\
 & \mid & \mathbf{if}\ \mathit{expr}\ \mathbf{then}\ \mathit{stmt}\ \mathbf{else}\ \mathit{stmt}\ \mathbf{end} & (\mathit{conditional}) \\
 & \mid & x := \mathbf{malloc}(\mathit{expr}) & (\mathit{memory\ allocation},\ x \in \mathbb{X})
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{expr} & ::= & c & (\mathit{constant},\ c \in \mathbb{V}) \\
 & \mid & x & (\mathit{variable},\ x \in \mathbb{X}) \\
 & \mid & \mathit{expr} \diamond \mathit{expr} & (\mathit{binary\ op.},\ \diamond \in \{+,-,\times,/,\leq,<,=,\neq,\&,|,\cdots\}) \\
 & \mid & *_{\ell}\mathit{expr} & (\mathit{memory\ read\ of\ size}\ \ell \in \mathbb{N})
\end{array}
$$

Figure 3.1: Syntax of While-memory.

i.e., the subset of Y whose elements are the image of some element of X. We let $[x_1 \mapsto y_1,\ x_2 \mapsto y_2,\ \ldots,\ x_n \mapsto y_n]$ denote the function f mapping x_1 to y_1, x_2 to y_2, etc. We denote as $f[x \leftarrow y]$ the function $f' : \mathrm{dom}(f) \cup \{x\} \to \mathrm{codom}(f) \cup \{y\}$ the function mapping every $z \in \mathrm{dom}(f) \setminus \{x\}$ to $f(z)$ and x to y. When that notation is not ambiguous, we shall also denote as $f[x \leftarrow \bot]$ the restriction of f to $\mathrm{dom}(f) \setminus \{x\}$.

The set of all subsets of X, or *powerset* of X, is denoted $\mathcal{P}(\mathrm{X})$. A *partially ordered set* (or *poset*) $(\mathcal{D}, \sqsubseteq)$ is a set $\mathcal{D}$ together with a partial order $\sqsubseteq$, that is, a binary relation that is reflexive, transitive, and antisymmetric. A *lattice* is a poset $(\mathcal{D}, \sqsubseteq)$ such that any two elements of $\mathcal{D}$ have a least upper bound and greatest lower bound. A *complete lattice* $(\mathcal{D}, \sqsubseteq)$ is a lattice such that any subset of $\mathcal{D}$ has a least upper bound and a greatest lower bound. In particular, for any set X, $(\mathcal{P}(\mathrm{X}), \subseteq)$ is a complete lattice.

If $(\mathcal{D}, \sqsubseteq)$ is a lattice and $\mathrm{F} : \mathcal{D} \to \mathcal{D}$, then a *fixpoint* of F is an element $x \in \mathcal{D}$ such that $\mathrm{F}(x) = x$; a *post-fixpoint* of F is an element $x \in \mathcal{D}$ such that $\mathrm{F}(x) \sqsubseteq x$. We shall use the fact that the set of fixpoints of a *monotone* function F over a lattice is itself a lattice. In particular, F has a least fixpoint, which we denote lfp F.

3.2 The While-memory language

Throughout this thesis, we will define our analyses on a simple imperative language, While-memory, whose syntax is given in Figure 3.1.

While-memory features basic assignments, usual arithmetic expressions, memory operations (allocation, read, write) and standard control flow commands. Memory locations include a finite set of variables $\mathbb{X}$ and a finite set of addresses $\mathbb{A}$ that can be computed using pointer arithmetic. The analysis is parameterized by the choice of an *application binary interface* (or ABI) that fixes endianness, basic types sizes and alignments. For the sake of readability, in what follows we assume a little-endian ABI. Let $\mathcal{W}$ denote the number of bytes in a machine word.

While-memory enables dynamic allocation via the $x := \mathbf{malloc}(e)$ construct, where x is a variable name, and e an expression evaluating to an integer size.

Since the two problems of dangling pointer use (use-after-free bugs) and memory leaks are out of the scope of this thesis, there is no "free" operator in the language. As a consequence, our static analyses will be unable to detect such bugs, or to detect memory leaks; for instance, in C, the `free` standard function will be treated as a no-op (see Section 7.1.1).

The values manipulated by While-memory are bit vectors of a certain size. We will represent them as a pair consisting of that size (in bytes) and of an integer, corresponding to the interpretation of the bit vector as a non-negative integer in base 2:

Definition 3.1 (Bit vectors). The set of bit vectors is:

$$\mathbb{V} \overset{\text{def}}{=} \{(\ell, v) \mid \ell \in \mathbb{N}, v \in [0, 2^{8\ell} - 1]\}$$

For $n \geq 0$, we let $\mathbb{V}_n$ denote the set of bit vectors of length n bytes.

We extend binary operator notation on bit vectors of the same byte length, i.e., the notation $(\ell, v_1) \diamond (\ell, v_2)$ means $(\ell,\ v_1 \diamond v_2)$. The fact that bit vectors have a definite length allows to define bit vector concatenation very simply:

Definition 3.2 (Bitvector concatenation). The concatenation of any two bit vectors x and y is denoted $x :: y$ and defined by:

$$(\ell_1, v_1) :: (\ell_2, v_2) = \left(\ell_1 + \ell_2,\ v_1 + 2^{8\ell_1} v_2\right)$$

Definition 3.3 (Addresses, variable stores, heaps, states). *Memory addresses* are word-sized bit vectors:

$$\mathbb{A} = \mathbb{V}_{\mathcal{W}}$$

We let *variable stores* (or simply *stores*) be maps from variable names to their contents:

$$\Sigma = \mathbb{X} \to \mathbb{V}$$

and *heaps* be partial functions from addresses to one-byte values:

$$\mathbb{H} = \mathbb{A} \rightharpoonup \mathbb{V}_1$$

Program states, whose set is denoted $\mathbb{S}$, pair a variable store and a heap:

$$\mathbb{S} = \Sigma \times \mathbb{H}.$$

Given that WHILE-MEMORY supports memory reads and writes of an arbitrary number of bytes, we introduce shorthand notations for such multi-byte operations.

Definition 3.4 (Multi-cell memory read). Given a heap $h \in \mathbb{H}$, $a \in \mathbb{A}$, and $\ell \in \mathbb{N}$, we let $h[a..a+\ell]$ denote the reading of a region of size ℓ at address a.

$$h[a..a+\ell] = h(a) :: h(a+1) :: \cdots :: h(a+\ell-1)$$

Note that this definition comes from our assumption of a little-endian architecture.

In addition, we denote by $\sigma[x \leftarrow v]$ the store σ with x now mapped to v, and by $h[a..a+\ell \leftarrow v]$ the heap h with values at addresses a (included) to $a+\ell$ (excluded) replaced with the bytes from v. Finally, dropping a range of mappings from a heap is noted $h[a..a+\ell \leftarrow \bot]$.

The semantics of the language is given in Figures 3.2 and 3.3 as a denotational semantics. The semantics of expressions $\mathcal{E}[\![\cdot]\!]$ is defined as a partial function that takes a state and returns a value, or is undefined in the case of a runtime error, i.e., a division by zero or a memory read at an invalid address. Arithmetic operations are performed modulo $2^{8\mathcal{W}}$.

The semantics of statements $[\![\cdot]\!]$, given a WHILE-MEMORY statement p and a set of states S, computes all non-error states reachable by executing p from one of the states in S. Error states —e.g., due to a division by zero or a write to an invalid heap address— are not included in this semantics.

The semantics of **skip** is the identity function, while the semantics of $\mathrm{P}_1; \mathrm{P}_2$ is the composition of the semantics of P_1 and P_2.

$$
\begin{aligned}
\mathcal{E}[\![\cdot]\!] &: \mathit{expr} \times \mathbb{S} \rightharpoonup \mathbb{V} \\
\mathcal{E}[\![c]\!]s &= c \\
\mathcal{E}[\![x]\!](\sigma, h) &= \sigma(x) \\
\mathcal{E}[\![*_{\ell}e]\!](\sigma, h) &= h[a..a+\ell] \quad \text{if } a = \mathcal{E}[\![e]\!](\sigma, h) \text{ and } [a, a+\ell[\; \subseteq \mathrm{dom}(h) \\
\mathcal{E}[\![e_1 / e_2]\!]s &= (\ell, v_1 / v_2) \quad \text{if } \mathcal{E}[\![e_1]\!]s = (\ell, v_1) \text{ and } \mathcal{E}[\![e_2]\!]s = (\ell, v_2) \text{ with } v_2 \neq 0 \\
\mathcal{E}[\![e_1 \diamond e_2]\!]s &= (\ell, v_1 \diamond v_2) \quad \text{if } \mathcal{E}[\![e_1]\!]s = (\ell, v_1) \text{ and } \mathcal{E}[\![e_2]\!]s = (\ell, v_2)
\end{aligned}
$$

Figure 3.2: Semantics of WHILE-MEMORY expressions.

The *variable assignment* $x := e$ evaluates e and, if the evaluation completed without runtime errors, updates the variable store with the resulting value.

The *memory write* $*_{\ell}e_1 := e_2$ evaluates e_1 and, if it results in a valid heap address —i.e., a word-sized value that is in the domain of h—, stores the result of evaluating e_2 at that address, if e_2 evaluates to a bit vector of size ℓ.

The *memory allocation* $x :=$ **malloc**(e) makes a non-deterministic choice: either it assigns x to 0, or, if e evaluates to a positive value ℓ, writes an indeterminate value to a region that was previously unmapped in h, and assigns to x the base address of that region.

The semantics of the conditional **if** e **then** P_1 **else** P_2 **end** contains the results of executing P_1 when e evaluates to a *non-zero* value, and P_2 when it evaluates to zero. Its definition uses the operator $\mathcal{C}[\![e]\!]$, which filters out states in which the evaluation of e is undefined or results in 0.

Finally, the semantics of loops **while** e **do** P **done** is standard; first, it computes the tightest loop invariant as the least fixpoint $\mathrm{lfp}\,\mathrm{F}$ of the operator $\mathrm{F(X)} = \mathrm{S} \cup [\![\mathrm{P}]\!]\big(\mathcal{C}[\![e]\!]\mathrm{X}\big)$ which corresponds to the loop semantics, i.e., executing P on all states that pass the condition e. This least fixpoint exists because $(\mathcal{P}(\mathbb{S}), \subseteq)$ is a lattice and F is monotone. Then, the states are filtered by $\mathcal{C}[\![\neg e]\!]$ to keep only the states that can exit the loop by failing the loop condition e.

3.3 Notion of abstraction

The semantics $[\![\mathrm{P}]\!]$ is not suitable for automated analysis as it is not computable.

Abstract interpretation [CC77] is a general framework to reason about program analyzers. In particular, it is often used as a formal blueprint for building sound, automated static analyzers. It consists in defining an *abstract semantics* of the language which, unlike the concrete semantics, is computable, and is an over-approximation of the concrete semantics, in a sense that we make precise below.

Definition 3.5 (Concretization, soundness, exactness). Let $(\mathbb{C}, \leq)$ and $(\mathbb{D}, \sqsubseteq)$ be two posets. A *concretization* is a monotone function $\gamma : \mathbb{D} \to \mathbb{C}$. The object $\mathbb{d} \in \mathbb{D}$ is said to be a *sound abstraction* of $c \in \mathbb{C}$ if $c \leq \gamma(\mathbb{d})$. It is an *exact* abstraction if $c = \gamma(\mathbb{d})$.

$\mathbb{C}$ is often called the *concrete domain* and $\mathbb{D}$ the *abstract domain*.

Example 3.1. Let $\mathbb{I}$ be the set of integer intervals whose elements are non-negative and representable on $\mathcal{W}$ bytes:

$$\mathbb{I} = \{[a, b] \mid a \in [0, 2^{8\mathcal{W}} - 1],\; b \in [0, 2^{8\mathcal{W}} - 1],\; a \leq b\} \cup \{\emptyset\}$$

Any set of integers $\mathrm{S} \in \mathcal{P}\big([0, 2^{8\mathcal{W}} - 1]\big)$ can be abstracted by an interval, the concretization being the identity. For example, $[1, 17]$ is a sound abstraction of $\{1, 3, 17\}$. In this setting, the concrete domain is $\big(\mathcal{P}\big([0, 2^{8\mathcal{W}} - 1]\big), \subseteq\big)$ and the abstract domain is $(\mathbb{I}, \subseteq)$.

$$
\begin{aligned}
\mathcal{C}\llbracket\cdot\rrbracket &\;:\; \mathit{expr} \times \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S}) \\
\mathcal{C}\llbracket e\rrbracket \mathrm{S} &\;=\; \{s \in \mathrm{S} \mid \mathcal{E}\llbracket e\rrbracket s = (\ell, v),\ v \neq 0\}
\end{aligned}
$$

$$
\begin{aligned}
\llbracket\cdot\rrbracket &\;:\; \mathit{stmt} \times \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S}) \\
\llbracket \textbf{skip} \rrbracket \mathrm{S} &\;=\; \mathrm{S} \\
\llbracket \mathrm{P}_1; \mathrm{P}_2 \rrbracket \mathrm{S} &\;=\; (\llbracket \mathrm{P}_2 \rrbracket \circ \llbracket \mathrm{P}_1 \rrbracket)\mathrm{S} \\
\llbracket x := e \rrbracket \mathrm{S} &\;=\; \{(\sigma[x \leftarrow v], h) \mid (\sigma, h) \in \mathrm{S},\ v = \mathcal{E}\llbracket e\rrbracket(\sigma, h)\} \\
\llbracket *_\ell e_1 := e_2 \rrbracket \mathrm{S} &\;=\; \left\{(\sigma, h[a..a+\ell \leftarrow v]) \;\middle|\; \begin{array}{l} (\sigma, h) \in \mathrm{S},\ \mathcal{E}\llbracket e_1\rrbracket(\sigma, h) = (\mathcal{W}, a), \\ \mathcal{E}\llbracket e_2\rrbracket(\sigma, h) = (\ell, v) \end{array} \right\} \\
\llbracket x := \textbf{malloc}(e) \rrbracket \mathrm{S} &\;=\; \left\{(\sigma[x \leftarrow a],\ h[a..a+\ell \leftarrow v]) \;\middle|\; \begin{array}{l} (\sigma, h) \in \mathrm{S},\ \mathcal{E}\llbracket e\rrbracket(\sigma, h) = (\mathcal{W}, \ell), \\ \ell > 0, [a, a+\ell[\cap \mathrm{dom}(h) = \varnothing, v \in \mathbb{V}_\ell \end{array} \right\} \\
&\qquad \cup \left\{(\sigma[x \leftarrow (\mathcal{W}, 0)],\ h) \;\middle|\; \begin{array}{l} (\sigma, h) \in \mathrm{S},\ \mathcal{E}\llbracket e\rrbracket(\sigma, h) = (\mathcal{W}, \ell), \\ \ell > 0 \end{array} \right\} \\
\llbracket \textbf{if}\ e\ \textbf{then}\ \mathrm{P}_1\ \textbf{else}\ \mathrm{P}_2\ \textbf{end} \rrbracket \mathrm{S} &\;=\; \llbracket \mathrm{P}_1 \rrbracket(\mathcal{C}\llbracket e\rrbracket \mathrm{S}) \cup \llbracket \mathrm{P}_2 \rrbracket(\mathcal{C}\llbracket \neg e\rrbracket \mathrm{S}) \\
\llbracket \textbf{while}\ e\ \textbf{do}\ \mathrm{P}\ \textbf{done} \rrbracket \mathrm{S} &\;=\; \mathcal{C}\llbracket \neg e\rrbracket(\mathrm{lfp}\,\mathrm{F}) \\
&\qquad \text{where } \mathrm{F}(\mathrm{X}) \stackrel{\text{def}}{=} \mathrm{S} \cup \llbracket \mathrm{P} \rrbracket(\mathcal{C}\llbracket e\rrbracket \mathrm{X})
\end{aligned}
$$

Figure 3.3: Denotational semantics of WHILE-MEMORY statements.

Example 3.2. Intervals can themselves be abstracted by representing them as a pair of bounds, or by a special element $\perp$ to denote the empty interval. Consider the abstract domain:

$$\hat{\mathbb{I}} = \{(a, b) \mid a \in [0, 2^{8\mathcal{W}} - 1],\ b \in [0, 2^{8\mathcal{W}} - 1],\ a \leq b\} \cup \{\perp\}$$

The concretization is defined by:

$$
\begin{aligned}
&\gamma_{\mathrm{I}} : \hat{\mathbb{I}} \to \mathbb{I} \\
&\gamma_{\mathrm{I}}(\,(a, b)\,) = \{x \in \mathbb{Z} \mid a \leq x \leq b\} \\
&\gamma_{\mathrm{I}}(\perp) = \varnothing
\end{aligned}
$$

And the partial order $\sqsubseteq_{\mathrm{I}}$ is defined by the fact that $(a, b) \sqsubseteq_{\mathrm{I}} (c, d)$ if and only if $c \leq a$ and $b \leq d$, and that $\perp$ is the lower bound. Note that this abstraction of intervals is more amenable to efficient machine representation and manipulation.

For the sake of readability, in what follows, we will use interval notation $[a, b]$ to denote a pair of bounds, and $\mathbb{I}$ to mean $\hat{\mathbb{I}}$.

Example 3.3. WHILE-MEMORY variable stores can be abstracted by mapping each variable to an interval, using as abstract domain $\mathbb{X} \to \mathbb{I}$ with concretization:

$$
\begin{aligned}
&\gamma : (\mathbb{X} \to \mathbb{I}) \to \mathcal{P}(\Sigma) \\
&\gamma(\hat{\sigma}) = \{\sigma \in \Sigma \mid \forall x \in \mathbb{X},\ \sigma(x) = (\mathcal{W}, v) \wedge v \in \gamma_{\mathrm{I}}(\hat{\sigma}(x))\}
\end{aligned}
$$

That is, possible bit vector values for $\sigma(x)$ are the $\mathcal{W}$-sized bit vectors that contain a value in the interval $\hat{\sigma}(x)$. The partial order of the abstract domain is the pointwise extension of interval inclusion, denoted

$\sqsubseteq_{\mathbb{X}\to\mathbb{I}}$ and defined as:

$$\hat{\sigma}_1 \sqsubseteq_{\mathbb{X}\to\mathbb{I}} \hat{\sigma}_2 \iff \forall x \in \mathbb{X}, \hat{\sigma}_1(x) \sqsubseteq_{\mathrm{I}} \hat{\sigma}_2(x)$$

A stronger form of correspondence between abstract and concrete domains may exist in the form of Galois connections [CC77]. However, sometime no Galois connection exists between the abstract and concrete domain, and the existence of a monotone concretization is sufficient to reason about abstraction and soundness. In this thesis, we will not use this stronger framework, so that our abstractions remain composable with abstract domains that do not enjoy a Galois connection, such as polyhedra [CH78].

3.3.1 Operator abstraction

Definition 3.6 (Operator abstraction)**.** Let $(\mathbb{D}, \sqsubseteq)$ denote an abstract domain concretizing to the concrete domain $(\mathbb{C}, \leq)$ through the concretization function γ, and let $f : \mathbb{C} \to \mathbb{C}$ be an operator on $\mathbb{C}$. $g : \mathbb{D} \to \mathbb{D}$ is a sound abstraction of f if, for all abstract objects $\mathbb{d} \in \mathbb{D}$, $f(\gamma(\mathbb{d})) \leq \gamma(g(\mathbb{d}))$.

Example 3.4. Consider the operator adding 1 to all integers of a set: $f = \lambda \mathrm{X}. \{x + 1 \mid x \in \mathrm{X}\}$. Then $\lambda[a, b].\,]{-\infty}, {+\infty}[$ is a sound abstraction of f in the interval domain $\mathbb{I}$. Another sound abstraction is $\lambda[a, b].\, [a + 1,\ b + 1]$. This abstraction is also exact.

Example 3.5. Consider the concrete operator $[\![x := 7]\!]$, corresponding to assigning the value 7 to the variable x. A sound abstraction of this operator in $\mathbb{X} \to \mathbb{I}$, which we will denote $[\![x := 7]\!]^\sharp : (\mathbb{X} \to \mathbb{I}) \to (\mathbb{X} \to \mathbb{I})$, can be defined as:

$$[\![x := 7]\!]^\sharp \hat{\sigma} = \hat{\sigma}[x \leftarrow [7, 7]]$$

To compute the abstract semantics of a program, one needs an abstraction of every operation that can be performed in the concrete semantics. In the case of While-memory, this means an abstract operator for variable assignment (denoted $[\![x := e]\!]^\sharp$), another for heap writing $[\![*_\ell e_{\mathrm{loc}} := e]\!]^\sharp$, and so on for all base constructs of the While-memory syntax (see below Section 3.4).

3.3.2 Relational and non-relational numerical abstractions

$\mathbb{X} \to \mathbb{I}$ is an example of what is commonly called a numerical abstract domain, as it abstracts a set of numerical variables. Abstract domains of the form $\mathbb{X} \to \mathbb{D}^\sharp$, where $\mathbb{D}^\sharp$ abstracts a single value, are called *non-relational abstractions*. Such abstractions abstract each variable independently, and thus fail to express relations between them.

Example 3.6. Consider this example from Miné [Min17] where x and y start out as containing unknown values between 0 and 10, before x is assigned the minimum of x and y:

```
// Assumption: x ∈ [0, 10], y ∈ [0, 10]
if y ≤ x then x := y else skip end;
```

It is clear that at the end of this program, the set of possible values for the (x, y) variable pair is

$$\{(x, y) \mid x \in [0, 10], y \in [0, 10], x \leq y\}$$

However, the most precise abstraction of this fact in $\mathbb{X} \to \mathbb{I}$ is:

$$\{(x, y) \mid x \in [0, 10], y \in [0, 10]\}$$

This situation is represented in Figure 3.4.

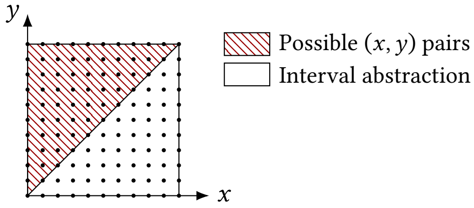


Figure 3.4: Set of possible (x, y) pairs and its approximation by a non-relational abstraction.

Example 3.7 (The zone abstract domain). Consider the abstract domain consisting in a conjunction of formulas of the form $x_i - x_j \leq c$ or $\pm x_i \leq c$, where $x_i, x_j \in \mathbb{X}$ and $c \in \mathbb{V}$. This domain is called the zone abstract domain [Min04], and contains an exact abstraction of the program property described in Example 3.6, namely the formula:

$$x \leq y \wedge -x \leq 0 \wedge x \leq 10 \wedge -y \leq 0 \wedge y \leq 10$$

In an implementation context, such abstract elements will typically be represented by difference bound matrices (DBMs).

The zone domain is an example of a *relational* abstract domain.

In a numerical domain concretizing to $\mathcal{P}(\mathbb{X} \to \mathbb{V})$, elements of $\mathbb{X}$ are usually called the *keys* or *dimensions* of the abstract domain. In the previous examples, the number of keys was finite and fixed. However, when analyzing heap-manipulating programs, there is often a need for a varying number of keys of the numerical domain, corresponding to a varying number of memory cells. This varying number of keys can be represented by a varying number of *symbolic variables*, which play the role of the keys of an independent numerical abstract domain. This is the mechanism used in shape abstract domains [LCR10; CR13].

The abstractions that we will define will also involve a varying number of symbolic variables (Chapters 4 to 6). To represent the use of numerical abstract domains with a varying number of keys, in the rest of this thesis we will formalize numerical domains as concretizing to a set of *valuations* of the symbolic variables, i.e., elements of $\mathbb{V}^\# \to \mathbb{V}$, where $\mathbb{V}^\# = \{\mathbb{v}_0, \mathbb{v}_1, ...\}$ is an *infinite*, countable set of symbolic variables.

Definition 3.7 (Numerical abstract domain accounting for the symbolic variables). In the rest of the thesis, we assume the existence of an abstract domain $\mathbb{D}_{\text{num}}$ with concretization:

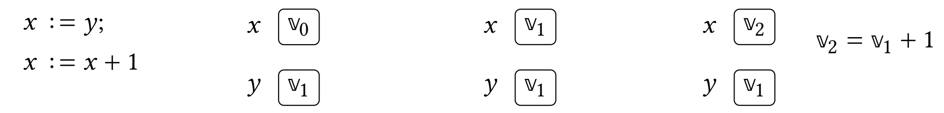
$$\gamma_{\text{num}} : \mathbb{D}_{\text{num}} \to \mathcal{P}(\mathbb{V}^\# \to \mathbb{V})$$

$\mathbb{D}_{\text{num}}$ is a parameter of the analysis. It may be either relational or non-relational.

Note that we formalize $\mathbb{V}^\#$ as an infinite set to avoid placing an *a priori* limit on the expressible numerical predicates; however, for the analysis to be computable, the abstract domains involved can only express constraints over a finite subset of $\mathbb{V}^\#$.

Example 3.8. Consider the abstract domain associating a symbolic variable to each variable name: $\mathbb{D}^\# = \mathbb{X} \to \mathbb{V}^\#$. To account for the values of symbolic variables, it concretizes not only to concrete variable stores, but also to the associated valuations:

$$\begin{aligned} \gamma_{\text{D}} &: \mathbb{D}^\# \to \mathcal{P}(\Sigma \times (\mathbb{V}^\# \to \mathbb{V})) \\ \gamma_{\text{D}}(\sigma^\#) &= \{(\sigma, \nu) \mid \nu : \mathbb{V}^\# \to \mathbb{V},\ \forall x \in \mathbb{X},\ \sigma(x) = \nu(\sigma^\#(x))\} \end{aligned}$$



(a) A simple WHILE-MEMORY program (b) Initial state (c) State after $x := y$ (d) Final state

Figure 3.5: Example of abstraction of Σ using symbolic variables.

Then, $\mathbb{D}^\#$ can be used in combination with $\mathbb{D}_{\text{num}}$, by using pairs of $\mathbb{D}^\# \times \mathbb{D}_{\text{num}}$ as an abstraction of states. This is a standard construct called *product abstraction* [CC79]. A possible concretization function is:

$$\gamma_{\text{comb}}(\sigma^\#, \upsilon^\#) = \{\sigma \mid (\sigma, \upsilon) \in \gamma_D(\sigma^\#) \wedge \upsilon \in \gamma_{\text{num}}(\upsilon^\#)\}$$

Assuming that $\mathbb{X} = \{x, y\}$, Figure 3.5b represents an example abstract state $(\sigma^\#, \upsilon^\#)$ such that $\sigma^\#(x) = \mathbb{v}_0$ and $\sigma^\#(y) = \mathbb{v}_1$ and $\upsilon^\#$ represents the absence of any constraint on any symbolic variables of $\mathbb{V}^\#$. Without attempting to define an entire abstract semantics for $\mathbb{D}^\#$, we show the effect of two simple WHILE-MEMORY statements: the statement $x := y$ causes $\sigma^\#$ to map x and y to the same symbolic variable (Figure 3.5c); the statement $x := x + 1$ first evaluates $x + 1$, which allocates a fresh symbolic variable $\mathbb{v}_2$ and creates a numerical abstract state $\upsilon^{\#\prime}$ that represents the constraint $\mathbb{v}_2 = \mathbb{v}_1 + 1$, all other symbolic variables being unconstrained. The newly constrained symbolic variable $\mathbb{v}_2$ is then assigned to x (Figure 3.5d) in a new abstract store $\sigma^{\#\prime}$. The concretization of the final abstract state is:

$$\gamma_D(\sigma^{\#\prime}) = \{(\sigma, \upsilon) \mid \sigma(x) = \upsilon(\mathbb{v}_1) \wedge \sigma(y) = \upsilon(\mathbb{v}_2) \wedge \upsilon(\mathbb{v}_2) = \upsilon(\mathbb{v}_1) + 1\}$$

and

$$\gamma_{\text{comb}}(\sigma^{\#\prime}, \upsilon^{\#\prime}) = \{\sigma \in \Sigma \mid \sigma(x) = \sigma(y) + 1\}$$

Our abstractions will often require to perform *requests* on the numerical abstract domain, i.e., check whether some numerical constraints hold. We introduce the following notation:

Definition 3.8 (Requests on the numerical abstract domain). Given a numerical abstract state $\upsilon^\# \in \mathbb{D}_{\text{num}}$ and a quantifier-free formula p with free variables in $\mathbb{V}^\#$, we denote:

$$\upsilon^\# \vDash p$$

the fact that the constraint p holds in all valuations of $\gamma_{\text{num}}(\upsilon^\#)$.

Finally, we assume the ability to *add a constraint* to a valuation: $\upsilon^\#[p]$ denotes the numerical abstract state $\upsilon^\#$ with the added constraint p.

Note that $\upsilon^\# \vDash p$ entails that p holds in all valuations $\upsilon \in \gamma_{\text{num}}(\upsilon^\#)$. But, due to the over-approximating nature of abstract interpretation, if $\upsilon^\# \vDash p$ does not hold, nothing can be deduced about $\gamma_{\text{num}}(\upsilon^\#)$.

In addition, the nature of the numerical predicates represented by $\mathbb{D}_{\text{num}}$ depends on the choice of the numerical abstract domain. For example, the interval abstract domain $\mathbb{V}^\# \to \mathbb{I}$ is unable to represent inequalities between symbolic variables, such as $\mathbb{v}_0 < \mathbb{v}_1$. As a consequence, it is not the case that $\upsilon^\#[p] \vDash p$ for all constraints p.

3.4 Abstract semantics of WHILE-MEMORY

The abstract semantics depends on each abstract domain and is part of its definition; however, all of our abstract semantics share a common structure and handle similarly the following language constructs, namely **skip**, statements sequences, conditionals, and loops.

Therefore, in this section, we give the structure of the abstract semantics of WHILE-MEMORY in an abstract domain $\mathbb{D}^\sharp$ with concretization $\gamma : \mathbb{D}^\sharp \to \mathcal{P}\big(\mathbb{S} \times (\mathbb{V}^\sharp \to \mathbb{V})\big)$. The elements of $\mathbb{D}^\sharp$ are called *abstract states*. Figure 3.8, page 31 summarizes the main definitions.

3.4.1 Abstract semantics of expressions

The operator $\mathcal{E}[\![e]\!]^\sharp : \mathbb{D}^\sharp \to \mathbb{V}_\mathrm{D} \times \mathbb{D}^\sharp$ evaluates the expression e in a given abstract state. It returns a pair consisting of two elements:

- What we call an *abstract value*, representing the evaluation result. Different abstract domains may use different sets of abstract values. We let $\mathbb{V}_\mathrm{D}$ denote the set of abstract values for domain $\mathbb{D}^\sharp$. For instance, the domain from Example 3.8 uses symbolic variables as abstract values, i.e., expression evaluation results in a symbolic variable: $\mathbb{V}_\mathrm{D} = \mathbb{V}^\sharp$. In addition, we assume the existence of a *value concretization* function $\gamma_\mathrm{D}^\mathrm{V} : (\mathbb{V}^\sharp \to \mathbb{V}) \times \mathbb{V}_\mathrm{D} \to \mathbb{V}$ such that $\gamma_\mathrm{D}^\mathrm{V}(\nu, \mathbb{v})$ is the value abstracted by $\mathbb{v}$ in the context of ν.
- An new abstract state. Indeed, in our abstract domains, expression evaluation may modify the abstract state; consider again the example of Figure 3.5: evaluating the expression $x + 1$ returns $\mathbb{v}_2$ and a new abstract state expressing a numerical constraint on $\mathbb{v}_2$:

$$\mathcal{E}[\![x+1]\!]^\sharp(\sigma^\sharp, \nu^\sharp) = \big(\mathbb{v}_2,\ (\sigma^\sharp, \nu^\sharp[\mathbb{v}_2 = \mathbb{v}_1 + 1])\big)$$

The definition of $\mathcal{E}[\![\cdot]\!]^\sharp$ is part of the abstract domain definition.

3.4.2 Abstract semantics of simple statements

Given a WHILE-MEMORY program P, $[\![\mathrm{P}]\!]^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ denotes the abstract semantics of P in the abstract domain $\mathbb{D}^\sharp$. It is defined by induction on the language syntax.

The abstract operator $[\![\textbf{skip}]\!]^\sharp$ is simply the identity, and $[\![\mathrm{P}_1; \mathrm{P}_2]\!]^\sharp$ is the composition of the effect of $[\![\mathrm{P}_1]\!]^\sharp$ and $[\![\mathrm{P}_2]\!]^\sharp$, as is in the concrete semantics.

The abstract operators for variable assignment, memory writes and memory allocation depend on the abstract domain and are part of its definition.

3.4.3 Conditionals and loops

The semantics of conditionals and loops are common to all abstract domains; but they depend on operators that are part of the abstract domain definition, namely a *guard function*, as well as an *abstract inclusion*, an *abstract join* and a *widening*.

Guards

The abstract state $\mathbf{guard}_\mathrm{D}(\mathbb{v}, \mathbb{s})$ represents the restriction of the states abstracted by $\mathbb{s}$ to the states in which the abstract value $\mathbb{v}$ represents a non-zero value. It depends on the abstract domain and is part of its definition.

The guard function should be sound in the following sense:

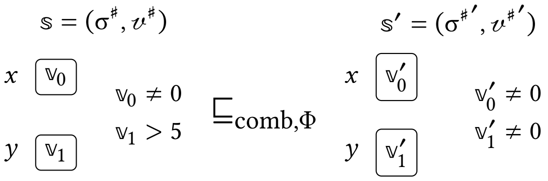


Figure 3.6: Example of abstract inclusion involving symbolic variables.

Theorem 3.1 (Soundness of $\mathbf{guard}_{\mathrm{D}}$). *Given an abstract state* $\mathbb{s} \in \mathbb{D}^\#$ *and an abstract value* $\mathbb{v} \in \mathbb{V}_{\mathrm{D}}$*:*

$$\forall (s, \nu) \in \gamma(\mathbf{guard}_{\mathrm{D}}(\mathbb{v}, \mathbb{s})),\ \gamma_{\mathrm{D}}^{\mathrm{V}}(\nu, \mathbb{v}) \neq 0$$

Abstract inclusion

In an abstract domain $(\mathbb{A}, \sqsubseteq)$ that abstract the concrete domain $(\mathcal{P}(\mathbb{C}), \subseteq)$, an *abstract inclusion* operator is an abstraction of concrete set inclusion $\subseteq$ (or more generally of the partial order of the concrete domain).

Definition 3.9 (Soundness of an abstract inclusion). $\sqsubseteq$ is a sound abstract inclusion if, for all abstract elements $\mathbb{a}_1$ and $\mathbb{a}_2$:

$$\mathbb{a}_1 \sqsubseteq \mathbb{a}_2 \implies \gamma(\mathbb{a}_1) \subseteq \gamma(\mathbb{a}_2)$$

However, in the case of an abstract domain like $\mathbb{D}^\#$ that concretizes to sets of (state, valuation) pairs, the definition of a sound abstract inclusion is more involved, due to the fact that abstract states may refer to distinct sets of symbolic variables.

Consider the inclusion between $\mathbb{s}$ and $\mathbb{s}'$, represented on Figure 3.6. It is clear that all concrete states abstracted by $\mathbb{s}$ are also abstracted by $\mathbb{s}'$. However, this is not true of the valuations, since the set of symbolic variables constrained in each case is different. But it is clear that $\mathbb{v}_0$ (resp. $\mathbb{v}_1$) and $\mathbb{v}_0'$ (resp. $\mathbb{v}_1'$) play similar roles in their respective states. We formalize this relation by a *renaming* $\Phi : \mathbb{V}^\# \to \mathbb{V}^\#$ of the symbolic variables.

In the rest of this thesis, we will assume that the numeric domain $\mathbb{D}_{\mathrm{num}}$ provides an abstract inclusion operator *up to a renaming* Φ:

Definition 3.10 (Abstract inclusion operator of $\mathbb{D}_{\mathrm{num}}$). For all $\Phi : \mathbb{V}^\# \to \mathbb{V}^\#$, $\mathbb{D}_{\mathrm{num}}$ is assumed to provide a computable relation $\sqsubseteq_{\mathrm{num},\Phi} \subseteq \mathbb{D}_{\mathrm{num}} \times \mathbb{D}_{\mathrm{num}}$ which is sound in the following sense: for all $\nu_1^\#, \nu_2^\# \in \mathbb{D}_{\mathrm{num}}$:

$$\nu_1^\# \sqsubseteq_{\mathrm{num},\Phi} \nu_2^\# \implies \forall \nu \in \gamma_{\mathrm{num}}(\nu_1^\#),\ \nu \circ \Phi \in \gamma_{\mathrm{num}}(\nu_2^\#)$$

Each abstract domain $\mathbb{D}^\#$ that we will define will provide two things as part of its definition:

- A definition of a *renaming* from one state to another, as well as an algorithm to construct such a renaming.
- An abstract inclusion up to a renaming Φ, denoted $\sqsubseteq_{\mathrm{D},\Phi}$, that is sound in the following sense:

Definition 3.11 (Soundness of $\sqsubseteq_{\mathrm{D},\Phi}$). Given $\mathbb{s}_1, \mathbb{s}_2 \in \mathbb{D}^\#$, if Φ is a renaming from $\mathbb{s}_2$ to $\mathbb{s}_1$ and $\mathbb{s}_1 \sqsubseteq_{\mathrm{D},\Phi} \mathbb{s}_2$ holds, then:

$$\forall (s, \nu) \in \gamma_{\mathrm{D}}(\mathbb{s}_1),\ (s,\ \nu \circ \Phi) \in \gamma_{\mathrm{D}}(\mathbb{s}_2)$$

Note that the renaming is from the right-hand-side operand of the abstract inclusion to the left-hand-side. In addition, a renaming is not required to be injective.

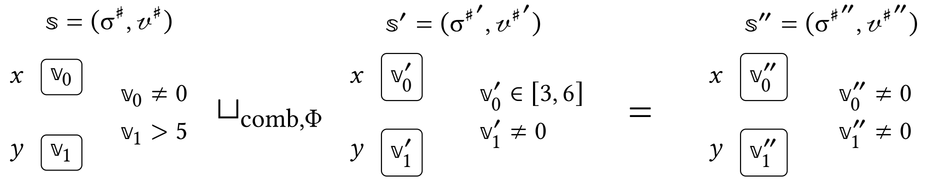


Figure 3.7: Example of abstract join involving symbolic variables.

Example 3.9. If $\mathbb{D}^\#$ is the domain $\mathbb{X} \to \mathbb{V}^\#$ from Example 3.8, then a *renaming* from the variables of $\sigma_2^\#$ to the variables of $\sigma_1^\#$ is any function $\Phi : \mathbb{V}^\# \to \mathbb{V}^\#$ such that:

$$\forall x \in \mathbb{X},\ \Phi(\sigma_2^\#(x)) = \Phi(\sigma_1^\#(x))$$

Clearly, such a renaming can be constructed by iterating over the contents of $\sigma_1^\#$ and $\sigma_2^\#$. A possible renaming from $\mathbb{s}'$ to $\mathbb{s}$, where $\mathbb{s}$ and $\mathbb{s}'$ are the abstract states depicted in Figure 3.6, is Φ such that $\Phi(\mathbb{v}_0') = \mathbb{v}_0$, $\Phi(\mathbb{v}_1') = \mathbb{v}_1$ and is the identity everywhere else. Then $\sigma^\# \sqsubseteq_{\text{D},\Phi} \sigma^{\#'}$ holds. Note that this abstract inclusion does not consider numerical constraints.

From there, we can define an abstract inclusion $\sqsubseteq_{\text{comb},\Phi}$ on $\mathbb{D}^\# \times \mathbb{D}_{\text{num}}$ by:

$$(\sigma^\#, v^\#) \sqsubseteq_{\text{comb},\Phi} (\sigma^{\#'}, v^{\#'}) \iff \sigma^\# \sqsubseteq_{\text{D},\Phi} \sigma^{\#'} \wedge v^\# \sqsubseteq_{\text{num},\Phi} v^{\#'}$$

This abstract inclusion is sound in the usual sense: if Φ is a renaming from the variables of $\mathbb{s}_2$ to the variables of $\mathbb{s}_1$, then:

$$\mathbb{s}_1 \sqsubseteq_{\text{comb},\Phi} \mathbb{s}_2 \implies \forall \sigma \in \gamma_{\text{comb}}(\mathbb{s}_1),\ \sigma \in \gamma_{\text{comb}}(\mathbb{s}_2)$$

We allow ourselves to omit the subscript Φ when unnecessary:

Notation 3.1 (Abstract inclusion operator). *Given $\mathbb{s}_1, \mathbb{s}_2 \in \mathbb{D}^\#$, we write $\mathbb{s}_1 \sqsubseteq_{\text{D}} \mathbb{s}_2$ if and only if there exists a valid renaming Φ from $\mathbb{s}_2$ to $\mathbb{s}_1$ such that $\mathbb{s}_1 \sqsubseteq_{\text{D},\Phi} \mathbb{s}_2$.*

Abstract join

In any abstract domain $(\mathbb{A}, \sqsubseteq)$ that abstracts the concrete domain $(\mathcal{P}(\mathbb{C}), \subseteq)$, a join operator is an abstraction of set union $\cup$ (or more generally of the greatest lower bound of the concrete domain, which is always assumed to have a lattice structure).

Definition 3.12 (Soundness of an abstract join). $\sqcup$ is a sound join operator if, for all abstract elements $\mathbb{a}_1$ and $\mathbb{a}_2$:

$$\gamma(\mathbb{a}_1) \cup \gamma(\mathbb{a}_2) \subseteq \gamma(\mathbb{a}_1 \sqcup \mathbb{a}_2)$$

However, like with abstract inclusion, in our framework the definition of a sound abstract join is more involved. Consider the problem of joining the two states $\mathbb{s}$ and $\mathbb{s}'$ in Figure 3.7. The result is a similar state containing fresh symbolic variables. The constraints on these variables are the disjunction of the constraints on their counterparts in the operands.

To make this notion of "counterpart" precise, we introduce *two-way variable renamings*, which consist in sets of tuples of variables:

Example 3.10. In the domain $\mathbb{D}^\#$ from Example 3.8, a *two-way renaming* between states $\sigma^\#$ and $\sigma^{\#'}$ is a set $\Psi \subseteq (\mathbb{V}^\#)^3$ such that for all $x \in \mathbb{X}$, there exists a unique $\mathbb{v}'' \in \mathbb{V}^\#$ such that $(\sigma^\#(x), \sigma^{\#'}(x), \mathbb{v}'') \in \Psi$ and $\mathbb{v}''$ is fresh (i.e. does not appear in one of the input states) and does not appear elsewhere in Ψ. In Figure 3.7, a possible two-way renaming is $\Psi = \{(\mathbb{v}_0, \mathbb{v}_0', \mathbb{v}_0''),\ (\mathbb{v}_1, \mathbb{v}_1', \mathbb{v}_1'')\}$.

Each abstract domain that we will define will provide:

- A definition of valid two-way renamings between two abstract states, as well as a method to construct them.
- An abstract join up to a renaming $\sqcup_{\mathrm{D},\Psi}$, which is sound in the following sense:

Definition 3.13 (Soundness of an abstract join up to a renaming). Given $\mathbb{s}_1, \mathbb{s}_2 \in \mathbb{D}^\sharp$, if Ψ is a valid two-way renaming between $\mathbb{s}_1$ and $\mathbb{s}_2$ then:

$$\forall (s, v) \in \gamma_{\mathrm{D}}(\mathbb{s}_i),\ (s,\ v \circ \Phi_i) \in \gamma_{\mathrm{D}}(\mathbb{s}_1 \sqcup_{\mathrm{D},\Psi} \mathbb{s}_2) \quad \textit{for } i \in \{1, 2\}$$

where Φ_1 and Φ_2 are defined by

$$(\Phi_1(\mathbb{v}''), \Phi_2(\mathbb{v}'')) = (\mathbb{v}, \mathbb{v}') \iff (\mathbb{v}, \mathbb{v}', \mathbb{v}'') \in \Psi$$

In addition, we assume that $\mathbb{D}_{\mathrm{num}}$ provides a sound abstract join up to a renaming:

Definition 3.14 (Abstract join in $\mathbb{D}_{\mathrm{num}}$). We assume the existence of an operator $\sqcup_{\mathrm{num},\Psi} : \mathbb{D}_{\mathrm{num}} \times \mathbb{D}_{\mathrm{num}} \to \mathbb{D}_{\mathrm{num}}$, which is sound in the following sense: for all $v_1^\sharp, v_2^\sharp \in \mathbb{D}_{\mathrm{num}}$, for all $\Psi \in \mathcal{P}\Big((\mathbb{V}^\sharp)^3\Big)$, if $v_3^\sharp = v_1^\sharp \sqcup_{\mathrm{num},\Psi} v_2^\sharp$, then:

$$\begin{aligned} \forall v_1 \in \gamma_{\mathrm{num}}(v_1^\sharp),\ v_1 \circ \Phi_1 \in \gamma_{\mathrm{num}}(v_3^\sharp) \\ \forall v_2 \in \gamma_{\mathrm{num}}(v_2^\sharp),\ v_2 \circ \Phi_2 \in \gamma_{\mathrm{num}}(v_3^\sharp) \end{aligned}$$

where

$$(\Phi_1(\mathbb{v}''), \Phi_2(\mathbb{v}'')) = (\mathbb{v}, \mathbb{v}') \iff (\mathbb{v}, \mathbb{v}', \mathbb{v}'') \in \Psi$$

As with abstract inclusion, we allow ourselves to omit the subscript Ψ when a valid renaming is known to exist:

Notation 3.2 (Abstract join operator). *Given $\mathbb{s}_1, \mathbb{s}_2 \in \mathbb{D}^\sharp$, we write $\mathbb{s}_1 \sqcup_{\mathrm{D}} \mathbb{s}_2$ to mean $\mathbb{s}_1 \sqcup_{\mathrm{D},\Phi} \mathbb{s}_2$, where Ψ is any two-way renaming between $\mathbb{s}_1$ and $\mathbb{s}_2$.*

Semantics of conditionals

In the abstract operator $[\![$**if** e **then** P_1 **else** P_2 **end**$]\!]^\sharp$, the semantics of each branch of the conditional are computed after applying the **guard** function. The results of evaluating the two branches are then joined using $\sqcup_{\mathrm{D}}$.

Semantics of loops

The abstract semantics of a loop $[\![$**while** e **do** P **done**$]\!]^\sharp$ shares many similarities with its concrete semantics. However, rather than a least fixpoint, it is computed as a post-fixpoint using the lim operator:

Definition 3.15 (Post-fixpoint operator). For any poset $(\mathbb{D}^\sharp, \sqsubseteq)$ containing a smallest element $\bot$, and for any monotone operator $\mathrm{F} : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$:

$$\lim \mathrm{F} = \mathrm{F}^{\delta}(\bot)$$

where δ is the smallest integer such that $\mathrm{F}^{\delta}(\bot) \sqsubseteq \mathrm{F}^{\delta+1}(\bot)$, if it exists.

If $\mathbb{D}^\sharp$ has infinite increasing chains, lim F may not be defined or, more practically, may take a long time to compute. To alleviate this, it is standard in abstract interpretation to mandate the existence of a widening operator. A widening operator is similar to a join operator, with the additional property that repeated applications of it should not result in infinitely increasing chains.

$$
\begin{aligned}
\mathcal{E}[\![\cdot]\!]^\sharp &: \textit{expr} \times \mathbb{D}^\sharp \to \mathbb{V}_{\mathrm{D}} \times \mathbb{D}^\sharp \\
&\ \textit{provided by the domain}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{guard}_{\mathrm{D}} &: \mathbb{V}_{\mathrm{D}} \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp \quad \textit{provided by the domain} \\
\sqsubseteq_{\mathrm{D}} &\subseteq \mathbb{D}^\sharp \times \mathbb{D}^\sharp \quad \textit{provided by the domain} \\
\sqcup_{\mathrm{D}} &: \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp \quad \textit{provided by the domain} \\
\nabla_{\mathrm{D}} &: \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp \quad \textit{provided by the domain}
\end{aligned}
$$

$$
\begin{aligned}
[\![\cdot]\!]^\sharp &: \textit{stmt} \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp \\
[\![\mathbf{skip}]\!]^\sharp \mathbb{s} &= \mathbb{s} \\
[\![\mathrm{P}_1; \mathrm{P}_2]\!]^\sharp \mathbb{s} &= [\![\mathrm{P}_2]\!]^\sharp \left([\![\mathrm{P}_1]\!]^\sharp \mathbb{s}\right) \\
[\![x := e]\!]^\sharp \mathbb{s} &\quad \textit{provided by the domain} \\
[\![x := \mathbf{malloc}_t(e_{\mathrm{size}})]\!]^\sharp \mathbb{s} &\quad \textit{provided by the domain} \\
[\![*_t e_{\mathrm{loc}} := e]\!]^\sharp \mathbb{s} &\quad \textit{provided by the domain} \\
[\![\mathbf{if}\ e\ \mathbf{then}\ \mathrm{P}_1\ \mathbf{else}\ \mathrm{P}_2\ \mathbf{end}]\!]^\sharp \mathbb{s} &= [\![\mathrm{P}_1]\!]^\sharp (\mathbf{guard}_{\mathrm{D}}(\mathbb{v}_{\mathrm{true}}, \mathbb{s}_{\mathrm{true}})) \sqcup_{\mathrm{D}} [\![\mathrm{P}_2]\!]^\sharp (\mathbf{guard}_{\mathrm{D}}(\mathbb{v}_{\mathrm{false}}, \mathbb{s}_{\mathrm{false}})) \\
\text{where } &\begin{cases} (\mathbb{v}_{\mathrm{true}}, \mathbb{s}_{\mathrm{true}}) = \mathcal{E}[\![e]\!]^\sharp \mathbb{s} \\ (\mathbb{v}_{\mathrm{false}}, \mathbb{s}_{\mathrm{false}}) = \mathcal{E}[\![\neg e]\!]^\sharp \mathbb{s} \end{cases} \\
[\![\mathbf{while}\ e\ \mathbf{do}\ \mathrm{P}\ \mathbf{done}]\!]^\sharp \mathbb{s} &= \mathbf{guard}_{\mathrm{D}}(\mathcal{E}[\![\neg e]\!]^\sharp (\lim \mathrm{F})) \\
\text{where } \mathrm{F}(\mathrm{X}) &\overset{\mathrm{def}}{=} \mathrm{X} \nabla_{\mathrm{D}} \left(\mathbb{s} \sqcup_{\mathrm{D}} [\![\mathrm{P}]\!]^\sharp \left(\mathbf{guard}_{\mathrm{D}}(\mathcal{E}[\![e]\!]^\sharp \mathrm{X})\right)\right)
\end{aligned}
$$

Figure 3.8: Abstract semantics of While-memory programs

Definition 3.16 (Widening). An operator $\nabla : \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ is a widening operator in the abstract domain $(\mathbb{D}^\sharp, \sqsubseteq)$, if:

1. it computes upper bounds: $\forall x, y \in \mathbb{D}^\sharp, x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$

2. it enforces convergence: for any sequence $(x_i)_{i \in \mathbb{N}}$ in $\mathbb{D}^\sharp$, the sequence $(y_i)_{i \in \mathbb{N}}$ defined by:

$$
\begin{cases} y_0 = x_0 \\ y_{i+1} = y_i \nabla x_{i+1} \end{cases}
$$

stabilizes in finite time: $\exists \mathrm{N} \in \mathbb{N}, y_{\mathrm{N}+1} \sqsubseteq y_{\mathrm{N}}$.

We assume that each domain, in addition to the abstract join $\sqcup_{\mathrm{D},\Psi}$, provides a widening $\nabla_{\mathrm{D},\Psi}$ that fulfills the conditions above. Like with the abstract join, we write ∇_{D} when the renaming can be left implicit.

$$\begin{tikzcd}
\mathbb{D}^\sharp \arrow[r, "\llbracket \mathrm{P} \rrbracket^\sharp"] \arrow[d, "\gamma"'] & \mathbb{D}^\sharp \arrow[d, "\gamma"] \\
\mathcal{P}\big(\mathbb{S} \times (\mathbb{V}^\sharp \to \mathbb{V})\big) \arrow[d, "\mathrm{erase\_val}"'] & \mathcal{P}\big(\mathbb{S} \times (\mathbb{V}^\sharp \to \mathbb{V})\big) \arrow[d, "\mathrm{erase\_val}"] \\
\mathcal{P}(\mathbb{S}) \arrow[r, "\llbracket \mathrm{P} \rrbracket"'] & \mathcal{P}(\mathbb{S})
\end{tikzcd}$$

Figure 3.9: Graphical representation of semantics and concretizations for our generic abstract domain.

3.5 Soundness of the abstract semantics

The abstract semantics should be *sound* with respect to the concrete semantics, meaning that its concretization should contain all program executions of the concrete semantics.

To formalize this, since $\mathbb{D}^\sharp$ concretizes to elements of $\mathcal{P}(\mathbb{S} \times (\mathbb{V}^\sharp \to \mathbb{V}))$, we define an operator erase_val that strips away the valuation component, which is not present in the concrete semantics.

Definition 3.17 (Valuation erasure operator). The function erase_val : $\mathcal{P}\big(\mathbb{S} \times (\mathbb{V}^\sharp \to \mathbb{V})\big) \to \mathcal{P}(\mathbb{S})$ is defined as:

$$\mathrm{erase\_val}(\mathrm{X}) = \{s \mid (s, v) \in \mathrm{X}\}$$

Definition 3.18 (Soundness of the abstract semantics). If we let $\gamma' : \mathbb{D}^\sharp \to \mathcal{P}(\mathbb{S})$ denote the concretization that ignores the valuation component:

$$\gamma' = \mathrm{erase\_val} \circ \gamma$$

Then $\llbracket \cdot \rrbracket^\sharp$ is *sound* with respect to $\llbracket \cdot \rrbracket$ if and only if, for every program $\mathrm{P} \in$ *stmt*:

$$\forall \mathbb{d} \in \mathbb{D}^\sharp,\ (\llbracket \mathrm{P} \rrbracket \circ \gamma')(\mathbb{d}) \subseteq (\gamma' \circ \llbracket \mathrm{P} \rrbracket^\sharp)(\mathbb{d})$$

Soundness proofs typically proceed by induction on the syntax of While-memory. For each abstract domain that we introduce, we will justify the soundness of its abstract semantics.

Chapter 4

Physical types

Outline of the current chapter

In this chapter, we introduce *physical types*, which constitute the foundation of our memory analysis. In Section 4.1, we informally introduce physical types on an example. In Section 4.2, we define physical types and give them meaning in terms of the *heap structural properties* that they allow to express. We also prove some results that we will use in the next chapters to establish the soundness of our analysis. We then give a typed semantics of the While-memory language (Section 4.3), which will be over-approximated by type-based shape abstract domain in Chapter 5. Finally, we discuss possible extensions of physical types (Section 4.4).

4.1 Overview example and informal presentation

Throughout Chapters 4 to 6, we demonstrate the main features of our analysis on a running example. This example is a low-level implementation of a classical union-find structure. Union-find algorithms allow to efficiently represent and update the partition of a set of "nodes" into equivalence classes, by providing a fast operation to `merge` two equivalence classes. The representation combines the union-find structure based on chains of pointers to class representatives in reverse tree shapes, with circular, doubly-linked lists for efficient iteration over the elements of an equivalence class, and was inspired

```c
typedef struct uf {
  struct uf* parent;
} uf;

typedef struct dll {
  struct dll *prev; /* != null. */
  struct dll *next; /* != null. */
} dll;

typedef unsigned int node_kind;
typedef struct node {
  node_kind kind; /* kind <= 5. */
  dll dll;
  uf uf;
} node;

uf *uf_find(uf *x) {
  while(x->parent != 0) {
    uf *parent = x->parent;
    if(parent->parent == 0)
      return parent;
    x->parent = parent->parent;
    x = parent->parent;
  }
  return x;
}
```

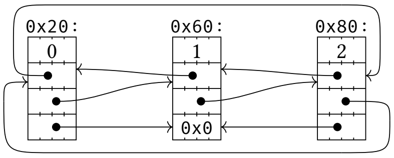
```c
void dll_union(dll *x, dll *y) {
  y->prev->next = x->next;
  x->next->prev = y->prev;
  x->next = y; y->prev = x;
}

void uf_union(uf *x, uf *y) {
  uf *rootx = uf_find(x);
  uf *rooty = uf_find(y);
  if(rootx != rooty)
    rootx->parent = rooty;
}

void merge(node *x, node *y) {
  dll_union(&x->dll, &y->dll);
  uf_union(&x->uf, &y->uf);
}

node *make(node_kind kind) {
  node *n = malloc(sizeof(node));
  n->kind = kind;
  n->dll.next = &n->dll;
  n->dll.prev = &n->dll;
  n->uf.parent = NULL;
  return n;
}
```

Figure 4.1: An algorithm for union-find and listing elements in a partition.



(a) Concrete state.

$$
\begin{aligned}
\mathcal{M}\ :\ \texttt{node\_kind} &\mapsto \{x\ :\ \texttt{word}_4 \mid x \leq 5\}\\
\texttt{uf} &\mapsto \texttt{uf}.(0)*\\
\texttt{dll} &\mapsto \{x\ :\ \texttt{dll}.(0)* \mid x \neq 0\} \times \{x\ :\ \texttt{dll}.(0)* \mid x \neq 0\}\\
\texttt{node} &\mapsto \texttt{node\_kind} \times \texttt{dll} \times \texttt{uf}
\end{aligned}
$$

(b) Physical types.

$$
\begin{array}{rcll}
\multicolumn{3}{l}{\forall \upsilon, \forall(\sigma, h, \mathcal{L}, \Gamma) \text{ well-typed state}, \forall v \text{ value }:} & \\
v \in \llparenthesis \texttt{node}.(0)* \rrparenthesis_{\mathcal{L},\upsilon} \wedge v \neq 0 & \Longrightarrow & v + 4 \in \llparenthesis \texttt{node}.(4) \rrparenthesis_{\mathcal{L},\upsilon} \wedge v + 4 \neq 0 & (1)\\
\llparenthesis \texttt{node}.(4)* \rrparenthesis_{\mathcal{L},\upsilon} & \subseteq & \llparenthesis \texttt{dll}.(0)* \rrparenthesis_{\mathcal{L},\upsilon} & (2)\\
v \in \llparenthesis \texttt{dll}.(0)* \rrparenthesis_{\mathcal{L},\upsilon} \wedge v \neq 0 & \Longrightarrow & h[v..v+4] \in \llparenthesis \{x\ :\ \texttt{dll}.(0)* \mid x \neq 0\} \rrparenthesis_{\mathcal{L},\upsilon} & (3)\\
\llparenthesis \texttt{uf}.(0)* \rrparenthesis_{\mathcal{L},\upsilon} & \cap & \llparenthesis \texttt{dll}.(0)* \rrparenthesis_{\mathcal{L},\upsilon} = \{0\} & (4)
\end{array}
$$

(c) Some structural invariants entailed by $\mathcal{M}$

Figure 4.2: Concrete state, physical types and example structural invariants.

by Kennedy [Ken07]. The whole code is presented in Figure 4.1. It is written in C and not in While-memory for the sake of readability; the correspondence between C and While-memory is covered in detail in Chapter 7, but this simple code can be trivially translated to While-memory by assuming that $\mathbb{X}$ is large enough to uniquely identify the local variables in each function, by translating field accesses to the corresponding pointer arithmetic (e.g. `x->next` becomes $*_4 x + 4$), and by inlining function calls. We assume a little-endian architecture with 4-bytes words.

Structures `uf` and `dll` respectively represent the union find and doubly linked list structures. Following a pattern common in low-level and system code [Bro09], the structure `node` comprises both sub-structures `uf` and `dll`. Function `uf_find` returns the representative of the class of an element and halves [TvL84] the paths to the root to speed up subsequent calls. Functions `dll_union` and `uf_union` respectively merge doubly linked-lists and union-finds. Last, function `merge` reunites the equivalence classes of two `node`s, and `make` creates a new node.

Figure 4.2a displays an example concrete state, with a class made of three nodes (and where the node at address `0x60` is the representative). Such states contain a very high degree of sharing due to the interleaved union-find and doubly-linked list structures. Moreover, this data representation is *unbounded*: for example, the number of objects of type `uf` traversed upon a call to `uf_find` can be arbitrarily large. Therefore, pointer analyses (which do not attempt to infer properties of heap objects that are not directly pointed to by program variables) require tricky and ad hoc adaptations regarding sensitivity to be precise, so as to divide heaps in regions of pointers with similar properties; these techniques are too imprecise to verify type or memory safety for C or assembly.

At the same time, shared data structures such as union-find are notoriously hard to handle for shape analysis abstractions, because they exhibit unstructured sharing, i.e. sharing that cannot be described in a local manner (not even inductively), and we are not aware of any successful shape analysis based verification for a structure similar to that of Figure 4.1.

One of our key contributions is to propose an abstract interpretation framework based on a semantic interpretation of physical types, that simultaneously verifies the preservation of type-based structural invariants, and uses these invariants to perform and improve the precision of the analysis. This contrasts with the usual method where syntactic type checking and type-based pointer analyses are separate analyses, each insufficiently precise to verify type safety for low-level languages like C or binary. The type-based structural invariant implies dividing the heap into partitions, to which are attached flow-insensitive information, allowing for efficient static analysis operations. To further improve precision, our analysis is strengthened by flow-sensitive points-to predicates (Chapter 6).

$$\begin{array}{rcll}
\mathbb{T} \ni t & ::= & \texttt{word}_n & \textit{(base type of size n bytes)} \\
 & | & \mathtt{n} & \textit{(type name, } \mathtt{n} \in \mathcal{N}\textit{)} \\
 & | & t_a* & \textit{(pointer type)} \\
 & | & t \times t & \textit{(product type)} \\
 & | & \{x : t \mid pred(x)\} & \textit{(type refined with predicate)} \\
 & | & t[s] & \textit{(array type of size } s \in \mathbb{N}\textit{)} \\
 & & & \\
\mathbb{T}_A \ni t_a & ::= & t.(k) & \textit{(address type with offset, } k \in \mathbb{N}\textit{)} \\
 & & & \\
pred(x) & ::= & predexpr(x) \bowtie predexpr(x) & \textit{(comparison, } \bowtie \in \{\leq, <, =, \neq\}\textit{)} \\
 & | & b & \textit{(boolean constant, } b \in \mathbb{B}\textit{)} \\
 & | & \neg\, pred(x) & \textit{(negation)} \\
 & | & pred(x) \wedge pred(x) & \textit{(conjunction)} \\
 & & & \\
predexpr(x) & ::= & x & \textit{(unknown of the predicate)} \\
 & | & c & \textit{(numeric constant, } c \in \mathbb{V}\textit{)} \\
 & | & \mathbb{v} & \textit{(symbolic variable, } \mathbb{v} \in \mathbb{V}^{\sharp}\textit{)} \\
 & | & predexpr(x) \diamond predexpr(x) & \textit{(binary op., } \diamond \in \{+, -, \times, \&, |, \cdots\}\textit{)}
\end{array}$$

Figure 4.3: Grammar of physical types.

Let us examine the types and structural invariants of our example code. The types are given in Figure 4.2b. They derive in part from the C types, as these types contain a lot of information about the layout of the data-structures, at least when programs preserve types (which the C semantics does not guarantee). Intuitively, $\texttt{uf}.(0)*$ denotes a pointer to the base address of a well-formed `uf` instance. This pointer is allowed to be null. In the case of `dll`, on the contrary, pointer types are *refined* with a predicate stating that the pointers should not be null, giving lieu to the type $\{x : \texttt{dll}.(0)* \mid x \neq 0\}$. Type `node_kind` is a generic 4-bytes integer, $\texttt{word}_4$, refined with a predicate restraining its possible values. Finally, these types can be assembled via the product sign × to form composite types, akin to C `struct`s. Thus, these types can be more precise than C types, although C types can be translated to our type language. But they are less precise than shape invariants, as they cannot represent the relation between different elements of a same type: for example, our `dll` type would accept not only doubly-linked lists, but also a binary tree with leaves pointing to the root.

These types entail structural invariants, some of which are presented in Figure 4.2c, that a well-typed state must fulfill. These invariants relate types, interpreted as sets of values: $(\!| t |\!)_{\mathcal{L},\mathbb{v}}$ represents the set of values for type t. Equation (1) relates adjacent addresses; Equation (2) describes a subtyping relationship; Equation (3) relates the type of an address with its contents; and Equation (4) describes a partitioning of the heap in distinct regions. We will now proceed to formally defining these structural invariants.

4.2 Definitions

Physical types describe the low-level memory representation of values, down to the byte level, as well as the points-to relations between values. Their grammar is defined in Figure 4.3. In addition to the base scalar types $\texttt{word}_n$, there are *pointer types*, *product types*, *refinement types* and *arrays*.

Types can be given *names* —from a finite set of names $\mathcal{N}$— via a mapping $\mathcal{M} : \mathcal{N} \to \mathbb{T}$ from type names to types. This mapping is fixed during the execution of a program. In what follows, we will always denote types in italic font such as t_1, while type names will be written in typewriter font: $\mathtt{n}_1$. Type names have two uses: first, they break cycles in the definition of recursive types; second, they distinguish types otherwise structurally equal (i.e. they allow the type system to be nominal), and in particular pointers to two structurally equal types with different names will not alias. We shall only consider "well-founded" recursive types, in which recursion cycles can only be created through pointers, as is the case in C types. More precisely:

Definition 4.1 (Physical types). The set of physical types is $\mathbb{T}$, defined by induction in Figure 4.3. Let $\mathcal{M} : \mathcal{N} \to \mathbb{T}$ be a mapping from type names to types. Let $\mathrm{emb} : \mathbb{T} \to \mathcal{P}(\mathbb{T})$ be a function that computes the types "embedded" in another type, defined by:

$$\begin{aligned}
\mathrm{emb}(\mathtt{word}_n) &= \{\mathtt{word}_n\}\\
\mathrm{emb}(\mathtt{n}) &= \{\mathtt{n}\}\\
\mathrm{emb}(t.(k)*) &= \emptyset\\
\mathrm{emb}(\{x : t \mid p(x)\}) &= \mathrm{emb}(t)\\
\mathrm{emb}(t_1 \times t_2) &= \mathrm{emb}(t_1) \cup \mathrm{emb}(t_2)\\
\mathrm{emb}(t[s]) &= \mathrm{emb}(t)
\end{aligned}$$

Let $\mathcal{G}_{\mathcal{M}}$ be the smallest directed graph that contains, for every mapping $\mathtt{n} \mapsto t$ in $\mathcal{M}$, a node labelled $\mathtt{n}$ and for each element $u \in \mathrm{emb}(t)$, a node labelled u and an edge from $\mathtt{n}$ to u. $\mathcal{M}$ is said well-formed if $\mathcal{G}_{\mathcal{M}}$ is acyclic.

For instance, if there is a type `circ` defined by $\mathcal{M}(\mathtt{circ}) = \mathtt{word}_4 \times \mathtt{circ}$, the set of type definitions is not valid: a type should not contain itself. We will only consider well-formed type mappings. As a consequence, we will be able to write definitions and proofs by induction on the grammar of types. In the inductions that follow, unless otherwise stated, the strictly decreasing measure ensuring well-foundedness will be the height of a type in its connected component in $\mathcal{G}_{\mathcal{M}}$.

The *address type* $t.(k) \in \mathbb{T}_A$ represents the k-th byte in a value of type t. A pointer type such as $t.(k)*$ should be interpreted as either: the address of the k-th byte of a value of type t, *or* the value 0. Types can be *refined* with a predicate. Note that a type refined by a predicate makes use of a local variable x that denotes the value of this type and is meant to be constrained in the matching *pred*(x) predicate, which is why grammar entries *predexpr*(x) and *pred*(x) take a variable as parameter. Predicates are quantifier-free, unary predicates with numeric and logical operators. They may refer to *symbolic variables*, which are existentially quantified variables, i.e., these variables represent fixed, constant values. For convenience, we use the set of symbolic variables $\mathbb{V}^\sharp$ defined in Chapter 3.

The product type $t_1 \times t_2$ represents values consisting of a value of type t_1 and a value of type t_2 as they appear side by side in memory, e.g. in a C `struct`. Similarly, array types of the form $t[n]$ represent contiguous sequences of n values of type t.

Each type has a size (in bytes) given by the function $\mathrm{size} : \mathbb{T} \to \mathbb{N}$.

$$\begin{aligned}
\mathrm{size}(\mathtt{word}_n) &= n\\
\mathrm{size}(\mathtt{n}) &= \mathrm{size}(\mathcal{M}(\mathtt{n}))\\
\mathrm{size}(t_a*) &= \mathcal{W}\\
\mathrm{size}(\{x : t \mid p(x)\}) &= \mathrm{size}(t)\\
\mathrm{size}(t_1 \times t_2) &= \mathrm{size}(t_1) + \mathrm{size}(t_2)\\
\mathrm{size}(t[s]) &= s \cdot \mathrm{size}(t)
\end{aligned}$$

C compilers sometimes introduce padding bytes in `struct`s to align their sizes on a multiple of 4 or

8. Since product types express the low-level representation of values in memory, they should translate these padding bytes explicitly, e.g. `struct { t a; u b; };` would be translated as $t \times \texttt{word}_2 \times u$ if there are two padding bytes between the fields.

4.2.1 Labellings

The typed semantics will attribute types to variables, but also to memory regions. We formalize this as a mapping from memory addresses to *address types*, which we call a labelling.

Definition 4.2 (Labelling). A labelling $\mathcal{L}$ is a function of $\mathbb{A} \to \mathbb{T}_A$ such that each tagging of a region with a type is whole and contiguous, i.e. for all types $t \in \mathbb{T}$, for all address $a \in \mathbb{A}$, if we define $n = \text{size}(t)$, then for all $k \in [0,\ n-1]$,

$$\mathcal{L}(a+k) = t.(k) \implies \begin{cases} \mathcal{L}(a) = t.(0) \\ \mathcal{L}(a+1) = t.(1) \\ \vdots \\ \mathcal{L}(a+n-1) = t.(n-1) \end{cases}$$

The set of labellings is denoted $\mathbb{L}$.

Example 4.1. Data structures in Figure 4.2a can be labelled like so:

$$\begin{array}{llllllll} \mathcal{L}: & \texttt{0x20} \mapsto \texttt{node}.(0) & \texttt{0x21} \mapsto \texttt{node}.(1) & \ldots & \texttt{0x2c} \mapsto \texttt{node}.(15) \\ & \texttt{0x60} \mapsto \texttt{node}.(0) & \texttt{0x61} \mapsto \texttt{node}.(1) & \ldots & \texttt{0x6c} \mapsto \texttt{node}.(15) \\ & \texttt{0x80} \mapsto \texttt{node}.(0) & \texttt{0x81} \mapsto \texttt{node}.(1) & \ldots & \texttt{0x8c} \mapsto \texttt{node}.(15) \end{array}$$

4.2.2 Subtyping between address types

Several address types may make sense for a single memory cell, but some are more precise than others:

Example 4.2. In the labelling of Example 4.1, the type of address `0x24` is $\texttt{node}.(4)$. But in some sense, it is also a $\texttt{dll}.(0)$, because `node` contains a `dll` at offset 4. The former is more precise, however: all memory cells that contain a $\texttt{node}.(4)$ contain a $\texttt{dll}.(0)$, but the converse is not true. We say that $\texttt{node}.(4)$ is a *subtype* of $\texttt{dll}.(0)$. Intuitively, "$t.(n)$ is a subtype of $u.(m)$" means that t "contains" a u somewhere in its structure.

Definition 4.3 (Weakening function w). The function $w\ :\ \mathbb{T}_A \rightharpoonup \mathbb{T}_A$ is defined inductively by:

$$\begin{aligned} w(\mathsf{n}.(k)) &= t.(k) && \text{if } 0 \le k < \text{size}(t),\ \text{where } t = \mathcal{M}(\mathsf{n}) \\ w((t_1 \times t_2).(k)) &= t_1.(k) && \text{if } 0 \le k < \text{size}(t_1) \\ w((t_1 \times t_2).(\text{size}(t_1)+k)) &= t_2.(k) && \text{if } 0 \le k < \text{size}(t_2) \\ w(t[s].(q \cdot \text{size}(t)+k)) &= t.(k) && \text{if } 0 \le q < s \text{ and } 0 \le k < \text{size}(t) \end{aligned}$$

Since all the cases above are disjoint, this constitutes a valid definition of the partial function w, in the sense that every type of $\mathbb{T}_A$ has at most one image.

Definition 4.4 (Weakening of an address type). Given $t.(i), u.(j) \in \mathbb{T}_A$, we say that $t.(i)$ *is the weakening of* $u.(j)$ if and only if $t.(i) = w(u.(j))$.

Definition 4.5 (Subtyping between address types). Let the relation $\preceq \subseteq \mathbb{T}_A \times \mathbb{T}_A$ be the transitive, reflexive closure of the relation $\{(\tau, \upsilon) \mid \upsilon = w(\tau)\}$.

Recall that $\mathcal{M} : \mathcal{N} \rightarrow \mathbb{T}$ maps type names to types, and is fixed for a given program. The choice of $\mathcal{M}$ influences the analysis. For instance, $\texttt{dll} \mapsto \{x : \texttt{dll}.(0)* \mid x \neq 0\} \times \{x : \texttt{dll}.(0)* \mid x \neq 0\}$ and $\texttt{dll} \mapsto \texttt{dll}.(0)* \times \texttt{dll}.(0)*$ may both be sensible choices to translate the C structure `dll`, but the former places stronger constraints on the inferred heap.

Let us now prove a useful property of the subtyping relation, namely, the fact that its graph is a forest.

Lemma 4.1. *For all address types $\tau, \upsilon, \phi \in \mathbb{T}_A$,*

$$\phi \preceq \tau \wedge \phi \preceq \upsilon \implies \tau \preceq \upsilon \vee \upsilon \preceq \tau$$

Proof. Let us assume that $\phi \preceq \tau$ and $\phi \preceq \upsilon$. Then there exists two finite chains $\phi \preceq \tau_1 \preceq \tau_2 \preceq \cdots \preceq \tau_n \preceq \tau$, and $\phi \preceq \upsilon_1 \preceq \upsilon_2 \preceq \cdots \preceq \upsilon_m \preceq \upsilon$ in which each element is the weakening of the previous one. Either one chain is included in the other, or not. If yes, then either $\tau \preceq \upsilon$ or $\upsilon \preceq \tau$, which ends the proof. If not, then let p be the minimal index such that $\tau_p \neq \upsilon_p$. But $\tau_p = w(\tau_{p-1})$ and $\upsilon_p = w(\tau_{p-1})$, so $\tau_{p-1} = \upsilon_{p-1}$, which contradicts the minimality of p. □

Definition 4.6 (Containment between types). For all types $t, u \in \mathbb{T}$, it is said that t contains u if:

$$\exists i \in [0, \text{size}(t)[, \forall k \in [0, \text{size}(u)[, t.(i+k) \preceq u.(k)$$

Lemma 4.2. *Given two types $t, u \in \mathbb{T}$, suppose that an address type in u is the weakening of an address type in t, i.e. suppose that there exist $i \in [0, \text{size}(t)[$ and $j \in [0, \text{size}(u)[$, such that $u.(j) = w(t.(i))$. Then:*

$$\forall k \in [0, \text{size}(u)[, u.(k) = w(t.(i - j + k))$$

Proof. By case distinction on t:

- if $t = \text{s} \in \mathcal{N}$: then by Definition 4.4, $w(t.(i)) = w(\text{s}.(i)) = (\mathcal{M}(s)).(i) = u.(j)$. Therefore $u = \mathcal{M}(s)$ and $i = j$, and still by Definition 4.4:

$$\forall k \in [0, \text{size}(u)[, w(t.(k)) = w(t.(i - j + k)) = u.(k)$$

- if $t = t_1 \times t_2$ and $0 \leq i < \text{size}(t_1)$, then $w(t.(i)) = t_1.(i) = u.(j)$. Therefore $u = t_1$ and $i = j$, and:

$$\forall k \in [0, \text{size}(u)[, w(t.(k)) = w(t.(i - j + k)) = u.(k)$$

- if $t = t_1 \times t_2$ and $\text{size}(t_1) \leq i < \text{size}(t)$, then by letting $m = i - \text{size}(t_1)$, Definition 4.4 applies: $w(t.(i)) = t_2.(m) = u.(j)$. Therefore $u = t_2$ and $j = m = i - \text{size}(t_1)$, and:

$$\forall k \in [0, \text{size}(u)[, w(t.(k)) = w(t.(i - j + k)) = u.(k)$$

- The other cases are similar.

□

Lemma 4.3. *Given two types of $t, u \in \mathbb{T}$, if any of the address types of t is a subtype of an address type u, i.e. if:*

$$\exists i \in [0, \text{size}(t)[, \exists j \in [0, \text{size}(u)[, t.(i) \preceq u.(j)$$

then t contains u.

Proof. Assume the existence of i and j such that the above is true. Then, there exist a finite chain of the form:

$$t.(i) \preceq t_1.(i_1) \preceq t_2.(i_2) \preceq \cdots \preceq t_n.(i_n) \preceq u.(j)$$

where each $t_{k+1}.(i_{k+1})$ is the weakening of $t_k.(i_k)$. This chain is finite because we only consider finite types that do not strictly contain themselves (Definition 4.1). Moreover, it is easy to show that for all $k \in \{0, \dots, n\}$, i_k is within the bounds of t_k (by Definition 4.4). Therefore, we can use Lemma 4.2 to show by straightforward induction that t contains t_1, which contains t_2, and so on. By transitivity of "contains", t contains u. □

We now show that address subtyping is an over-approximation of aliasing relations. This will also enable us to verify the safety of memory writes by verifying that they preserve the typing of the heap.

Definition 4.7 (Set of addresses covered by a type). Let $\mathcal{L} \in \mathbb{L}$. The set of addresses covered by a type $t \in \mathbb{T}$ is:

$$\mathrm{addr}_{\mathcal{L}}(t) = \bigcup_{i=0}^{\mathrm{size}(t)-1} \{a \in \mathbb{A} \mid \mathcal{L}(a) \preceq t.(i)\}$$

Theorem 4.4. *Let $t, u \in \mathbb{T}$. Either t and u cover disjoint regions, or one contains the other, i.e.:*

$$\mathrm{addr}_{\mathcal{L}}(t) \cap \mathrm{addr}_{\mathcal{L}}(u) \neq \emptyset \implies t \textit{ contains } u \textit{ or } u \textit{ contains } t$$

Proof. Straightforward consequence of Lemmas 4.1 and 4.3. □

Address types can therefore be used to soundly infer absence of aliasing, in a way similar to Type-based Alias Analysis [DMM98] but on pointer types with offsets rather than on value types.

Example 4.3. The type `uf` does not contain `dll`, nor does `dll` contain `uf`. This implies that the memory regions covered by these two types are disjoint. In Example 4.1, we have $\mathrm{addr}_{\mathcal{L}}(\texttt{uf}) = [\texttt{0x2c}, \texttt{0x2f}] \cup [\texttt{0x6c}, \texttt{0x6f}] \cup [\texttt{0x8c}, \texttt{0x8f}]$, while $\mathrm{addr}_{\mathcal{L}}(\texttt{dll}) = [\texttt{0x24}, \texttt{0x2b}] \cup [\texttt{0x64}, \texttt{0x6b}] \cup [\texttt{0x84}, \texttt{0x8b}]$.

4.2.3 Types as sets of values

We now give the meaning of types in terms of an interpretation function. The interpretation of a type depends on a labelling, and on a valuation of the symbolic variables $\mathcal{v} : \mathbb{V}^\# \to \mathbb{V}$. Given such a valuation and an argument value, a predicate can be evaluated to a truth value in a straightforward way. We denote $\mathrm{eval}_{\mathcal{v}} : \mathit{pred} \times \mathbb{V} \to \mathbb{B}$ this evaluation function.

Definition 4.8 (Interpretation of a type). The interpretation operator with respect to a labelling $\mathcal{L}$ and a valuation $\mathcal{v} \in \mathbb{V}^\# \to \mathbb{V}$, denoted $(\!|\cdot|\!)_{\mathcal{L},\mathcal{v}} : \mathbb{T} \to \mathcal{P}(\mathbb{N})$, is defined by:

$$\begin{aligned}
(\!|\texttt{word}_n|\!)_{\mathcal{L},\mathcal{v}} &= \mathbb{V}_n \\
(\!|\mathsf{n}|\!)_{\mathcal{L},\mathcal{v}} &= (\!|\mathcal{M}(n)|\!)_{\mathcal{L},\mathcal{v}} \\
(\!|t_1 \times t_2|\!)_{\mathcal{L},\mathcal{v}} &= \left\{v_1 :: v_2 \mid v_1 \in (\!|t_1|\!)_{\mathcal{L},\mathcal{v}}, v_2 \in (\!|t_2|\!)_{\mathcal{L},\mathcal{v}}\right\} \\
(\!|t[s]|\!)_{\mathcal{L},\mathcal{v}} &= \{v_0 :: v_1 :: \cdots :: v_{s-1} \mid v_0, \dots, v_{s-1} \in (\!|t|\!)_{\mathcal{L},\mathcal{v}}\} \\
(\!|\{x : t \mid p(x)\}|\!)_{\mathcal{L},\mathcal{v}} &= \{v \in (\!|t|\!)_{\mathcal{L},\mathcal{v}} \mid \mathrm{eval}_{\mathcal{v}}(p, v) = \mathrm{true}\} \\
(\!|t.(k){*}|\!)_{\mathcal{L},\mathcal{v}} &= \{0\} \cup \{a \in \mathbb{A} \mid \mathcal{L}(a) \preceq t.(k)\}
\end{aligned}$$

We now define another order relation, this time on types, that expresses a subtyping at the value level:

Definition 4.9 (Value-centered subtyping). The relation $\preceq_{\mathbb{T}} \subseteq \mathbb{T} \times \mathbb{T}$ is defined inductively by the following rules:

$$\frac{}{t \preceq_{\mathbb{T}} \texttt{word}_{\mathrm{size}(t)}} \qquad \frac{}{\{x : t \mid p(x)\} \preceq_{\mathbb{T}} t} \qquad \frac{t.(i) \preceq u.(j)}{t.(i)* \preceq_{\mathbb{T}} u.(j)*}$$

From this definition follows that $\llparenthesis \cdot \rrparenthesis_{\mathcal{L},\mathcal{v}}$ is monotone with respect to the $\preceq_{\mathbb{T}}$ relation:

Lemma 4.5 (Monotonicity). *Given $\mathcal{L} \in \mathbb{L}$, $\mathcal{v} : \mathbb{V}^{\#} \to \mathbb{V}$, given two types t and u in $\mathbb{T}$,*

$$t \preceq_{\mathbb{T}} u \implies \llparenthesis t \rrparenthesis_{\mathcal{L},\mathcal{v}} \subseteq \llparenthesis u \rrparenthesis_{\mathcal{L},\mathcal{v}}$$

As a consequence, $\preceq_{\mathbb{T}}$ is a subtyping in the sense of Liskov [LW94], i.e. if $t \preceq_{\mathbb{T}} u$ then all properties of values of type t hold for values of type u.

Example 4.4. In the labelling of Example 4.1, $\llparenthesis \texttt{node}.(4)* \rrparenthesis_{\mathcal{L},\mathcal{v}}$ is the set $\{\texttt{0x0}, \texttt{0x24}, \texttt{0x64}, \texttt{0x84}\}$, which is the same set as $\llparenthesis \texttt{dll}.(0)* \rrparenthesis_{\mathcal{L},\mathcal{v}}$.

The following lemma states that if a value has a compound type u, modifying a member of type t of that compound type preserves typing as long as the new sub-value is in $\llparenthesis t \rrparenthesis_{\mathcal{L},\mathcal{v}}$.

Example 4.5 (Modifying a field of a value of type `node`). Given $\mathcal{L}$ a labelling and $\mathcal{v}$ a valuation, consider a value $v \in \llparenthesis \texttt{node} \rrparenthesis_{\mathcal{L},\mathcal{v}}$. Because $\texttt{node}.(4) \preceq \texttt{dll}.(0)$, it is guaranteed that replacing the `dll` field of v, i.e. replacing bytes 4 to 11 of v with a value $w \in \llparenthesis \texttt{dll} \rrparenthesis_{\mathcal{L},\mathcal{v}}$ results in a value that is still in the interpretation of `node`.

More formally:

Lemma 4.6. *Given $u.(i), t.(0) \in \mathbb{T}_{\mathrm{A}}$, given $\mathcal{v} : \mathbb{V}^{\#} \to \mathbb{V}$ a valuation and $\mathcal{L}$ a labelling, suppose $u.(i) \preceq t.(0)$. For all $v, v' \in \mathbb{V}_{\mathrm{size}(t)}$, for all $a \in \mathbb{V}_i, b \in \mathbb{V}_{\mathrm{size}(u)-i-\mathrm{size}(t)}$:*

$$a :: v :: b \in \llparenthesis u \rrparenthesis_{\mathcal{L},\mathcal{v}} \land v' \in \llparenthesis t \rrparenthesis_{\mathcal{L},\mathcal{v}} \implies a :: v' :: b \in \llparenthesis u \rrparenthesis_{\mathcal{L},\mathcal{v}}$$

Proof. By induction on the structure of u.

- If $u = \texttt{word}_n$: then necessarily from the subtyping definition (Definition 4.5) $u = t$ and $i = 0$; thus $a, b \in \mathbb{V}_0$ (i.e. a and b are empty bit vectors) and the result trivially holds.
- If $u = t_0.(i)*$: same proof as the previous case.
- If $u = \{x : t_0 \mid p(x)\}$: same proof as the previous case.
- If $u = \mathsf{n} \in \mathcal{N}$: then $\llparenthesis u \rrparenthesis_{\mathcal{L},\mathcal{v}} = \llparenthesis \mathcal{M}(u) \rrparenthesis_{\mathcal{L},\mathcal{v}}$, and the induction hypothesis on $\mathcal{M}(u)$ yields the result.
- If $u = t_1 \times t_2$: then either $t_1.(i) \preceq t.(0)$ or $t_2.(i - \mathrm{size}(t_1)) \preceq t.(0)$. Let us treat the first case, the second being similar. Let b_1 denote the bit vector consisting in the first $\mathrm{size}(t_1) - i - \mathrm{size}(t)$ bits of b, and b_2 the remaining part of b. The induction hypothesis on t_1 yields $a :: v' :: b_1 \in \llparenthesis t_1 \rrparenthesis_{\mathcal{L},\mathcal{v}}$, from which $a :: v' :: b_1 :: b_2 = a :: v' :: b \in \llparenthesis u \rrparenthesis_{\mathcal{L},\mathcal{v}}$.
- If $u = t_0[n]$: The proof for arrays is quite similar to the product case.

□

4.3 Typed semantics

We now define a typed semantics for While-memory. This semantics is conservative in that it excludes some executions of the untyped semantics $\llbracket \cdot \rrbracket$. Like the untyped semantics, it is not computable in general. Its goal is to serve as a step towards building an analysis that verifies the preservation of typing invariants: we will show that, by enforcing the well-typedness of operations, a non-trivial invariant on the state can be preserved, namely, the "structural invariants" that we informally presented in Section 4.1. Our shape abstract domain that we introduce in Chapter 5 uses this principle.

The set of store typings is $\mathbb{X} \to \mathbb{T}$ and the set of typed states is $\mathbb{S}_t = \Sigma \times (\mathbb{X} \to \mathbb{T}) \times \mathbb{H} \times \mathbb{L}$. We introduce the notion of well-typed state, in which the values in memory are consistent with their type labels, and values in the variable store are consistent with the *interpretation* of their types.

Definition 4.10. A state $(\sigma, \Gamma, h, \mathcal{L})$ is said to be *well typed under valuation* υ if

1. The labelling is consistent with heap values: for every address $a \in \mathbb{A}$, if there exists a type t such that $\mathcal{L}(a) = t.(0)$, then $h[a..(a+\mathrm{size}(t))] \in \llparenthesis\, t \,\rrparenthesis_{\mathcal{L},\upsilon}$;

2. Variables are well-typed: for all variable name $x \in \mathbb{X}$, $\sigma(x) \in \llparenthesis\, \Gamma(x) \,\rrparenthesis_{\mathcal{L},\upsilon}$.

4.3.1 Typed semantics of expressions

Typing of expressions aims at proving that the evaluation of an expression will either return a value consistent with the type or a runtime error, such as division by zero or null pointer dereference. We do not attempt to use types to prevent all runtime errors; for instance, well-typed computations are not necessarily free from divisions by zero; the analysis must discharge such verifications separately. Given a valuation υ, a store σ, a heap h, a labeling $\mathcal{L}$, a typing of variables Γ, an expression e, and a type t, we write $(\sigma, h, \mathcal{L}, \Gamma) \vdash_{\upsilon} e : t$ when expression e can be given type t in the typing state $(\sigma, h, \mathcal{L}, \Gamma)$.

The typing rules for expressions are given in Figure 4.4.

The type of variables is resolved by Γ (rule Env). Constants are given type $\mathtt{word}_n$ (rule Const).

Memory reads and pointer arithmetic are typed using corresponding offset calculation over physical types (rules Load, AddR, AddL and SubR). If one of the operands of an addition is a non-null pointer, and the other operand is such that the resulting offset is not out of bounds, then the resulting type is a pointer with the new offset. Note that this is only one of the possible choices: another semantics could be defined, in which pointer offsets are allowed to go out of bounds, and the bound checking is only made upon dereferencing. Pointer arithmetic can also be performed with subtractions (rule SubR), but the pointer can only be the left-hand-side operand. For both additions and subtractions, when the premises of rules AddR, AddL or SubR do not apply, then the Binop rule applies.

The Binop rule states that other binary operators result in the generic type $\mathtt{word}_n$.

Finally, rule SubVal is an *upcasting* rule in that it states that an expression of some type t is also judged to have all the supertypes of t (in the sense of $\preceq_{\mathbb{T}}$); rule Refine states that the type of an expression can be refined with all predicates verified by the expression value —a form of *downcasting*. Due to these two rules, an expression will generally have several types.

This typing is sound in the following sense:

Theorem 4.7 (Soundness of typing of expressions). *For all expressions e of While-memory, for all valuations $\upsilon \in \mathbb{V}^{\#} \to \mathbb{V}$, states $s = (\sigma, \Gamma, h, \mathcal{L}) \in \mathbb{S}_t$ and types $t \in \mathbb{T}$, if s is well typed under υ and $s \vdash_{\upsilon} e : t$ and $\mathcal{E}[\![e]\!](\sigma, h) = v$, then $v \in \llparenthesis\, t \,\rrparenthesis_{\mathcal{L},\upsilon}$.*

Proof. By induction on the typing rules.

- *rule Env:* $\mathcal{E}[\![e]\!](\sigma, h) = \sigma(x)$ and $t = \Gamma(x)$. Therefore since s is well typed, $v \in \llparenthesis\, t \,\rrparenthesis_{\mathcal{L},\upsilon}$.
- *rule Const:* $\llparenthesis\, \mathtt{word}_n \,\rrparenthesis_{\mathcal{L},\upsilon} = \mathbb{V}_n$, thus $v \in \llparenthesis\, t \,\rrparenthesis_{\mathcal{L},\upsilon}$.
- *rule Load:* Then by the induction hypothesis if $\mathcal{E}[\![e_{\mathrm{loc}}]\!](\sigma, h)$ is defined then $\mathcal{E}[\![e_{\mathrm{loc}}]\!](\sigma, h) = a \in \llparenthesis\, u.(i)\star \,\rrparenthesis_{\mathcal{L},\upsilon}$, where $u.(i) \preceq t.(0)$. If $\mathcal{E}[\![e_{\mathrm{loc}}]\!](\sigma, h)$ is undefined then so is v, which ends the proof. Otherwise, we know that $a \in \llparenthesis\, u.(i)\star \,\rrparenthesis_{\mathcal{L},\upsilon} \subseteq \llparenthesis\, t.(0)\star \,\rrparenthesis_{\mathcal{L},\upsilon} = \{a \in \mathbb{A} \mid \mathcal{L}(a) \preceq t.(0)\} \cup \{0\}$. But we also have $a \neq 0$. Therefore $\mathcal{L}(a) \preceq t.(0)$, and by well-typedness of s, $v = h[a..a + \ell] \in \llparenthesis\, t \,\rrparenthesis_{\mathcal{L},\upsilon}$.

$$\frac{}{(\sigma,\Gamma,h,\mathcal{L}) \vdash_v x : \Gamma(x)}\ \textsc{Env} \qquad \frac{c \in \mathbb{V}_n}{(\sigma,\Gamma,h,\mathcal{L}) \vdash_v c : \texttt{word}_n}\ \textsc{Const} \qquad \frac{t \preceq_{\mathbb{T}} u \qquad (\sigma,\Gamma,h,\mathcal{L}) \vdash_v e : t}{(\sigma,\Gamma,h,\mathcal{L}) \vdash_v e : u}\ \textsc{SubVal}$$

$$\frac{(\sigma,\Gamma,h,\mathcal{L}) \vdash_v e : t \qquad \mathrm{eval}_v(p, \mathcal{E}[\![e]\!](\sigma,h)) = \mathrm{true}}{(\sigma,\Gamma,h,\mathcal{L}) \vdash_v e : \{x : t \mid p(x)\}}\ \textsc{Refine}$$

$$\frac{(\sigma,\Gamma,h,\mathcal{L}) \vdash_v e : u.(i)* \qquad u.(i) \preceq t.(0) \qquad \mathrm{size}(t) = \ell \qquad \mathcal{E}[\![e]\!](\sigma,h) \neq 0}{(\sigma,\Gamma,h,\mathcal{L}) \vdash_v *_\ell e : t}\ \textsc{Load}$$

$$\frac{\begin{array}{c}(\sigma,\Gamma,h,\mathcal{L}) \vdash_v e_1 : t.(i)* \\ (\sigma,\Gamma,h,\mathcal{L}) \vdash_v e_2 : \texttt{word}_{\mathcal{W}} \qquad \mathcal{E}[\![e_1]\!](\sigma,h) = v_1 \neq 0 \qquad \mathcal{E}[\![e_2]\!](\sigma,h) = v_2 \qquad 0 \leq i + v_2 < \mathrm{size}(t)\end{array}}{(\sigma,\Gamma,h,\mathcal{L}) \vdash_v e_1 + e_2 : t.(i+v_2)*}\ \textsc{AddR}$$

$$\frac{\begin{array}{c}(\sigma,\Gamma,h,\mathcal{L}) \vdash_v e_1 : \texttt{word}_{\mathcal{W}} \\ (\sigma,\Gamma,h,\mathcal{L}) \vdash_v e_2 : t.(i)* \qquad \mathcal{E}[\![e_1]\!](\sigma,h) = v_1 \qquad \mathcal{E}[\![e_2]\!](\sigma,h) = v_2 \neq 0 \qquad 0 \leq v_1 + i < \mathrm{size}(t)\end{array}}{(\sigma,\Gamma,h,\mathcal{L}) \vdash_v e_1 + e_2 : t.(v_1+i)*}\ \textsc{AddL}$$

$$\frac{\begin{array}{c}(\sigma,\Gamma,h,\mathcal{L}) \vdash_v e_1 : t.(i)* \\ (\sigma,\Gamma,h,\mathcal{L}) \vdash_v e_2 : \texttt{word}_{\mathcal{W}} \qquad \mathcal{E}[\![e_1]\!](\sigma,h) = v_1 \neq 0 \qquad \mathcal{E}[\![e_2]\!](\sigma,h) = v_2 \qquad 0 \leq i - v_2 < \mathrm{size}(t)\end{array}}{(\sigma,\Gamma,h,\mathcal{L}) \vdash_v e_1 - e_2 : t.(i-v_2)*}\ \textsc{SubR}$$

$$\frac{(\sigma,\Gamma,h,\mathcal{L}) \vdash_v e_1 : \texttt{word}_n \qquad (\sigma,\Gamma,h,\mathcal{L}) \vdash_v e_2 : \texttt{word}_n \qquad \diamond \in \{+,-,\times,/,\cdots\}}{(\sigma,\Gamma,h,\mathcal{L}) \vdash_v e_1 \diamond e_2 : \texttt{word}_n}\ \textsc{Binop}$$

Figure 4.4: Typing rules for WHILE-MEMORY expressions.

- *rule ADDR*: In this case, $t = u.(i+v_2)$ for some $u \in \mathbb{T}$, and $v = \mathcal{E}[\![e_1+e_2]\!](\sigma,h) = v_1+v_2$. By induction hypothesis $v_1 = \mathcal{E}[\![e_1]\!](\sigma,h) \in (\!| u.(i)* |\!)_{\mathcal{L},v}$. Similarly to the above case, since $v_1 \neq 0$ we deduce that $\mathcal{L}(v_1) \preceq u.(i)$. And since $0 \leq i + v_2 < \mathrm{size}(u)$, Lemma 4.3 yields $\mathcal{L}(v_1 + v_2) \preceq u.(i + v_2)$.
- *rule ADDL*: Symmetric to the proof for the ADDR rule.
- *rule SUBR:* Similar to the ADDR case.
- *rule BINOP:* In this case $v = \mathcal{E}[\![e_1 \diamond e_2]\!](\sigma,h)$, if defined, is necessarily an element of $\mathbb{V}_n = (\!| \texttt{word}_n |\!)_{\mathcal{L},v}$.
- *rule SUBVAL:* By induction hypothesis and the monotonicity of $(\!| \cdot |\!)_{\mathcal{L},v}$ with regard to $\preceq_{\mathbb{T}}$ (Lemma 4.5).
- *rule REFINE:* By induction hypothesis and by definition of the interpretation of a type (Definition 4.8).

□

4.3.2 Typed semantics of statements

The typed semantics of instructions is defined in Figure 4.5. It is identical to the untyped semantics, except for assignments, memory writes, and dynamic allocation. An assignment not only updates the store with the new value, but updates the type environment with a valid type for the right-hand-side expression. Formally, since an expression will in general have several types, this makes the semantics non-deterministic.

Memory writes, on the other hand, are deterministic. The difference between memory writes in the untyped semantics and in this one is that this semantics excludes ill-typed memory writes, i.e. writes that would introduce an inconsistency between the heap labelling and the values.

$$
\begin{array}{rcl}
\mathcal{C}\llbracket\cdot\rrbracket_t & : & \mathit{stmt} \times \mathcal{P}(\mathbb{S}_t) \rightarrow \mathcal{P}(\mathbb{S}_t) \\
\mathcal{C}\llbracket e\rrbracket_t\,\mathrm{S} & = & \{(\sigma, \Gamma, h, \mathcal{L}) \in \mathrm{S} \mid \mathcal{E}\llbracket e\rrbracket(\sigma, h) = (\ell, v) \wedge v \neq 0\} \\
\\
\llbracket\cdot\rrbracket_t & : & \mathit{stmt} \times \mathcal{P}(\mathbb{S}_t) \rightarrow \mathcal{P}(\mathbb{S}_t) \\
\llbracket\mathbf{skip}\rrbracket_t\,\mathrm{S} & = & \mathrm{S} \\
\llbracket\mathrm{P}_1;\mathrm{P}_2\rrbracket_t\,\mathrm{S} & = & \left(\llbracket\mathrm{P}_2\rrbracket_t \circ \llbracket\mathrm{P}_1\rrbracket_t\right)\mathrm{S} \\
\\
\llbracket x := e\rrbracket_t\,\mathrm{S} & = & \left\{(\sigma[x \leftarrow v],\ \Gamma[x \leftarrow t],\ h,\ \mathcal{L}) \left| \begin{array}{l} s \overset{\text{def}}{=} (\sigma, \Gamma, h, \mathcal{L}) \in \mathrm{S}, \\ v = \mathcal{E}\llbracket e\rrbracket(\sigma, h), \\ s \vdash_{\mathcal{V}} e : t \end{array} \right.\right\} \\
\\
\llbracket *_\ell e_1 := e_2\rrbracket_t\,\mathrm{S} & = & \left\{(\sigma, \Gamma, h[a..a + \ell \leftarrow v], \mathcal{L}) \left| \begin{array}{l} s \overset{\text{def}}{=} (\sigma, \Gamma, h, \mathcal{L}) \in \mathrm{S}, \\ s \vdash_{\mathcal{V}} e_1 : u.(i)*,\ u.(i) \preceq t.(0), \\ s \vdash_{\mathcal{V}} e_2 : t,\ \mathrm{size}(t) = \ell, \\ \mathcal{E}\llbracket e_1\rrbracket(\sigma, h) = (\mathcal{W}, a), \\ a \neq 0, \\ \mathcal{E}\llbracket e_2\rrbracket(\sigma, h) = (\ell, v) \end{array} \right.\right\} \\
\llbracket x := \mathbf{malloc}(e)\rrbracket_t\,\mathrm{S} & = & \left\{\left(\begin{array}{l} \sigma[x \leftarrow a],\ \Gamma[x \leftarrow \mathtt{word}_\ell.(0)*], \\ h[a..a + \ell \leftarrow v], \\ \mathcal{L}[a..a + \ell \leftarrow \mathtt{word}_\ell] \end{array}\right) \left| \begin{array}{l} (\sigma, \Gamma, h, \mathcal{L}) \in \mathrm{S}, \\ \mathcal{E}\llbracket e\rrbracket(\sigma, h) = (\mathcal{W}, \ell), \\ \ell > 0,\ v \in \mathbb{V}_\ell, \\ [a, a + \ell[\cap \mathrm{dom}(h) = \emptyset \end{array} \right.\right\} \\
& & \cup \left\{\left(\begin{array}{l} \sigma[x \leftarrow (\mathcal{W}, 0)], \\ \Gamma[x \leftarrow \mathtt{word}_\ell.(0)*], \\ h,\ \mathcal{L} \end{array}\right) \left| \begin{array}{l} (\sigma, \Gamma, h, \mathcal{L}) \in \mathrm{S}, \\ \mathcal{E}\llbracket e\rrbracket(\sigma, h) = (\mathcal{W}, \ell), \\ \ell > 0 \end{array} \right.\right\} \\
\\
\llbracket\mathbf{if}\ e\ \mathbf{then}\ \mathrm{P}_1\ \mathbf{else}\ \mathrm{P}_2\ \mathbf{end}\rrbracket_t\,\mathrm{S} & = & \llbracket\mathrm{P}_1\rrbracket_t\left(\mathcal{C}\llbracket e\rrbracket_t\,\mathrm{S}\right) \cup \llbracket\mathrm{P}_2\rrbracket_t\left(\mathcal{C}\llbracket\neg e\rrbracket_t\,\mathrm{S}\right) \\
\llbracket\mathbf{while}\ e\ \mathbf{do}\ \mathrm{P}\ \mathbf{done}\rrbracket_t\,\mathrm{S} & = & \mathcal{C}\llbracket\neg e\rrbracket_t\,(\mathrm{lfp}\,\mathrm{F}) \\
& & \text{where } \mathrm{F}(\mathrm{X}) \overset{\text{def}}{=} \mathrm{S} \cup \llbracket\mathrm{P}\rrbracket_t\left(\mathcal{C}\llbracket e\rrbracket_t\,\mathrm{X}\right)
\end{array}
$$

Figure 4.5: Typed semantics of While-memory statements.

Finally, dynamic allocation behaves identically to its untyped counterpart, except that it attributes the type $\texttt{word}_\ell$ to the allocated region, where ℓ is the size of that region.

The typed semantics preserves the well-typedness of states:

Theorem 4.8 (Preservation of typing of states). *Let $\upsilon : \mathbb{V}^\sharp \to \mathbb{V}$ be a valuation and $\mathrm{S} \subseteq \mathbb{S}_t$ be a set of well-typed states under υ. For all programs P, for all $s_1 \in [\![\mathrm{P}]\!]_t\, \mathrm{S}$, s_1 is well typed under υ.*

Proof. By induction on the structure of P. We skip the easy cases of **skip** and sequence.

- *assignment* ($x := e$): The only change between s_1 and the states of S is the value and type of x, so we only have to verify that they are consistent. We have $\sigma(x) = \mathcal{E}[\![e]\!](\sigma, h)$, thus by Theorem 4.7 $\sigma(x) \in (\!| t |\!)_{\mathcal{L},\upsilon}$.
- *memory write* ($*_\ell e_1 := e_2$): Let $h' = h[a..a + \ell \leftarrow v]$. We need to verify the consistency between the modified heap region and the (unchanged) labelling, i.e. we need to verify:

$$\forall a_0 \in \mathbb{A}, \forall w \in \mathbb{T},\ \mathcal{L}(a_0) = w.(0) \implies h'[a_0..a_0 + \mathrm{size}(w)] \in (\!| w |\!)_{\mathcal{L},\upsilon}.$$

  Let a_0 and w be such that $\mathcal{L}(a_0) = w.(0)$. Two cases are possible: either the heap region $[a_0, a_0 + \mathrm{size}(w)[$ intersects the modified region, or not.

  - If $[a_0, a_0 + \mathrm{size}(w)[\cap [a, a + \ell[= \emptyset$, then $h'[a_0..a_0 + \mathrm{size}(w)] = h[a_0..a_0 + \mathrm{size}(w)] \in (\!| w |\!)_{\mathcal{L},\upsilon}$.
  - Otherwise, then $a = a_0 + j$ for some j, and $\mathcal{L}(a_0 + j) = w.(j) = \mathcal{L}(a)$. Since $s \vdash_\upsilon e_1 : u.(i)*$ and $a \neq 0$, Theorem 4.7 yields $\mathcal{L}(a) \preceq u.(i)$. Therefore $w.(j) \preceq u.(i) \preceq t.(0)$. By well-typing of s, we have $h[a_0..a_0 + \mathrm{size}(w)] = v_1 :: v_2 :: v_3 \in (\!| w |\!)_{\mathcal{L},\upsilon}$ (where $v_1 \in \mathbb{V}_i$, $v_2 \in \mathbb{V}_{\mathrm{size}(t)}$, $v_3 \in \mathbb{V}_{\mathrm{size}(w)-i-\mathrm{size}(t)}$). Since $s \vdash_\upsilon e_2 : t$, $v \in (\!| t |\!)_{\mathcal{L},\upsilon}$, Lemma 4.6 yields:

$$h'[a_0..a_0 + \mathrm{size}(w)] = v_1 :: v :: v_3 \in (\!| w |\!)_{\mathcal{L},\upsilon}$$

- *memory allocation* ($x :=$ **malloc**(e)): We let $(\sigma_1, \Gamma_1, h_1, \mathcal{L}_1) = s_1$. If $\sigma_1(x) = (\mathcal{W}, 0)$, then trivially $\sigma_1(x) \in (\!| \Gamma_1(x) |\!)_{\mathcal{L}_1,\upsilon}$ and thus s_1 is well typed. Otherwise, s_1 is also well typed because $\sigma_1(x)$ is indeed the address of a region of type $\texttt{word}_\ell$, and in addition the contents of that region is in $(\!| \texttt{word}_\ell |\!)_{\mathcal{L}_1,\upsilon} = \mathbb{V}_\ell$.
- *conditional* (**if** e **then** P_1 **else** P_2 **end**): The condition operators $\mathcal{C}[\![e]\!]_t$ and $\mathcal{C}[\![\neg e]\!]_t$ only remove states, and by induction hypothesis $[\![\mathrm{P}_1]\!]_t$ and $[\![\mathrm{P}_2]\!]_t$ preserve typing, therefore all elements of the union are well typed under υ.
- *loop* (**while** e **do** P **done**): F is a monotone function that preserves typing (by induction hypothesis), therefore all elements of lfp F are well typed.

□

4.4 Extending the type system: directions and pitfalls

It is worth pointing out that the grammar of physical types could be extended, and the criteria of subtyping between address types modified, in ways that retain the preservation of typing by the typed semantics (which constitutes the base of our static analysis).

The main property that extensions should preserve is the fact that types either cover disjoint regions, or are in a containment relation (Theorem 4.4).

4.4.1 Invalid address subtyping rules

Intuitively, one may want to introduce the rule that the address of a refinement type is also the address of the underlying type without the predicate, *via* subtyping. In other words, one may want to introduce the following weakening:

$$\forall k \in [0, \mathrm{size}(t)[,\ w(\{x : t \mid p(x)\}.(k)) = t.(k)$$

such that

$$\forall k \in [0, \mathrm{size}(t)[,\ \{x : t \mid p(x)\}.(k) \preceq t.(k).$$

However, this leads to a non-type-preserving semantics:

Example 4.6. Assume the above rule, and consider the semantics of the following program, applied to an initial state s in which variable y has the type $\{x : \texttt{word}_4 \mid x \leq 5\}.(0)*$:

$$\begin{aligned}&\textit{// Assumption: } y : \{x : \texttt{word}_4 \mid x \leq 5\}.(0)* \\ &*_4 y := 42\end{aligned}$$

When computing its semantics $[\![*_4 y := 42]\!]_t$ applied to $\{s\}$, the memory write is considered valid because 42 has type $\texttt{word}_4$, and $\{x : \texttt{word}_4 \mid x \leq 5\}.(0) \preceq \texttt{word}_4.(0)$. Yet, the resulting memory states are ill-typed: the region pointed to by y contains the value 42, which is not in the interpretation of its type.

The key idea is that address subtyping describes *containment relations between types in memory*, and is not related to the possible values contained by these types. This is the reason why value-centered subtyping $\preceq_{\mathbb{T}}$ is distinct from address subtyping. More generally, it would break the semantics to introduce the rule that:

$$t \preceq_{\mathbb{T}} u \implies t.(k) \preceq u.(k)$$

as the above example testifies. Similarly, for pointer types, one may try to introduce the rule that, if two address types are in a subtyping relation, then so are the pointers to these addresses, i.e. to add the following weakening rule:

$$\forall k \in [0, \mathcal{W}[,\ w((t.(0)*).(k)) = (u.(0)*).(k) \quad \text{if } t.(0) \preceq u.(0)$$

such that:

$$\forall k \in [0, \mathcal{W}[,\ (t.(0)*).(k) \preceq (u.(0)*).(k) \quad \text{if } t.(0) \preceq u.(0)$$

However, this also leads to a non-type-preserving semantics, as exemplified by the following program:

Example 4.7. Consider again type `node`:

$$\begin{array}{rrcl} \mathcal{M}: & \texttt{node\_kind} & \mapsto & \{x : \texttt{word}_4 \mid x \leq 5\} \\ & \texttt{uf} & \mapsto & \texttt{uf}.(0)* \\ & \texttt{dll} & \mapsto & \{x : \texttt{dll}.(0)* \mid x \neq 0\} \times \{x : \texttt{dll}.(0)* \mid x \neq 0\} \\ & \texttt{node} & \mapsto & \texttt{node\_kind} \times \texttt{dll} \times \texttt{uf} \end{array}$$

And the program:

$$\begin{aligned}&\textit{// Assumption: } x : (\texttt{node}.(0)*).(0)*,\ y : \texttt{int}.(0)* \\ &*_4 x := y; \\ &*_4(*_4 x + 4) := 0\end{aligned}$$

Type `node` contains the type `node_kind` of size 4 bytes as its first component. Therefore, assuming the incorrect rule above, $(\texttt{node}.(0)*).(0) \preceq (\texttt{node\_kind}.(0)*).(0)$. As a consequence, the first statement is considered well-typed; x now points to a region of type $\texttt{int}.(0)*$, and not $\texttt{node}.(0)*$ as its type should guarantee. The execution of the second instruction succeeds because $(*_4 x + 4)$ has type $\texttt{node}.(4)*$; however, this statement writes past the region initially pointed to by y (of size 4), into a region of arbitrary type. This corresponds to an out-of-bounds memory write.

4.4.2 Possible extensions

Relations between structure fields

So far, we have only considered predicates on individual fields of composite types. One may want to consider predicates relating two or more fields, such as:

```
struct foo { int x; int y; int z; }; /* x < y */
```

Physical types already allow to express this kind of predicates, but in a rather cumbersome way since it requires to use bitwise operators. For example, the physical type corresponding to the above C structure would be:

$$\mathcal{M}(\texttt{foo}) = \{v : \texttt{word}_4 \times \texttt{word}_4 \times \texttt{word}_4 \mid (v \,\&\, \texttt{0xffffffff}) < ((v >> 32) \,\&\, \texttt{0xffffffff})\}$$

where >> is the usual left shift operator as used in C. Then, assuming a numerical abstract domain able to express this kind of bitwise predicates, such typing predicates can be verified (see analysis details in next chapter). However, there is clearly room for improvement in the syntax of such predicates, possibly by introducing record types and letting predicates relate record fields. More generally, introducing named fields, rather than relying on simple binary products, may improve the usability of the tools constructed from this type system.

Another limitation is that, due to the definition of address subtyping, it is currently impossible to write a well-typed program that modifies only the `z` field of structure `foo`. This is because the following address subtyping does not hold:

$$\texttt{foo}.(8) \preceq \texttt{word}_4.(0)$$

However, it would be a reasonable subtyping to have, since it preserves Lemma 4.6 and thus does not break the type-preserving properties of the semantics. Maybe address subtyping could be adapted so that local modifications of non-constrained fields can be allowed, without changing other components of the type system.

Dependent types

The current type system allows to relate different fields of a composite type, but still has limits. Consider, for instance, a program manipulating strings, represented as an array of bytes, and the length of that array, e.g.:

```
struct string {
  unsigned int length;
  char * ptr_to_start;
};

struct string s;
s.length = 24;
s.ptr_to_start = malloc(24 * sizeof(char));
```

Current types allow to relate the length of the array and the `length` field, but only *via* global symbolic value. For instance, the `string` type above could be represented by the physical type:

$$\mathcal{M}(\texttt{string}) = \{x : \texttt{word}_4 \mid x = \mathbb{v}_\ell\} \times \texttt{word}_1[\mathbb{v}_\ell].(0)*$$

where $\mathbb{v}_\ell$ is a symbolic variable, which is used here as the array length. Note that this uses a symbolic variable as the length of an array, which is not possible using physical types defined up to now, but is possible in *abstract physical types* defined in Chapter 5, and hence is possible in our analysis.

The limitation with this approach is that all instances of `string` refer to the same $\mathbb{v}_\ell$, and therefore all buffers must have the same length. A natural extension would be to introduce types *quantified by a value*, either universally or existentially. The grammar of types may be extended as follows, drawing inspiration from dependent types:

$$\begin{array}{rcll}
\mathbb{T} \ni t & ::= & \cdots & \\
& \mid & \Pi\mathbb{v}.\, t & \textit{(type constructor quantified by a value)} \\
& \mid & t(\mathbb{v}) & \textit{(type constructor application)} \\
& \mid & \Sigma\mathbb{v}.\, t & \textit{(existentially quantified type)}
\end{array}$$

For instance, the `string` type above would be represented by a type universally quantified by a value which is both the value of `length` and the length of the pointed array:

$$\mathcal{M}(\texttt{string}) = \Pi\mathbb{v}.\, \{x : \texttt{word}_4 \mid x = \mathbb{v}\} \times \texttt{word}_1[\mathbb{v}].(0)*$$

Then, the C type:

```
struct msg {
  int flags;
  struct string str;
};
```

Could be represented by:

$$\mathcal{M}(\texttt{msg}) = \texttt{word}_4 \times (\Sigma\mathbb{v}.\ \texttt{string}(\mathbb{v}))$$

i.e. the product of an integer and a `string` whose parameter is an existentially quantified value.

The extension of our definitions to accommodate the addition of full-fledged dependent types remains to be done. However, two reasons suggest that this direction may be relevant: our abstract interpretation framework seems particularly suited to introducing symbolic variables on the fly; and in our system, type inference is already undecidable (but useful in practice, as the next chapters will show), therefore introducing value-quantified types will not necessarily make it intractable.

Chapter 5

Type-based shape abstract domain

Outline of the current chapter

In this chapter, we describe our abstract domain based on physical types. First, we present the abstraction informally on the union-find example introduced in Chapter 4. We then show that some common properties cannot be represented in physical types, leading us to define an abstraction of these types (Section 5.2) in which pointer offsets are managed by an independent numerical abstract domain. Then, Section 5.3 covers the main abstraction, and Section 5.4 describes the abstract semantics of the analysis and the associated soundness theorems. Finally, in Section 5.4.5 we describe how to use physical types to approximate aliasing between regions, which we will use in the refined abstraction of Chapter 6.

```
typedef struct uf {
  struct uf* parent;
} uf;

typedef struct dll {
  struct dll *prev; /* != null. */
  struct dll *next; /* != null. */
} dll;

typedef unsigned int node_kind;
typedef struct node {
  node_kind kind; /* kind <= 5. */
  dll dll;
  uf uf;
} node;

uf *uf_find(uf *x) {
  while(x->parent != 0) {
    uf *parent = x->parent;
    if(parent->parent == 0)
      return parent;
    x->parent = parent->parent;
    x = parent->parent;
  }
  return x;
}
```

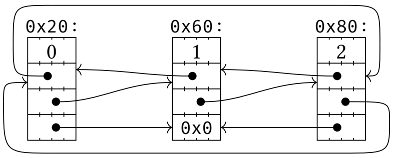
```
void dll_union(dll *x, dll *y) {
  y->prev->next = x->next;
  x->next->prev = y->prev;
  x->next = y; y->prev = x;
}

void uf_union(uf *x, uf *y) {
  uf *rootx = uf_find(x);
  uf *rooty = uf_find(y);
  if(rootx != rooty)
    rootx->parent = rooty;
}

void merge(node *x, node *y) {
  dll_union(&x->dll, &y->dll);
  uf_union(&x->uf, &y->uf);
}

node *make(node_kind kind) {
  node *n = malloc(sizeof(node));
  n->kind = kind;
  n->dll.next = &n->dll;
  n->dll.prev = &n->dll;
  n->uf.parent = NULL;
  return n;
}
```

Figure 4.1: An algorithm for union-find and listing elements in a partition. (repeated from page 34)



(a) Concrete state.

$$
\begin{aligned}
\mathcal{M}\ :\ \texttt{node\_kind} &\mapsto \{x : \texttt{word}_4 \mid x \leq 5\}\\
\texttt{uf} &\mapsto \texttt{uf}.(0)*\\
\texttt{dll} &\mapsto \{x : \texttt{dll}.(0)* \mid x \neq 0\} \times \{x : \texttt{dll}.(0)* \mid x \neq 0\}\\
\texttt{node} &\mapsto \texttt{node\_kind} \times \texttt{dll} \times \texttt{uf}
\end{aligned}
$$

(b) Physical types.

$$
\begin{array}{rcll}
\multicolumn{4}{l}{\forall \mathcal{v}, \forall(\sigma, h, \mathcal{L}, \Gamma) \text{ well-typed state}, \forall v \text{ value } :}\\
v \in \llparenthesis \texttt{node}.(0)* \rrparenthesis_{\mathcal{L},\mathcal{v}} \wedge v \neq 0 & \Longrightarrow & v+4 \in \llparenthesis \texttt{node}.(4) \rrparenthesis_{\mathcal{L},\mathcal{v}} \wedge v+4 \neq 0 & (1)\\
\llparenthesis \texttt{node}.(4)* \rrparenthesis_{\mathcal{L},\mathcal{v}} & \subseteq & \llparenthesis \texttt{dll}.(0)* \rrparenthesis_{\mathcal{L},\mathcal{v}} & (2)\\
v \in \llparenthesis \texttt{dll}.(0)* \rrparenthesis_{\mathcal{L},\mathcal{v}} \wedge v \neq 0 & \Longrightarrow & h[v..v+4] \in \llparenthesis \{x : \texttt{dll}.(0)* \mid x \neq 0\} \rrparenthesis_{\mathcal{L},\mathcal{v}} & (3)\\
\llparenthesis \texttt{uf}.(0)* \rrparenthesis_{\mathcal{L},\mathcal{v}} & \cap & \llparenthesis \texttt{dll}.(0)* \rrparenthesis_{\mathcal{L},\mathcal{v}} = \{0\} & (4)
\end{array}
$$

(c) Some structural invariants entailed by $\mathcal{M}$

Figure 4.2: Concrete state, physical types and example structural invariants. (repeated from page 35)

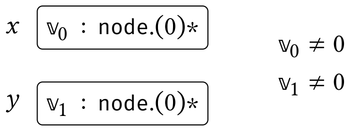


Figure 5.1: Abstract state at beginning of the `merge` function.

5.1 Informal overview of the abstraction

Consider again the example code of Figure 4.1, repeated on page 50. The abstract state shown in Figure 5.1 represents the initial state when execution of the `merge` function begins (this function requires that it is given non-null pointers to `node` as arguments). Each variable is associated to both an abstract type describing possible values stored in the variable, and to a symbolic variable used to attach numerical constraints to this value. For instance, variable `x` is bound to physical type `node`.(0)*, meaning that its value belongs to the set of possible values of that type, $(\!|\,\texttt{node}.(0)*\,|\!)_{\mathcal{L},\mathrm{V}}$; furthermore it is bound to symbolic variable $\mathbb{v}_0$ which is constrained to be non-null. Combined with structural invariants of Equations (1), (2) and (3), we can verify that `x+4` (the low-level counterpart of `&x->dll`) points to a valid address, that can be safely casted as type `dll`.(0)*, and that reading 4 bytes from this address will return a value that has type $\{x : \texttt{dll}.(0)* \mid x \neq 0\}$. Eventually, we can verify that each statement preserves these structural invariants, which entails that all memory accesses performed by the call to `dll_union` are valid.

5.2 Abstract physical types

5.2.1 Motivation

Address types can represent pointers into arrays very precisely, e.g., if `int` is a type of size 4 bytes, then `int`[10].(8) represents the address of the third element in an array of 10 `int`s. However, there is no type representing "an element of unknown index in a `int`[10] array". In array analysis, in order to compute invariants, analysis tools must be able to represent elements that are at a valid, but unknown, index. Less commonly, it can sometimes be useful to represent pointers that may point to several fields in a C structure.

We therefore define a domain of abstract types, $\mathbb{T}^\sharp$, which replaces exact pointer offsets by symbolic values.

Example 5.1. The following abstract type represents the address of a valid element in an array of ten `int`s:

$$\texttt{int}[10].(\mathbb{i}) \text{ where } \mathbb{i} \in [0, 39] \text{ and } \mathbb{i} \equiv 0 \pmod 4$$

The form of logical constraints expressible on symbolic values depends on the numeric domain $\mathbb{D}_{\mathrm{num}}$ used in the analysis, of concretization $\gamma_{\mathrm{D}} : \mathbb{D}_{\mathrm{num}} \to \mathcal{P}(\mathbb{V}^\sharp \to \mathbb{V})$. The numeric constraints in the example above can be expressed by the numeric domain of intervals with congruence information.

The second motivation for abstract types is to add the possibility to represent arrays of symbolic size. This can be useful to analyze a number of programs, notably programs that allocate arrays of dynamic size. The size of such arrays can be represented by a symbolic variable. Symbolic variables also allow to represent statically allocated arrays whose size is a parameter of the analysis; this is critical to analyze embedded OS kernels, as they often manipulate a statically allocated arrays of tasks.

$$
\begin{array}{rcll}
\mathbb{T}^\sharp \ni \mathbb{t} & ::= & t & \textit{(concrete type } t \in \mathbb{T}\textit{)} \\
 & | & \mathbb{t}_a{*} & \textit{(pointer type)} \\
 & & & \\
\mathbb{t}_a & ::= & t.(\mathbb{i}) & \textit{(address type with symbolic offset } \mathbb{i} \in \mathbb{V}^\sharp, t \in \mathbb{T}\textit{)} \\
 & | & t[\mathbb{s}].(\mathbb{i}) & \textit{(address in array, } \mathbb{s}, \mathbb{i} \in \mathbb{V}^\sharp, t \in \mathbb{T}\textit{)}
\end{array}
$$

Figure 5.2: Grammar of abstract physical types.

5.2.2 Definition

Definition 5.1. The grammar of abstract types is defined in Figure 5.2. It is similar to the grammar of concrete types (Figure 4.3) with two differences:

1. Pointer types may have a *symbolic* offset rather than a simple integer.

2. A special kind of pointer types, *pointer to arrays of symbolic length*, is added. Their offsets are also symbolic.

The concretization γ_{T} of an abstract type must account for the symbolic variables and the concrete values they represent, so it also yields a valuation: $\gamma_{\mathrm{T}} : \mathbb{T}^\sharp \to \mathcal{P}\big(\mathbb{T} \times (\mathbb{V}^\sharp \to \mathbb{V})\big)$ is defined as:

$$
\begin{aligned}
\gamma_{\mathrm{T}}(t) &= \{(t, v) \mid v \in \mathbb{V}^\sharp \to \mathbb{V}\} \text{ if } t \in \mathbb{T} \\
\gamma_{\mathrm{T}}(t.(\mathbb{i}){*}) &= \{(t.(v(\mathbb{i})){*},\ v) \mid 0 \le v(\mathbb{i}) < \mathrm{size}(t)\} \\
\gamma_{\mathrm{T}}(t[\mathbb{s}].(\mathbb{i}){*}) &= \{(t[v(\mathbb{s})].(v(\mathbb{i})){*},\ v) \mid 0 \le v(\mathbb{i}) < v(\mathbb{s}) \cdot \mathrm{size}(t)\}
\end{aligned}
$$

Note that abstract types are not nested: abstract pointers point to concrete types, and similarly arrays of symbolic sizes contain concrete types.

5.2.3 Abstract subtyping

We now proceed to define an *abstract subtyping* relation that approximates the value-centered subtyping relation $\preceq_{\mathbb{T}}$ of concrete types. By "approximating $\preceq_{\mathbb{T}}$", we mean that if $\mathbb{t}_1$ is a subtype of $\mathbb{t}_2$, then for all types t_1, t_2 in their respective concretizations, $t_1 \preceq_{\mathbb{T}} t_2$ should hold.

Intuitively, abstract subtyping works as follows:

- If $\mathbb{t}_1$ and $\mathbb{t}_2$ are two concrete types, they can be compared directly using $\preceq_{\mathbb{T}}$.
- Otherwise, if they are pointers to the same type, abstract subtyping holds if the values represented by their symbolic offsets are equal.
- If none of the above applies, then we attempt to *approximate* the abstract types into concrete types in order to compare them using $\preceq_{\mathbb{T}}$.

We assume that the numerical domain provides the possibility to test equality on two symbolic variables in two different numerical abstract states:

Definition 5.2 (Equality test on symbolic variables). We assume that the numerical domain $\mathbb{D}_{\mathrm{num}}$ provides an operation $\mathbb{i}\ {}^{v_1^\sharp}{=}^{v_2^\sharp}\ \mathbb{j}$ for any $v_1^\sharp, v_2^\sharp \in \mathbb{D}_{\mathrm{num}}$ and $\mathbb{i}, \mathbb{j} \in \mathbb{V}^\sharp$, such that:

$$
\mathbb{i}\ {}^{v_1^\sharp}{=}^{v_2^\sharp}\ \mathbb{j} \implies \forall v_1 \in \gamma_{\mathrm{num}}(v_1^\sharp),\ \forall v_2 \in \gamma_{\mathrm{num}}(v_2^\sharp),\ v_1(\mathbb{i}) = v_2(\mathbb{j})
$$

To simplify the expression of the soundness theorem, let us define a "combined" type-numeric concretization function:

Definition 5.3. The function $\gamma_{\mathrm{T,num}} : \mathbb{T}^\# \times \mathbb{D}_{\mathrm{num}} \to \mathcal{P}\big(\mathbb{T} \times (\mathbb{V}^\# \to \mathbb{V})\big)$ is defined as:

$$\gamma_{\mathrm{T,num}}(\mathbb{t}, \nu^\#) = \{(t, \nu) \in \gamma_{\mathrm{T}}(\mathbb{t}) \mid \nu \in \gamma_{\mathrm{num}}(\nu^\#)\}$$

We now define an operator to obtain a concrete type from an abstract one.

Definition 5.4. Given $\nu^\# \in \mathbb{D}_{\mathrm{num}}$, the function $\mathrm{conc}_{\nu^\#} : \mathbb{T}^\# \to \mathbb{T}$ is defined as follows:

$$\begin{aligned}
\mathrm{conc}_{\nu^\#}(t) &= t && \text{if } t \in \mathbb{T}\\
\mathrm{conc}_{\nu^\#}(t.(\mathbb{i})*) &= t.(i)* && \text{if } \nu^\# \vDash \mathbb{i} = i \wedge 0 \le i < \mathrm{size}(t), \text{ with } i \in \mathbb{N}\\
\mathrm{conc}_{\nu^\#}(t[\mathbb{s}].(\mathbb{i})*) &= t.(k)* && \text{if } \nu^\# \vDash 0 \le \mathbb{i} < \mathbb{s} \cdot \mathrm{size}(t) \ \wedge\ \mathbb{i} \equiv k \pmod{\mathrm{size}(t)}, \text{ with } k \in \mathbb{N}\\
\mathrm{conc}_{\nu^\#}(t[\mathbb{s}].(\mathbb{i})*) &= \mathtt{word}_{\mathcal{W}} && \text{otherwise}\\
\mathrm{conc}_{\nu^\#}((t_1 \times t_2).(\mathbb{i})*) &= t_1.(i)* && \text{if } t_1 = t_2 \text{ and } \nu^\# \vDash \mathbb{i} \in \{i,\ \mathrm{size}(t_1) + i\} \wedge 0 \le i < \mathrm{size}(t_1), \text{ with } i \in \mathbb{N}\\
\mathrm{conc}_{\nu^\#}(t.(\mathbb{i})*) &= \mathtt{word}_{\mathcal{W}} && \text{otherwise}
\end{aligned}$$

The $\mathrm{conc}_{\nu^\#}$ preserves value-centered subtyping in the following sense:

Lemma 5.1. *Given $\mathbb{t} \in \mathbb{T}^\#$ and $\nu^\# \in \mathbb{D}_{\mathrm{num}}$, for all $(t, \nu) \in \gamma_{\mathrm{T,num}}(\mathbb{t}, \nu^\#)$, $t \preceq_{\mathbb{T}} \mathrm{conc}_{\nu^\#}(\mathbb{t})$.*

Proof. Straightforward from the definition of the weakening of an address type (Definition 4.4), and the definition of $\preceq_{\mathbb{T}}$ (Definition 4.9). □

However, $\mathrm{conc}_{\nu^\#}$ loses precision on pointers to arrays, in the sense that the resulting pointer type is not a pointer to an array, but a weakening thereof. This is due to the fact that there is no way to represent pointers to *more than one* element of an array, as we highlighted in Section 5.2.1 when motivating abstract types.

We can now define an abstract subtyping on pairs of abstract types and numerical states:

Definition 5.5 (Abstract subtyping relation on $\mathbb{T}^\# \times \mathbb{D}_{\mathrm{num}}$). Let $\mathbb{t}_1, \mathbb{t}_2 \in \mathbb{T}^\#$ and $\nu_1^\#, \nu_2^\# \in \mathbb{D}_{\mathrm{num}}$. The relation $\sqsubseteq_{\mathrm{T}} \subseteq (\mathbb{T}^\# \times \mathbb{D}_{\mathrm{num}}) \times (\mathbb{T}^\# \times \mathbb{D}_{\mathrm{num}})$ is defined as follows:

- If $\mathbb{t}_1 = t.(\mathbb{i})*$ and $\mathbb{t}_2 = t.(\mathbb{j})*$ for some $t \in \mathbb{T}$, then
$$(\mathbb{t}_1, \nu_1^\#) \sqsubseteq_{\mathrm{T}} (\mathbb{t}_2, \nu_2^\#) \iff \mathbb{i} \overset{\nu_1^\#}{=}{}^{\nu_2^\#}\, \mathbb{j}$$
- If $\mathbb{t}_1 = t[\mathbb{s}].(\mathbb{i})*$ and $\mathbb{t}_2 = t[\mathbb{s}'].(\mathbb{j})*$, for some $t \in \mathbb{T}$, then
$$(\mathbb{t}_1, \nu_1^\#) \sqsubseteq_{\mathrm{T}} (\mathbb{t}_2, \nu_2^\#) \iff \mathbb{s} \overset{\nu_1^\#}{=}{}^{\nu_2^\#}\, \mathbb{s}' \ \wedge\ \mathbb{i} \overset{\nu_1^\#}{=}{}^{\nu_2^\#}\, \mathbb{j}$$
- Otherwise:
$$(\mathbb{t}_1, \nu_1^\#) \sqsubseteq_{\mathrm{T}} (\mathbb{t}_2, \nu_2^\#) \iff \mathbb{t}_2 \in \mathbb{T} \ \wedge\ \mathrm{conc}_{\nu_1^\#}(\mathbb{t}_1) \preceq_{\mathbb{T}} \mathbb{t}_2$$

The idea of $(\mathbb{t}_1, \nu_1^\#) \sqsubseteq_{\mathrm{T}} (\mathbb{t}_2, \nu_2^\#)$ is that a subtyping relation exists on the concrete types for all valuations matching the constraints of $\nu_1^\#$ and $\nu_2^\#$. In other words, abstract subtyping is a sound abstraction of concrete subtyping:

Theorem 5.2 (Soundness of $\sqsubseteq_{\mathrm{T}}$). *Given $\mathbb{t}_1, \mathbb{t}_2 \in \mathbb{T}^\#$ and $\nu_1^\#, \nu_2^\# \in \mathbb{D}_{\mathrm{num}}$, for all $(t_1, \nu_1) \in \gamma_{\mathrm{T,num}}(\mathbb{t}_1, \nu_1^\#)$ and $(t_2, \nu_2) \in \gamma_{\mathrm{T}}(\mathbb{t}_2, \nu_2^\#)$:*
$$(\mathbb{t}_1, \nu_1^\#) \sqsubseteq_{\mathrm{T}} (\mathbb{t}_2, \nu_2^\#) \implies t_1 \preceq_{\mathbb{T}} t_2$$

$$\mathtt{node}[10].(\mathbb{i})* \qquad \sqsubseteq_{\mathrm{T}} \quad \mathtt{dll}.(2)*$$

$$\mathbb{i} \in [48, 160[$$
$$\mathbb{i} \equiv 6 \pmod{16}$$

Figure 5.3: Example of abstract type inclusion.

Proof. If $\mathbb{t}_1 = t.(\mathbb{i})*$ and $\mathbb{t}_2 = t.(\mathbb{j})*$ for some $t \in \mathbb{T}$, then $t_1 = t.(\mathcal{v}(\mathbb{i}))*$ and $t_2 = t.(\mathcal{v}(\mathbb{j}))* = t.(\mathcal{v}(\mathbb{i}))*$ by the definitions of γ_{T} and of the equality test operator (Definitions 5.1 and 5.2). In other words, $t_1 = t_2$ and therefore $t_1 \preceq_{\mathbb{T}} t_2$. Similar proof if $\mathbb{t}_1 = t[\mathbb{s}].(\mathbb{i})*$ and $\mathbb{t}_2 = t[\mathbb{s}'].(\mathbb{j})*$ for some $t \in \mathbb{T}$. Otherwise, Lemma 5.1 applies and yields the result. □

Example 5.2. Consider the types $\mathbb{t}_1 = \mathtt{node}[10].(\mathbb{i})*$ and $\mathbb{t}_2 = \mathtt{dll}.(2)*$, represented in Figure 5.3. We test the abstract inclusion of $(\mathbb{t}_1, \mathcal{v}_1^\#)$ in $(\mathbb{t}_2, \mathcal{v}_2^\#)$, where $\mathcal{v}_1^\#$ is such that $\mathcal{v}_1^\# \vDash \mathbb{i} \in [48, 160[\wedge \mathbb{i} \equiv 6 \pmod{16}$. Here the inclusion holds because (third case of Definition 5.5) $\mathbb{t}_2$ is a concrete type and $\mathrm{conc}_{\mathcal{v}_1^\#}(\mathbb{t}_1) = \mathtt{node}.(6)*$, and $\mathtt{node}.(6)* \preceq_{\mathbb{T}} \mathtt{dll}.(2)*$.

5.2.4 Abstract join

The join operator on abstract types is constructed with the same ideas as abstract subtyping: we first define a join operator on concrete types. Then, joining pointers to the same type yields the most precise result if those offsets can be verified to be equal. Other cases are handled by using $\mathrm{conc}_{\mathcal{v}^\#}$ to get back to the concrete type case.

Definition 5.6 (Concrete type join). The join operator on concrete types $\vee_{\mathbb{T}} : \mathbb{T} \times \mathbb{T} \rightharpoonup \mathbb{T}$ is defined as follows:

- If $t_1 = t.(i)*$ and $t_2 = u.(j)*$, then $t_1 \vee_{\mathbb{T}} t_2 = w.(k)*$, where $w.(k)$ is the least upper bound of $t.(i)$ and $u.(j)$ for $\preceq$, if it exists. The least upper bound can be computed as the first common element in the two chains of successive weakenings starting from $t.(i)$ and $u.(j)$. If there is no such upper bound, then $t_1 \vee_{\mathbb{T}} t_2 = \mathtt{word}_{\mathcal{W}}$.
- If $t_2 = \{x : t_1 \mid p(x)\}$ then $t_1 \vee_{\mathbb{T}} t_2 = t_1$. Symmetrically, if $t_1 = \{x : t_2 \mid p(x)\}$, then $t_1 \vee_{\mathbb{T}} t_2 = t_2$.
- Otherwise, if $\mathrm{size}(t_1) = \mathrm{size}(t_2)$, then $t_1 \vee_{\mathbb{T}} t_2 = \mathtt{word}_{\mathrm{size}(t_1)}$. The join of t_1 and t_2 is undefined if they are of different sizes.

Implementation-wise, a simple algorithm is to generate the two chains of weakenings of t_1 and t_2 and compare the chains starting from the the greatest elements (in the sense of $\preceq$). If there is no common element, then the join is $\mathtt{word}_{\mathcal{W}}$. Otherwise, we just iterate until the last common element is found: this element is the least upper bound. This algorithm is linear in the height of the abstract syntax trees (ASTs) (in the sense of the type grammar of Definition 4.1) of its operands; and it does not seem to cause performance problems in practice (see Section 7.3).

Definition 5.7 (Abstract join in $\mathbb{T}^\# \times \mathbb{D}_{\mathrm{num}}$). Let $\mathbb{t}_1, \mathbb{t}_2 \in \mathbb{T}^\#$ and $\mathcal{v}_1^\#, \mathcal{v}_2^\# \in \mathbb{D}_{\mathrm{num}}$. The operator $\sqcup_{\mathrm{T}} \subseteq (\mathbb{T}^\# \times \mathbb{D}_{\mathrm{num}}) \times (\mathbb{T}^\# \times \mathbb{D}_{\mathrm{num}}) \rightharpoonup \mathbb{T}^\#$ is defined as follows:

- If $\mathbb{t}_1 = t.(\mathbb{i})*$ and $\mathbb{t}_2 = t.(\mathbb{i}')*$ for some $t \in \mathbb{T}$, and there exists some $i \in \mathbb{N}$ such that $\mathcal{v}_1^\# \vDash \mathbb{i} = i$ and $\mathcal{v}_2^\# \vDash \mathbb{i}' = i$, then
$$(\mathbb{t}_1, \mathcal{v}_1^\#) \sqcup_{\mathrm{T}} (\mathbb{t}_2, \mathcal{v}_2^\#) = t.(i)*$$

$$
\begin{array}{llllll}
\texttt{node}[10].(\mathbb{i})* & \sqcup_{\mathrm{T}} & \texttt{node}[10].(\mathbb{i}')* & = & \texttt{node}[10].(54)* \\
\mathbb{i} = 54 & & \mathbb{i}' = 54 & & \\
\\
\texttt{node}[10].(\mathbb{i})* & \sqcup_{\mathrm{T}} & \texttt{node}[10].(\mathbb{i}')* & = & \texttt{word}_{\mathcal{W}} \\
\mathbb{i} = 54 & & \mathbb{i}' \in [0, 160[& & \\
 & & \mathbb{i}' \equiv 6 \pmod{16} & & \\
\\
\texttt{node}[10].(\mathbb{i})* & \sqcup_{\mathrm{T}} & (\texttt{dll} \times \texttt{word}_8).(2)* & = & \texttt{dll}.(2)* \\
\mathbb{i} \in [0, 160[& & & & \\
\mathbb{i} \equiv 6 \pmod{16} & & & &
\end{array}
$$

Figure 5.4: Examples of abstract type joins.

- If $\mathbb{t}_1 = t[\mathbb{s}].(\mathbb{i})$ and $\mathbb{t}_2 = t[\mathbb{s}].(\mathbb{j})$, for some $t \in \mathbb{T}$, and $\mathbb{s} \ {}^{v_1^\# = v_2^\#}\ \mathbb{s}'$ and there exists some $i \in \mathbb{N}$ such that $v_1^\# \vDash \mathbb{i} = i$ and $v_2^\# \vDash \mathbb{j} = i$,then

$$(\mathbb{t}_1, v_1^\#) \sqcup_{\mathrm{T}} (\mathbb{t}_2, v_2^\#) = t[\mathbb{s}].(i)*$$

- In all other cases, the join is equal to the concrete join of $\mathrm{conc}_{v_1^\#}(\mathbb{t}_1)$ and $\mathrm{conc}_{v_2^\#}(\mathbb{t}_2)$, if it exists:

$$(\mathbb{t}_1, v_1^\#) \sqcup_{\mathrm{T}} (\mathbb{t}_2, v_2^\#) = \mathrm{conc}_{v_1^\#}(\mathbb{t}_1) \vee_{\mathbb{T}} \mathrm{conc}_{v_2^\#}(\mathbb{t}_2)$$

If it does not exist, then $(\mathbb{t}_1, v_1^\#) \sqcup_{\mathrm{T}} (\mathbb{t}_2, v_2^\#)$ is undefined.

Note that cases where the result of $\sqcup_{\mathrm{T}}$ is undefined occur when joining types of different sizes. This will not happen in static analysis of C or machine code, as it could only happen if different execution paths store values of different lengths in the same lvalue (or in the same register or memory region, in the case of machine code), which is not permitted by the typing of C, nor by the semantics of machine code.

Note that $\sqcup_{\mathrm{T}}$ computes upper bounds in the sense of value subtyping $\preceq_{\mathbb{T}}$, not in the sense of concrete set inclusion.

Example 5.3. Figure 5.4 shows three examples of abstract joins. In the second example, the first type $\mathbb{t}_1 = \texttt{node}[10].(\mathbb{i})$ represents a single concrete type, namely $t_1 = \texttt{node}[10].(54)$, which is also in the concretization of $\mathbb{t}_2 = \texttt{node}[10].(\mathbb{i}')$. However, it would be incorrect to have $\mathbb{t}_2$ be the result of the join, because it also represents some types that are not supertypes of t_1, such as $\texttt{node}[10].(0)$.

Theorem 5.3 (Soundness of $\sqcup_{\mathrm{T}}$). *Given $\mathbb{t}_1, \mathbb{t}_2 \in \mathbb{T}^\#$, $v_1^\#, v_2^\# \in \mathbb{D}_{\mathrm{num}}$, for all $(t_1, v_1) \in \gamma_{\mathrm{T,num}}(\mathbb{t}_1, v_1^\#)$ and $(t_2, v_2) \in \gamma_{\mathrm{T,num}}(\mathbb{t}_2, v_2^\#)$ such that $t_\vee = (\mathbb{t}_1, v_1^\#) \sqcup_{\mathrm{T}} (\mathbb{t}_2, v_2^\#)$ is defined:*

$$t_1 \preceq_{\mathbb{T}} t_\vee \text{ and } t_2 \preceq_{\mathbb{T}} t_\vee$$

Proof. Similar to the proof of Theorem 5.2. □

5.3 State abstraction

5.3.1 Type-based shape domain

The type-based shape domain $\mathbb{H}$ describes each variable with an abstract type. In order to also express non-type related constraints over the contents of variables, this abstraction also needs to attach to each variable a symbolic variable denoting its value.

The elements of $\mathbb{H}$ are $(\sigma^\sharp, \Gamma^\sharp)$ pairs, where $\sigma^\sharp \in \mathbb{X} \to \mathbb{V}^\sharp$ is a mapping from variables to symbolic values, and $\Gamma^\sharp \in \mathbb{X} \to \mathbb{T}^\sharp$ is a mapping from variables to abstract types; we call such pairs *abstract stores*. The concrete states represented by an abstract store $(\sigma^\sharp, \Gamma^\sharp)$ are the well-typed states of the form $(\sigma, \Gamma, h, \mathcal{L})$ in which σ is abstracted by $\sigma^\sharp$ and Γ is abstracted by $\Gamma^\sharp$.

$$\begin{aligned}
\gamma_{\mathrm{H}} &: \mathbb{H} \to \mathcal{P}\big(\mathbb{S}_t \times (\mathbb{V}^\sharp \to \mathbb{V})\big) \\
\gamma_{\mathrm{H}}\big(\sigma^\sharp, \Gamma^\sharp\big) &= \left\{ \big((\sigma, \Gamma, h, \mathcal{L}),\ \upsilon\big) \;\middle|\; \begin{array}{l} (\sigma, \Gamma, h, \mathcal{L}) \text{ is well typed under } \upsilon \\ \text{and } \forall x \in \mathbb{X},\ \forall \big(t, \upsilon'\big) \in \gamma_{\mathrm{T}}(\Gamma^\sharp(x)), \Gamma(x) \preceq_{\mathbb{T}} t \\ \text{and } \forall x \in \mathbb{X},\ \sigma(x) = \upsilon(\sigma^\sharp(x)) \end{array} \right\}
\end{aligned}$$

Note that there is no explicit representation of the heap in elements of $\mathbb{H}$; however, possible concrete heaps are constrained by the requirement that the state must be well typed.

This definition does not provide an abstraction suitable for analysis yet; we still need to reason over the possible numerical values denoted by symbolic variables.

5.3.2 Combined shape-numeric abstraction

The abstraction is enriched with numeric constraints on values by using a product of the shape domain with a numeric domain $\mathbb{D}_{\mathrm{num}}$ that has a concretization $\gamma_{\mathrm{num}} : \mathbb{D}_{\mathrm{num}} \to \mathcal{P}\big(\mathbb{V}^\sharp \to \mathbb{V}\big)$.

Definition 5.8 (Shape-numeric abstract domain). Given a numeric domain $\mathbb{D}_{\mathrm{num}}$, the combined shape-numeric abstract domain is:

$$\mathbb{S}^\sharp \overset{\mathrm{def}}{=} \mathbb{H} \times \mathbb{D}_{\mathrm{num}}$$

Its concretization is

$$\begin{aligned}
\gamma_{\mathrm{S}} &: \mathbb{S}^\sharp \to \mathcal{P}\big(\mathbb{S}_t \times (\mathbb{V}^\sharp \to \mathbb{V})\big) \\
\gamma_{\mathrm{S}}(h^\sharp, \upsilon^\sharp) &= \{(s, \upsilon) \in \gamma_{\mathrm{S}}(h^\sharp) \mid \upsilon \in \gamma_{\mathrm{num}}(\upsilon^\sharp)\}
\end{aligned}$$

Example 5.4. Consider the type `node` from our running example. Suppose that there are only two variables in the language: $\mathbb{X} = \{x, y\}$. A possible abstract state $\mathbb{s} \in \mathbb{S}^\sharp$ is the following:

$$\mathbb{s} = (([x \mapsto \mathbb{v}_0,\ y \mapsto \mathbb{v}_1], [x \mapsto \mathtt{node}.(0),\ y \mapsto \mathtt{node}.(0)]), \upsilon^\sharp)$$

where $\upsilon^\sharp$ expresses the numerical predicates $\mathbb{v}_0 \neq 0$ and $\mathbb{v}_1 \neq 0$:

$$\gamma_{\mathrm{num}}(\upsilon^\sharp) = \{\upsilon \in \mathbb{V}^\sharp \to \mathbb{V} \mid \upsilon(\mathbb{v}_0) \neq 0 \wedge \upsilon(\mathbb{v}_1) \neq 0\}$$

This abstract state can be represented graphically as follows:

$$\begin{array}{ll}
x \ \boxed{\mathbb{v}_0 : \mathtt{node}.(0)*} & \mathbb{v}_0 \neq 0 \\
y \ \boxed{\mathbb{v}_1 : \mathtt{node}.(0)*} & \mathbb{v}_1 \neq 0
\end{array}$$

5.4 Abstract semantics

For the combined shape-numeric abstract domain, abstract values are pairs of a symbolic value and a type: $\mathbb{V}_S = \mathbb{V}^\sharp \times \mathbb{T}^\sharp$.

The transfer functions of $\mathbb{S}^\sharp$ are in many ways similar to the typed semantics presented in Section 4.3. However, since the heap is represented only indirectly —through types—, memory reads and writes are also handled through types:

- The result of a memory read is abstracted by an unknown value and the type of the region read; if the types in that region are constrained with predicates, the value is refined using those predicates.
- The result of a memory write is abstracted by the identity, so long as the write operation is well typed; if it is ill-typed, the analysis emits an alarm[1].

To perform these two operations, the analysis must be able, first, to produce a value of a given type, and second, to decide whether a value is consistent with a type.

We call *abstract type checking* the operation of verifying that a value is consistent with a given type:

Definition 5.9 (Abstract type checking). For $\mathbb{t}, \mathbb{u} \in \mathbb{T}^\sharp$ and $v^\sharp \in \mathbb{D}_{\text{num}}$, we denote

$$\mathbb{v} : \mathbb{t} \xhookrightarrow{v^\sharp} \mathbb{u}$$

the fact that value $\mathbb{v}$, of type $\mathbb{t}$, can be safely cast into type $\mathbb{u}$. The rules for abstract type checking are given in Figure 5.5.

The rule SubVal expresses the possibility of *upcasting*, i.e., of turning a type into its supertype. On the contrary, Refine enables *downcasting*: the type of a value may be refined with a predicates that holds on the value. The rule NullPtr expresses that the value zero can be safely cast into any pointer type. Finally, the three rules MakeNamed, MakeProd, MakeArray enable to inductively check named or compound types.

Example 5.5 (Upcasting a pointer). Following Definition 5.5, given some $v^\sharp \in \mathbb{D}_{\text{num}}$, the subtyping relation $(\texttt{int}[10].(\mathbb{i})*, v^\sharp) \sqsubseteq_T (\texttt{int}.(0)*, v^\sharp)$ holds when the numerical constraints $\mathbb{i} \in [0, 39] \wedge \exists k \in \mathbb{N}, \mathbb{i} = 4k$ hold. In that case, $\texttt{int}[10].(\mathbb{i})*$ can be safely cast into an $\texttt{int}.(0)*$, which we denote, for any $\mathbb{v} \in \mathbb{V}^\sharp$:

$$\mathbb{v} : \texttt{int}[10].(\mathbb{i})* \xhookrightarrow{v^\sharp} \texttt{int}.(0)*.$$

Note that querying the numerical domain is necessary to check the safety of this cast.

Example 5.6 (Downcasting to a non-null pointer). Consider again the types from Figure 4.2b:

$$\begin{array}{llll} \mathcal{M} : & \texttt{node\_kind} & \mapsto & \{x : \texttt{word}_4 \mid x \leq 5\} \\ & \texttt{uf} & \mapsto & \texttt{uf}.(0)* \\ & \texttt{dll} & \mapsto & \{x : \texttt{dll}.(0)* \mid x \neq 0\} \times \{x : \texttt{dll}.(0)* \mid x \neq 0\} \\ & \texttt{node} & \mapsto & \texttt{node\_kind} \times \texttt{dll} \times \texttt{uf} \end{array}$$

Given a value $\mathbb{v}$ of type $\texttt{node}.(4)*$, it can be cast into a $\texttt{dll}.(0)*$ by rule SubVal. In addition, if the value is non-zero, i.e., if $v^\sharp \models \mathbb{v} \neq 0$, it can be downcast into a non-null pointer type by the Refine rule:

$$\mathbb{v} : \texttt{node}.(4)* \xhookrightarrow{v^\sharp} \{x : \texttt{dll}.(0)* \mid x \neq 0\}$$

[1] As is usual in static analysis, the emission of an alarm signals that the analysis results after this point are only correct on traces where the error did not occur.

$$\frac{(\mathbb{t}, v^\#) \sqsubseteq_{\mathrm{T}} (\mathbb{u}, v^\#)}{\mathbb{v} : \mathbb{t} \overset{v^\#}{\hookrightarrow} \mathbb{u}}\ \textsc{SubVal} \qquad \frac{v^\# \models p(\mathbb{v})}{\mathbb{v} : \mathbb{t} \overset{v^\#}{\hookrightarrow} \{x : \mathbb{t} \mid p(x)\}}\ \textsc{Refine} \qquad \frac{v^\# \models \mathbb{v} = 0 \qquad u \in \mathbb{T}}{\mathbb{v} : \mathbb{t} \overset{v^\#}{\hookrightarrow} u.(0)*}\ \textsc{NullPtr}$$

$$\frac{}{\mathbb{v} : \mathcal{M}(\mathsf{n}) \overset{v^\#}{\hookrightarrow} \mathsf{n}}\ \textsc{MakeNamed} \qquad \frac{\mathbb{v}_1 : \mathbb{t}_1 \overset{v^\#}{\hookrightarrow} \mathbb{t}_a \qquad \mathbb{v}_2 : \mathbb{t}_2 \overset{v^\#}{\hookrightarrow} \mathbb{t}_b}{\mathbb{v}_3 : \mathbb{t}_3 \xhookrightarrow{v^\#[\mathbb{v}_3 = \mathbb{v}_1 :: \mathbb{v}_2]} \mathbb{t}_a \times \mathbb{t}_b}\ \textsc{MakeProd}$$

$$\frac{\mathbb{v}_1 : \mathbb{t}_1 \overset{v^\#}{\hookrightarrow} \mathbb{t} \qquad \mathbb{v}_2 : \mathbb{t}_2 \overset{v^\#}{\hookrightarrow} \mathbb{t} \qquad \cdots \qquad \mathbb{v}_n : \mathbb{t}_n \overset{v^\#}{\hookrightarrow} \mathbb{t}}{\mathbb{v}' : \mathbb{t}' \xhookrightarrow{v^\#[\mathbb{v}' = \mathbb{v}_1 :: \cdots :: \mathbb{v}_n]} \mathbb{t}[n]}\ \textsc{MakeArray}$$

Figure 5.5: Type casting rules.

Example 5.7 (Obtaining a named type from an unnamed one). A value of type $\{x : \mathtt{word}_4 \mid x \leq 5\}$ can be attributed type `node_kind` through rule MakeNamed.

Abstract type checking is sound in the sense that in the context of an abstract state, if abstract type checking succeeds then all concrete values in the interpretation of the initial type are also in the interpretation of the final type:

Theorem 5.4 (Soundness of abstract type checking). *Let* $\mathbb{s} = (h^\#, v^\#) \in \mathbb{S}^\#$, $\mathbb{v} \in \mathbb{V}^\#$ *and* $\mathbb{t}, \mathbb{u} \in \mathbb{T}^\#$. *If* $\mathbb{v} : \mathbb{t} \overset{v^\#}{\hookrightarrow} \mathbb{u}$, *then for all* $((\sigma, \Gamma, h, \mathcal{L}),\ v) \in \gamma_{\mathrm{S}}(\mathbb{s})$, *for all* $t, u \in \mathbb{T}$ *such that* $(t, v) \in \gamma_{\mathrm{T}}(\mathbb{t}) \wedge (u, v) \in \gamma_{\mathrm{T}}(\mathbb{u})$*:*

$$v(\mathbb{v}) \in (\!| t |\!)_{\mathcal{L},v} \implies v(\mathbb{v}) \in (\!| u |\!)_{\mathcal{L},v}$$

Proof. By induction on the syntax tree of type u, using the definition of the interpretation (Definition 4.8), and the soundness of abstract subtyping (Theorem 5.2). □

5.4.1 Abstract semantics of expressions

Figure 5.6, page 60 summarizes the rules for abstract semantics of expressions in $\mathbb{S}^\#$. We proceed below to precisions and clarifications. Intuitively, this semantics simultaneously performs typing and over-approximates the possible numerical values in the program.

Constants Constant expressions are given the type ⊤ (rule Const$^\#$).

Variables A variable is evaluated using the abstract value store and the abstract type store (rule Env$^\#$).

Abstract addition and subtraction Abstract addition and subtraction follow rules similar to the expression typing rules given in the typed semantics, albeit on abstract types. If one of the operands of an addition is a pointer—possibly refined with a predicate—, and the other operand is such that the resulting offset is not out of bounds, then the addition is considered pointer arithmetic and the resulting type is a pointer with the new offset. The resulting value is assumed non-zero (assumption $\mathbb{v}_3 \neq 0$ in the rule conclusion), due to the property that well-typed pointer arithmetic cannot result in a null pointer.

Pointer arithmetic can also be performed using the subtraction operation (rule SUBR$^\#$), but the pointer can only be the right operand.

Note that subtraction is also sometimes used on two pointers into the same block in order to compute an offset. This abstract domain cannot compute the result of such operations precisely in general, as two pointers can have the same type and still point to different memory regions. In our analysis, the user can mark a type as covering only a single memory region, in which case the difference between two pointers to this type is computed as the difference between the pointers' offsets. For example, the type $\texttt{Task}[\mathbb{m}]$ in the kernel types of Figure 9.1, p. 123, could be marked thus (there is only one array of tasks).

For both additions and subtractions, when the premises of rules ADDR$^\#$, ADDL$^\#$ or SUBR$^\#$ do not hold, then the BINOP$^\#$ rule applies.

Other arithmetic and comparison operators. All other operators use the abstract functions of the numerical domain and return a result of type $\top$, regardless of the types of their operands (rule BINOP$^\#$).

Memory reads The result of a memory read is based on the type of the address read. If the address type is a pointer type, then the type of the pointed region is determined by the "deref" function. The "deref" function takes an abstract type and a size ℓ and, if applicable, returns the abstract type of the pointed region of size ℓ.

Definition 5.10 ("deref" function). Given $\mathcal{v}^\# \in \mathbb{D}_{\text{num}}$, the *type dereferencing operator* $\text{deref}_{\mathcal{v}^\#} : \mathbb{T}^\# \times \mathbb{N} \rightharpoonup \mathbb{T}$ is the partial function defined inductively as follows:

$$\begin{aligned}
\text{deref}_{\mathcal{v}^\#}(t.(0)*, \ell) &= t && \text{if } \text{size}(t) = \ell \\
\text{deref}_{\mathcal{v}^\#}(t.(i)*, \ell) &= \text{deref}_{\mathcal{v}^\#}(w(t.(i))*) && \text{if } i \in \mathbb{N} \\
\text{deref}_{\mathcal{v}^\#}(t[\mathbb{s}].(\mathbb{i})*, \ell) &= \text{deref}_{\mathcal{v}^\#}(u, \ell) && \text{where } u = \text{conc}_{\mathcal{v}^\#}(t[\mathbb{s}].(\mathbb{i})*), \text{ if } u \neq \texttt{word}_{\mathcal{W}} \\
\text{deref}_{\mathcal{v}^\#}(t[\mathbb{s}].(\mathbb{i})*, \ell) &\text{ is undefined} && \text{otherwise} \\
\text{deref}_{\mathcal{v}^\#}(t.(\mathbb{i})*, \ell) &= \text{deref}_{\mathcal{v}^\#}(u, \ell) && \text{where } u = \text{conc}_{\mathcal{v}^\#}(t.(\mathbb{i})*), \text{ if } u \neq \texttt{word}_{\mathcal{W}} \\
\text{deref}_{\mathcal{v}^\#}(t.(\mathbb{i})*, \ell) &\text{ is undefined} && \text{otherwise} \\
\text{deref}_{\mathcal{v}^\#}(\mathsf{n}, \ell) &= \text{deref}_{\mathcal{v}^\#}(\mathcal{M}(\mathsf{n}), \ell) \\
\text{deref}_{\mathcal{v}^\#}(\{x : t \mid p(x)\}, \ell) &= \text{deref}_{\mathcal{v}^\#}(t, \ell) \\
\text{deref}_{\mathcal{v}^\#}(\texttt{word}_n, \ell) &\text{ is undefined} \\
\text{deref}_{\mathcal{v}^\#}(t_1 \times t_2, \ell) &\text{ is undefined} \\
\text{deref}_{\mathcal{v}^\#}(t[n], \ell) &\text{ is undefined}
\end{aligned}$$

The result of reading a value of length ℓ in memory at address $\mathbb{a}$ of type $\mathbb{u}$ (rule LOAD$^\#$) is determined as follows. If the read is well typed, i.e., if $\text{deref}_{\mathcal{v}^\#}(\mathbb{u}, \ell)$ is defined, and the address is not null, i.e., $\mathcal{v}^\# \models \mathbb{a} \neq 0$, then the analysis creates a fresh symbolic variable $\mathbb{v}$. This value is returned with type $\mathbb{t}$, the type of the region read. In addition, $\mathbb{v}$ is refined with the predicates present in type $\mathbb{t}$. This is expressed in the rule using abstract type checking: the result value $\mathbb{v}$ can safely be cast from the generic type $\texttt{word}_\ell$ to the type t:

$$\mathbb{v} : \texttt{word}_\ell \overset{\mathcal{v}_1^\#}{\hookrightarrow} \mathbb{t}$$

Error cases When evaluating an expression, if none of the rules of Figure 5.6 applies, then the result of the expression evaluation is a completely unknown value and an unknown state: the analysis loses all precision. This may happen when applying a binary operator to two bit vectors of different sizes, or when the well-typedness of a load cannot be ensured. In a practical implementation of the analysis,

$$\dfrac{c \in \mathbb{V}_n \qquad \mathbb{s} = ((\sigma^\sharp, \Gamma^\sharp), \upsilon^\sharp) \qquad \mathbb{v} \text{ fresh}}{\mathcal{E}\llbracket c \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = \big((\mathbb{v}, \mathtt{word}_n),\ ((\sigma^\sharp, \Gamma^\sharp),\ \upsilon^\sharp[\mathbb{v} = c])\big)}\ \textsc{Const}^\sharp \qquad \dfrac{\mathbb{s} = ((\sigma^\sharp, \Gamma^\sharp), \upsilon^\sharp)}{\mathcal{E}\llbracket x \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = \big((\sigma^\sharp(x), \Gamma^\sharp(x)),\ \mathbb{s}\big)}\ \textsc{Env}^\sharp$$

$$\dfrac{\begin{array}{c}\mathcal{E}\llbracket e_1 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = ((\mathbb{v}_1, \{x : u.(\mathbb{i})* \mid p(x)\}),\ \mathbb{s}_1) \\ \mathcal{E}\llbracket e_2 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s}_1 = \big((\mathbb{v}_2, \mathbb{t}_2),\ ((\sigma^\sharp_2, \Gamma^\sharp_2), \upsilon^\sharp_2)\big) \qquad \upsilon^\sharp_2 \vDash \mathbb{v}_1 \neq 0 \wedge 0 \leq \mathbb{i} + \mathbb{v}_2 < \mathrm{size}_{\upsilon^\sharp}(u) \qquad \mathbb{v}_3, \mathbb{j} \text{ fresh}\end{array}}{\mathcal{E}\llbracket e_1 + e_2 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = \left((\mathbb{v}_3, u.(\mathbb{j})*),\ \left((\sigma^\sharp_2, \Gamma^\sharp_2), \upsilon^\sharp_2\left[\begin{array}{l} \mathbb{v}_3 = \mathbb{v}_1 + \mathbb{v}_2 \\ \wedge\ \mathbb{v}_3 \neq 0 \wedge \mathbb{j} = \mathbb{i} + \mathbb{v}_2 \end{array}\right]\right)\right)}\ \textsc{AddR}^\sharp$$

$$\dfrac{\begin{array}{c}\mathcal{E}\llbracket e_2 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s}_1 = \big((\mathbb{v}_2, \{x : u.(\mathbb{i})* \mid p(x)\}),\ ((\sigma^\sharp_2, \Gamma^\sharp_2), \upsilon^\sharp_2)\big) \\ \mathcal{E}\llbracket e_1 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = ((\mathbb{v}_1, \mathbb{t}_1),\ \mathbb{s}_1) \qquad \upsilon^\sharp_2 \vDash \mathbb{v}_2 \neq 0 \wedge 0 \leq \mathbb{i} + \mathbb{v}_1 < \mathrm{size}_{\upsilon^\sharp}(u) \qquad \mathbb{v}_3, \mathbb{j} \text{ fresh}\end{array}}{\mathcal{E}\llbracket e_1 + e_2 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = \left((\mathbb{v}_3, u.(\mathbb{j})*),\ \left((\sigma^\sharp_2, \Gamma^\sharp_2), \upsilon^\sharp_2\left[\begin{array}{l} \mathbb{v}_3 = \mathbb{v}_1 + \mathbb{v}_2 \\ \wedge\ \mathbb{v}_3 \neq 0 \wedge \mathbb{j} = \mathbb{i} + \mathbb{v}_1 \end{array}\right]\right)\right)}\ \textsc{AddL}^\sharp$$

$$\dfrac{\begin{array}{c}\mathcal{E}\llbracket e_1 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = ((\mathbb{v}_1, \{x : u.(\mathbb{i})* \mid p(x)\}),\ \mathbb{s}_1) \\ \mathcal{E}\llbracket e_2 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s}_1 = \big((\mathbb{v}_2, \mathbb{t}_2),\ ((\sigma^\sharp_2, \Gamma^\sharp_2), \upsilon^\sharp_2)\big) \qquad \upsilon^\sharp_2 \vDash \mathbb{v}_1 \neq 0 \wedge 0 \leq \mathbb{i} - \mathbb{v}_2 < \mathrm{size}_{\upsilon^\sharp}(u) \qquad \mathbb{v}_3, \mathbb{j} \text{ fresh}\end{array}}{\mathcal{E}\llbracket e_1 - e_2 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = \left((\mathbb{v}_3, u.(\mathbb{j})*),\ \left((\sigma^\sharp_2, \Gamma^\sharp_2), \upsilon^\sharp_2\left[\begin{array}{l} \mathbb{v}_3 = \mathbb{v}_1 - \mathbb{v}_2 \\ \wedge\ \mathbb{v}_3 \neq 0 \wedge \mathbb{j} = \mathbb{i} - \mathbb{v}_2 \end{array}\right]\right)\right)}\ \textsc{SubR}^\sharp$$

$$\dfrac{\begin{array}{c}\diamond \in \{+, -, \times, /, \cdots\} \\ \mathcal{E}\llbracket e_1 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = ((\mathbb{v}_1, \mathbb{t}_1),\ \mathbb{s}_1) \qquad \mathcal{E}\llbracket e_2 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s}_1 = \big((\mathbb{v}_2, \mathbb{t}_2),\ ((\sigma^\sharp_2, \Gamma^\sharp_2), \upsilon^\sharp_2)\big) \qquad \mathrm{size}(\mathbb{t}_1) = \mathrm{size}(\mathbb{t}_2) \qquad \mathbb{v}_3 \text{ fresh}\end{array}}{\mathcal{E}\llbracket e_1 \diamond e_2 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = \big((\mathbb{v}_3, \mathtt{word}_{\mathrm{size}(\mathbb{t}_1)}),\ ((\sigma^\sharp_2, \Gamma^\sharp_2), \upsilon^\sharp_2[\mathbb{v}_3 = \mathbb{v}_1 \diamond \mathbb{v}_2])\big)}\ \textsc{Binop}^\sharp$$

$$\dfrac{\mathcal{E}\llbracket e \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = \big((\mathbb{a}, \mathbb{u}),\ ((\sigma^\sharp_1, \Gamma^\sharp_1), \upsilon^\sharp_1)\big) \qquad \mathrm{deref}_{\upsilon^\sharp_1}(\mathbb{u}, \ell) = \mathbb{t} \qquad \upsilon^\sharp_1 \vDash \mathbb{a} \neq 0 \qquad \mathbb{v} : \mathtt{word}_\ell \xhookrightarrow{\upsilon^\sharp_1} \mathbb{t} \qquad \mathbb{v} \text{ fresh}}{\mathcal{E}\llbracket *_\ell e \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = \big((\mathbb{v}, \mathbb{t}),\ ((\sigma^\sharp_1, \Gamma^\sharp_1), \upsilon^\sharp_1)\big)}\ \textsc{Load}^\sharp$$

Figure 5.6: Abstract semantics of expressions in $\mathbb{S}^\sharp$.

this will be reported as an alarm with information to help understand the origin of the alarm (such as location and state information) and the analysis will carry on assuming the error was a false alarm. This way, if the user can verify by other means that it was indeed a false alarm, the analysis still delivers useful results.

5.4.2 Soundness of expression semantics

The numerical component of the result of abstract expression evaluation is sound with respect to the concrete semantics. As for the type component, although it does not cover all possible types computed by concrete expression typing, it corresponds to a subset of those.

Theorem 5.5 (Soundness of abstract expression evaluation). *Let $\mathbb{s} \in \mathbb{S}^\sharp$ and $e \in$ expr. Let $((\mathbb{v}, \mathbb{t}), \mathbb{s}') = \mathcal{E}\llbracket e \rrbracket^\sharp_{\mathrm{S}} \mathbb{s}$. For all $((\sigma, \Gamma, h, \mathcal{L}), \upsilon) \in \gamma_{\mathrm{S}}(\mathbb{s})$, if $(\sigma, \Gamma, h, \mathcal{L})$ is well typed under υ, then:*

1. *Expression evaluation does not modify the concrete state, therefore all states abstracted by $\mathbb{s}$ are also*

abstracted by $\mathbb{s}'$*:*

$$\forall (s, v) \in \gamma_S(\mathbb{s}),\ \exists v' : \mathbb{V}^\# \to \mathbb{V},\ (s, v') \in \gamma_S(\mathbb{s}')$$

2. *The numerical component of the result is sound:*

$$\{\mathcal{E}[\![e]\!](\sigma, h) \mid ((\sigma, \Gamma, h, \mathcal{L}),\ v) \in \gamma_S(\mathbb{s})\} \subseteq \{v(\mathbb{v}) \mid \exists s \in \mathbb{S}_t,\ (s, v) \in \gamma_S(\mathbb{s})\}$$

3. *The concrete types abstracted by* $\mathbb{t}$ *are possible types for* e *(but not all types):*

$$\gamma_T(\mathbb{t}) \subseteq \{(t, v) \mid \forall s \in \mathbb{S}_t,\ (s, v) \in \gamma_S(\mathbb{s}) \implies s \vdash_v e : t\}$$

Proof. By induction on expression syntax. Each rule of abstract expression evaluation is similar to one of the rules in concrete expression typing (Figure 4.4), so it can be shown that each rule yields a similar type judgment. Soundness of the numerical component is straightforward from the definitions and the fact that the initial state is well typed. □

We see here that the abstract expression semantics is sound with regard to the untyped semantics $\mathcal{E}[\![\cdot]\!]$ (defined on p. 22), but not to the typed semantics $[\![\cdot]\!]_t$ (defined on p. 43), although the type component enjoys a form of completeness with regard to the typed semantics.

Indeed, the typed semantics non-deterministically computes all possible types of an expression. In contrast, the abstract semantics $\mathcal{E}[\![\cdot]\!]^\#_S$ only computes some of the possible types, and uses the typing invariants to maintain both soundness and a certain level of precision.

Example 5.8. Consider the abstract state $\mathbb{s} = ((\sigma^\#, \Gamma^\#), v^\#)$ from Example 5.4. Evaluating the expression $*_4 x$ in $\mathbb{s}$ results in the following (value, state) pair:

$$((\mathbb{v}_2, \texttt{node\_kind}),\ ((\sigma^\#, \Gamma^\#), v^\#[\mathbb{v}_2 \leq 5]))$$

by application of the Load$^\#$ rule. There is only one derived type. By contrast, in the typed semantics the same expression would be attributed an infinity of types. These types would include `node_kind`, but also $\{x : \texttt{node\_kind} \mid x \leq 5\}$, $\{x : \texttt{node\_kind} \mid x \leq 5 \land x \neq 42\}$, $\texttt{word}_4$, $\{x : \texttt{word}_4 \mid x \leq 5\}$, etc. by applications of rules Refine and SubVal.

5.4.3 Abstract semantics of statements

Transfer functions for atomic statements

There is one rule for each type of atomic statement, gathered in Figure 5.7. Rules for compound statements are generic and given in Section 3.4. Each rule specifies the semantics of one kind of statement under a number of premises. As with expression evaluation, when none of the rules applies, the analysis returns an entirely unknown state (or more practically, emits an alarm).

Assignment The result of an assignment is computed (rule Assign$^\#$) by evaluating the expression into a symbolic variable and a type, and updating the abstract value store and the abstract type store accordingly.

Memory write The result of evaluating the statement $*_t e_1 := e_2$ is determined as follows. First e_1 is evaluated into $(\mathbb{a}, \mathbb{t}_a)$ and e_2 into $(\mathbb{v}, \mathbb{t}_v)$. If the destination address is not null, and the address type can

$$\frac{\mathcal{E}\llbracket e \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = \big((\mathbb{v}, \mathbb{t}),\ ((\sigma^\sharp_1, \Gamma^\sharp_1),\ \upsilon^\sharp_1)\big)}{\llbracket x\ := e \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = ((\sigma^\sharp_1[x \leftarrow \mathbb{v}], \Gamma^\sharp_1[x \leftarrow \mathbb{t}]),\ \upsilon^\sharp_1)}\ \textsc{Assign}^\sharp$$

$$\frac{\begin{array}{c}\mathcal{E}\llbracket e_1 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = ((\mathbb{a}, \mathbb{t}_a),\ \mathbb{s}_1) \\ \mathcal{E}\llbracket e_2 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s}_1 = \big((\mathbb{v}, \mathbb{t}_v),\ ((\sigma^\sharp_2, \Gamma^\sharp_2), \upsilon^\sharp_2)\big) \qquad \mathrm{deref}_{\upsilon^\sharp_2}(\mathbb{t}_a, \ell) = \mathbb{t} \qquad \upsilon^\sharp_2 \vDash \mathbb{a} \neq 0 \qquad \mathbb{v}\ :\ \mathbb{t}_v \xhookrightarrow{\upsilon^\sharp_2} \mathbb{t}\end{array}}{\llbracket *_\ell e_1\ := e_2 \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = ((\sigma^\sharp_2, \Gamma^\sharp_2), \upsilon^\sharp_2)}\ \textsc{Store}^\sharp$$

$$\frac{\mathcal{E}\llbracket e \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = \big((\mathbb{v}_\ell, \mathbb{t}_\ell),\ ((\sigma^\sharp_1, \Gamma^\sharp_1),\ \upsilon^\sharp_1)\big) \qquad \upsilon^\sharp_1 \vDash \mathbb{v}_\ell = \mathrm{size}(t) \qquad \mathbb{a}, \mathbb{v}\ \text{fresh} \qquad \mathbb{v}\ :\ \texttt{word}_{\mathrm{size}(t)} \xhookrightarrow{\upsilon^\sharp_1} t}{\llbracket x\ := \mathbf{malloc}_t(e) \rrbracket^\sharp_{\mathrm{S}} \mathbb{s} = ((\sigma^\sharp_1[x \leftarrow \mathbb{a}], \Gamma^\sharp_1[x \leftarrow t.(0)*]),\ \upsilon^\sharp_1)}\ \textsc{Malloc}^\sharp$$

Figure 5.7: Abstract semantics of simple statements in $\mathbb{S}^\sharp$.

be dereferenced (i.e. $\mathrm{deref}_{\upsilon^\sharp_2}(\mathbb{t}_a, \ell)$ is defined), then the analysis checks that the typed value written can be safely converted into a $\mathbb{t}$, i.e., it checks that:

$$\mathbb{v}\ :\ \mathbb{t}_v \xhookrightarrow{\upsilon^\sharp_2} \mathbb{t}$$

If that is the case, then the memory write preserves the heap typing, and the abstract state is returned unmodified. (Recall that our abstract domain does not represent any constraints on the heap, other than the fact that it is well typed.)

Memory allocation We give the possibility to the user of the analysis to provide an *expected type* for the allocated region, as a hint to the static analysis. This is not part of the While-memory syntax, nor of its concrete semantics; but in the abstract operator for **malloc** we denote the presence of such a hint with a subscript type: $\llbracket x\ := \mathbf{malloc}_t(e) \rrbracket^\sharp$, where $t \in \mathbb{T}$. To compute the effect of $x\ := \mathbf{malloc}_t(e)$, the analysis simply assigns a fresh, unconstrained symbolic variable $\mathbb{a}$ to x, and gives it the type $t.(0)*$. In addition, it checks that the type hint t can be soundly used in this way: it creates a fresh, unconstrained symbolic variable $\mathbb{v}$, to account for the unknown contents of the newly allocated region, and performs the abstract type checking:

$$\mathbb{v}\ :\ \texttt{word}_{\mathrm{size}(t)} \xhookrightarrow{\upsilon^\sharp_1} t$$

This boils down to asking the question: “Can this unknown memory region be given type t without violating typing?” If the answer is no, then this is an error case and the analysis emits an alarm: the type hint cannot be used in a sound way.

Example 5.9 (Imprecision of memory allocation in $\mathbb{S}^\sharp$). The transfer function for memory allocation may induce a significant loss of precision on a common code pattern, namely initialization of allocated objects, resulting in the failure to verify many programs. Consider for example the following program,

$$\begin{array}{ll} x & \boxed{\mathbb{v}_0 \ : \ \texttt{node[10].(}\mathbb{i}\texttt{)*}} \\ y & \boxed{\mathbb{v}_1 \ : \ \texttt{word}_8} \end{array} \quad \begin{array}{l} \mathbb{v}_0 \neq 0 \\ \mathbb{v}_1 > 5 \\ \mathbb{i} \in [32, 160[\\ \mathbb{i} \equiv 6 \pmod{16} \end{array} \quad \sqsubseteq \quad \begin{array}{ll} x & \boxed{\mathbb{v}'_0 \ : \ \texttt{dll.(2)}} \\ y & \boxed{\mathbb{v}'_1 \ : \ \texttt{word}_8} \end{array} \quad \begin{array}{l} \mathbb{v}'_0 \neq 0 \\ \mathbb{v}'_1 \neq 0 \end{array}$$

Figure 5.8: Example of abstract inclusion between two abstract states. A valid renaming function is Φ such that $\Phi(\mathbb{v}'_0) = \mathbb{v}_0$, $\Phi(\mathbb{v}'_1) = \mathbb{v}_1$ and Φ is the identity everywhere else.

that allocates and initializes a region of type `dll`:

$$\begin{array}{l} n \ := \mathbf{malloc}_{\texttt{dll}}(8); \\ *_4 n \ := n; \\ *_4(n+4) \ := n \end{array}$$

This program is reasonable: it allocates a doubly-linked list node, and initializes its `next` and `prev` fields to point to itself.

However, the analysis using the abstract domain $\mathbb{S}^\#$ will emit an alarm on the first line, because it would be unsound to give the type `dll` to a region with unknown contents. If we remove the type hint (or, equivalently, replace it with $\texttt{word}_8$), then no alarm is emitted, but the information that n points to a region of type `dll` is lost.

This example illustrates the fact that sometimes, precision can be gained by delaying the type-related checks. Our staged predicate domain (Chapter 6) will exploit this intuition to gain precision.

Transfer function for conditionals

The function $\mathbf{guard}_{\mathrm{S}} : \mathbb{V}_{\mathrm{S}} \times \mathbb{S}^\# \to \mathbb{S}^\#$ ignores the type of its operand, and simply adds to the numerical abstract state the constraint that the symbolic variable in its argument should be non-zero:

$$\mathbf{guard}_{\mathrm{S}}((\mathbb{v}, \mathbb{t}),\ ((\sigma^\#, \Gamma^\#), \upsilon^\#)) = ((\sigma^\#, \Gamma^\#), \upsilon^\#[\mathbb{v} \neq 0])$$

Abstract inclusion and join

Abstract inclusion and abstract join are computed point-wise, based on the abstract type join and the join of the numerical domain.

Definition 5.11 (Valid variable renamings in $\mathbb{S}^\#$). $\Phi : \mathbb{V}^\# \to \mathbb{V}^\#$ is a renaming from the variables of $\mathbb{s}_2 = ((\sigma^\#_2, \Gamma^\#_2),\ \upsilon^\#_2)$ to the variables of $\mathbb{s}_1 = ((\sigma^\#_1, \Gamma^\#_1),\ \upsilon^\#_1)$ if:

$$\forall x \in \mathbb{X}, \Phi(\sigma^\#_2(x)) = \sigma^\#_1(x)$$

Note that from this definition, it is easy to deduce an algorithm to construct a renaming.

Definition 5.12 (Abstract inclusion in $\mathbb{S}^\#$, with renaming). Let $\mathbb{s}_1 = ((\sigma^\#_1, \Gamma^\#_1),\ \upsilon^\#_1)$ and $\mathbb{s}_2 = ((\sigma^\#_2, \Gamma^\#_2),\ \upsilon^\#_2)$ be two elements of $\mathbb{S}^\#$. Then $\mathbb{s}_1 \sqsubseteq_{\mathrm{S},\Phi} \mathbb{s}_2$ holds if and only if:

$$\upsilon^\#_1 \sqsubseteq_{\mathrm{num},\Phi} \upsilon^\#_2$$

and

$$\forall x \in \mathbb{X}, (\Gamma^\#_1(x), \upsilon^\#_1) \sqsubseteq_{\mathrm{T}} (\Gamma^\#_2(x), \upsilon^\#_2).$$

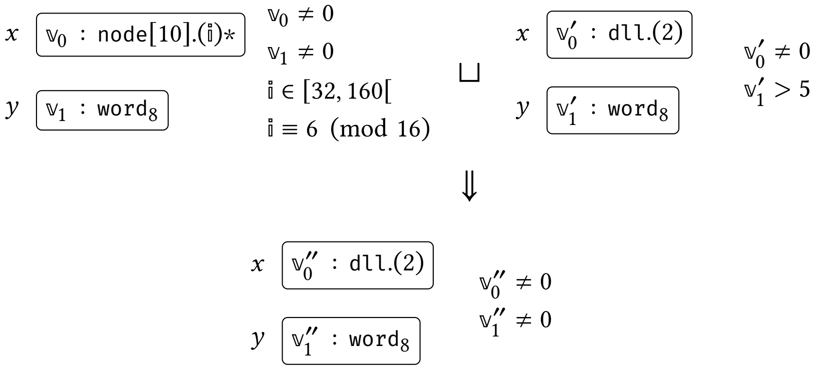

x $\mathbb{v}_0$: node[10].(𝕚)*
y $\mathbb{v}_1$: word$_8$
$\mathbb{v}_0 \neq 0$
$\mathbb{v}_1 \neq 0$
$\mathbb{i} \in [32, 160[$
$\mathbb{i} \equiv 6 \pmod{16}$
$\sqcup$
x $\mathbb{v}'_0$: dll.(2)
y $\mathbb{v}'_1$: word$_8$
$\mathbb{v}'_0 \neq 0$
$\mathbb{v}'_1 > 5$
$\Downarrow$
x $\mathbb{v}''_0$: dll.(2)
y $\mathbb{v}''_1$: word$_8$
$\mathbb{v}''_0 \neq 0$
$\mathbb{v}''_1 \neq 0$

Figure 5.9: Example of join between two abstract states. A possible valid two-way renaming is $\Psi = \{(\mathbb{v}_0, \mathbb{v}'_0, \mathbb{v}''_0),\ (\mathbb{v}_1, \mathbb{v}'_1, \mathbb{v}''_1)\}$.

Definition 5.13 (Abstract inclusion in $\mathbb{S}^\sharp$). For any two abstract states $\mathbb{s}_1, \mathbb{s}_2 \in \mathbb{S}^\sharp$, the relation $\mathbb{s}_1 \sqsubseteq_S \mathbb{s}_2$ holds if and only if there exists Φ a renaming from the variables of $\mathbb{s}_2$ to the variables of $\mathbb{s}_1$ such that $\mathbb{s}_1 \sqsubseteq_{S,\Phi} \mathbb{s}_2$.

Example 5.10. Figure 5.8 shows an example of an abstract state included in another, along with a valid renaming that enables to verify it.

Note that instead of using the abstract subtyping operator $\sqsubseteq_T$, we could have used abstract type checking, which is more precise. Indeed, abstract type checking uses abstract subtyping, but is also able to inspect compound types, and to perform downcasts. In our experiments (Section 7.3 and Chapter 10), we found that the precision given by abstract subtyping was sufficient for our purposes. It could be useful to implement abstract inclusion using abstract type checking and evaluate the precision gains against the performance cost. In any case, abstract type checking cannot be used to construct the join operation.

Definition 5.14 (Construction of a two-way variable renaming in $\mathbb{S}^\sharp$). $\Psi \subseteq (\mathbb{V}^\sharp)^3$ is a valid two-way renaming between $\mathbb{s}_1$ and $\mathbb{s}_2$ if for all $x \in \mathbb{X}$ there exists $\mathbb{v}_x \in \mathbb{V}^\sharp$ such that:

$$\Psi \subseteq \bigcup_{x \in \mathbb{X}} (\sigma_1^\sharp(x), \sigma_2^\sharp(x), \mathbb{v}_x)$$

where for all $x \in \mathbb{X}$, $\mathbb{v}_x$ is does not appear in $\mathbb{s}_1$ and $\mathbb{s}_2$, and does not appear elsewhere in Ψ.

Definition 5.15 (Abstract join in $\mathbb{S}^\sharp$). Let $\mathbb{s}_1 = ((\sigma_1^\sharp, \Gamma_1^\sharp),\ \nu_1^\sharp)$ and $\mathbb{s}_2 = ((\sigma_2^\sharp, \Gamma_2^\sharp),\ \nu_2^\sharp)$ be two elements of $\mathbb{S}^\sharp$. Let $\Psi \subseteq (\mathbb{V}^\sharp)^3$ be a valid two-way renaming between $\mathbb{s}_1$ and $\mathbb{s}_2$. The join $\mathbb{s}_1 \sqcup_{S,\Psi} \mathbb{s}_2$ is defined as $\mathbb{s}_3 = ((\sigma_3^\sharp, \Gamma_3^\sharp),\ \nu_3^\sharp)$, where:

- $\sigma_3^\sharp$ is such that:
$$\forall x \in \mathbb{X},\ (\sigma_1^\sharp(x), \sigma_2^\sharp(x), \sigma_3^\sharp(x)) \in \Psi$$
- $\Gamma_3^\sharp$ is such that:
$$\forall x \in \mathbb{X},\ \Gamma_3^\sharp(x) = (\Gamma_1^\sharp(x), \nu_1^\sharp) \sqcup_T (\Gamma_2^\sharp(x), \nu_2^\sharp)$$
- The numerical abstract state $\nu_3^\sharp$ is the result of using the join operator from the numerical domain:
$$\nu_3^\sharp = \nu_1^\sharp \sqcup_{\text{num},\Psi} \nu_2^\sharp.$$

$$\begin{tikzcd}
\mathbb{S}^\sharp \arrow[r, "{[\![\mathrm{P}]\!]^\sharp_{\mathrm{S}}}"] \arrow[d, "\gamma_{\mathrm{S}}"'] & \mathbb{S}^\sharp \arrow[d, "\gamma_{\mathrm{S}}"] \\
\mathcal{P}\big(\mathbb{S}_t \times (\mathbb{V}^\sharp \to \mathbb{V})\big) \arrow[r, dashed, "{[\![\mathrm{P}]\!]_t}"] \arrow[d, "\mathrm{untyp}"'] & \mathcal{P}\big(\mathbb{S}_t \times (\mathbb{V}^\sharp \to \mathbb{V})\big) \arrow[d, "\mathrm{untyp}"] \\
\mathcal{P}(\mathbb{S}) \arrow[r, "{[\![\mathrm{P}]\!]}"'] & \mathcal{P}(\mathbb{S})
\end{tikzcd}$$

Figure 5.10: Graphical representation of semantics and concretizations.

Example 5.11. Figure 5.9 shows an example of joining two abstract states, along with the associated renaming.

Theorem 5.6 (Soundness of $\sqcup_{\mathrm{S},\Psi}$). *Given* $\mathbb{s}_1, \mathbb{s}_2 \in \mathbb{S}^\sharp$ *and* Ψ *a two-way renaming between* $\mathbb{s}_1$ *and* $\mathbb{s}_2$*, if* $\mathbb{s}_3 = \mathbb{s}_1 \sqcup_{\mathrm{S},\Psi} \mathbb{s}_2$*, then:*

$$\forall (s_1, v_1) \in \gamma_{\mathrm{S}}(\mathbb{s}_1),\ (s_1,\ v_1 \circ \Phi_1) \in \gamma_{\mathrm{S}}(\mathbb{s}_3)$$
$$\forall (s_2, v_2) \in \gamma_{\mathrm{S}}(\mathbb{s}_2),\ (s_2,\ v_2 \circ \Phi_2) \in \gamma_{\mathrm{S}}(\mathbb{s}_3)$$

where Φ_1 *and* Φ_2 *are defined by:*

$$(\Phi_1(\mathbb{v}''), \Phi_2(\mathbb{v}'')) = (\mathbb{v}, \mathbb{v}') \iff (\mathbb{v}, \mathbb{v}', \mathbb{v}'') \in \Psi$$

Widening

The abstract join on types $\sqcup_{\mathrm{T},\Psi}$ is also a widening operator, in the sense that it does not induce infinite increasing chains (Definition 3.16). This is due to the fact that the weakening chain of an address type (Definition 4.3) is always finite. Therefore, the widening of domain $\mathbb{S}^\sharp$ uses the abstract join on types, and the widening of the numerical domain:

Definition 5.16 (Widening in $\mathbb{S}^\sharp$). The widening $\triangledown_{\mathrm{S},\Psi}$ is defined identically to the abstract join, except that $\sqcup_{\mathrm{num},\Psi}$ is replaced with $\triangledown_{\mathrm{num},\Psi}$.

5.4.4 Soundness of the abstract semantics

As mentioned above in Section 5.4.2, the abstract semantics of $\mathbb{S}^\sharp$ is sound, not with regard to the typed semantics $[\![\cdot]\!]_t$, but with regard to the untyped one $[\![\cdot]\!]$.

Because $\mathbb{S}^\sharp$ concretizes to $\mathcal{P}\big(\mathbb{S}_t \times (\mathbb{V}^\sharp \to \mathbb{V})\big)$, to define soundness with regard to the untyped semantics we must first define an operator that strips away the types and valuations:

Definition 5.17 (Type erasure operator). The function untyp : $\mathcal{P}\big(\mathbb{S}_t \times (\mathbb{V}^\sharp \to \mathbb{V})\big) \to \mathcal{P}(\mathbb{S})$ is defined as:

$$\mathrm{untyp}(\mathrm{X}) = \{(\sigma, h) \mid ((\sigma, \Gamma, h, \mathcal{L}), v) \in \mathrm{X}\}$$

Theorem 5.7 (Soundness of the abstract semantics of $\mathbb{S}^\#$). *Let $\mathbb{s} \in \mathbb{S}^\#$ and $p \in$ stmt. Suppose that for all $(s, v) \in \gamma_S(\mathbb{s})$, s is well typed under v. Then:*

$$(\llbracket p \rrbracket \circ \text{untyp} \circ \gamma_S)(\mathbb{s}) \subseteq (\text{untyp} \circ \gamma_S \circ \llbracket p \rrbracket_S^\#)(\mathbb{s}).$$

In addition, the abstract semantics preserve the well-typedness of states:

$$\forall (s', v') \in (\gamma_S \circ \llbracket p \rrbracket_S^\#)(\mathbb{s}), s' \textit{ is well typed under } v'.$$

Proof. The conjunction of the two propositions can be proven by induction on the syntax of statements: following the same sketch as Theorem 4.8 and using Theorem 5.5, it can be proven that the abstract semantics preserves typing; in addition, the soundness with regard to $\llbracket \cdot \rrbracket$ can be proven from the definitions and the typing invariants. □

5.4.5 Approximation of aliasing relations

Finally, we now detail how the types inferred by the analysis can be used to infer aliasing relations between memory regions, as it is used by the abstractions that we define in Chapter 6.

We introduce a function $\textbf{may\_alias}_S : \mathbb{S}^\# \times \mathbb{V}_S \times \mathbb{N} \times \mathbb{V}_S \times \mathbb{N} \to \mathbb{B}$ that soundly approximates aliasing relations between two memory regions, in the sense that if $\textbf{may\_alias}_S(\mathbb{s}, (\mathbb{v}_1, \mathbb{t}_1), \ell_1, (\mathbb{v}_2, \mathbb{t}_2), \ell_2)$ is *false*, then it is guaranteed that the region of size ℓ_1 starting at address $(\mathbb{v}_1, \mathbb{t}_1)$ does *not* overlap with the region of size ℓ_2 starting at address $(\mathbb{v}_2, \mathbb{t}_2)$.

This function will be used by the domains we define in Chapter 6 to check the safety of some static analysis operations.

First, we define an auxiliary predicate $\textbf{in\_bounds}_{v^\#}$ to check whether an abstract pointer offset is still in the bounds of its pointed type after adding a constant length to it.

Definition 5.18 ($\textbf{in\_bounds}_{v^\#}$ predicate). Given $v^\# \in \mathbb{D}_{\text{num}}$, the predicate $\textbf{in\_bounds}_{v^\#} : \mathbb{T}^\# \times \mathbb{N}$ takes a (possibly abstract) pointer type and an integer, and is defined by:

$$\begin{aligned}
\textbf{in\_bounds}_{v^\#}(t[\mathbb{s}].(\mathbb{i})*, \ell) &\iff v^\# \vDash 0 \leq \mathbb{i} + \ell < \mathbb{s} \cdot \text{size}(t) \\
\textbf{in\_bounds}_{v^\#}(t.(\mathbb{i})*, \ell) &\iff v^\# \vDash 0 \leq \mathbb{i} + \ell < \text{size}(t) \\
\textbf{in\_bounds}_{v^\#}(t.(i)*, \ell) &\iff v^\# \vDash 0 \leq i + \ell < \text{size}(t)
\end{aligned}$$

It is undefined on non-pointer types.

Definition 5.19 (May-alias relations between regions in $\mathbb{S}^\#$). The result of $\textbf{may\_alias}_S(\mathbb{s}, (\mathbb{v}_1, \mathbb{t}_1), \ell_1, (\mathbb{v}_2, \mathbb{t}_2), \ell_2)$ is determined as follows. Let $((\sigma^\#, \Gamma^\#), v^\#) = \mathbb{s}$.

- If both $\mathbb{v}_1$ and $\mathbb{v}_2$ *may* be equal to zero, that is, if neither $v^\# \vDash \mathbb{v}_1 \neq 0$ nor $v^\# \vDash \mathbb{v}_2 \neq 0$ hold, then the result is true.
- If one of $\mathbb{t}_1$ and $\mathbb{t}_2$ is a non-pointer type, then the result is true.
- Otherwise, let $u_1.(\mathbb{i}_1)* = \mathbb{t}_1$ and $u_2.(\mathbb{i}_2)* = \mathbb{t}_2$, where $\mathbb{i}_1$ and $\mathbb{i}_2$ are in $\mathbb{V}^\# \uplus \mathbb{N}$. If either $\textbf{in\_bounds}_{v^\#}(u_1.(\mathbb{i}_1)*, \ell_1)$ or $\textbf{in\_bounds}_{v^\#}(u_2.(\mathbb{i}_2)*, \ell_2)$ does not hold, or if u_1 contains u_2 or u_2 contains u_1, then the result is true. Otherwise, the result is false.

This function is sound in the following sense:

Theorem 5.8 (Soundness of **may_alias**$_\mathrm{S}$). *Given $\mathbb{s} \in \mathbb{S}^\#$, $(\mathbb{v}_1, \mathbb{t}_1)$ and $(\mathbb{v}_2, \mathbb{t}_2)$ in $\mathbb{V}_\mathrm{S}$, given $\ell_1, \ell_2 \in \mathbb{N}$, if* **may_alias**$_\mathrm{S}(\mathbb{s}, (\mathbb{v}_1, \mathbb{t}_1), \ell_1, (\mathbb{v}_2, \mathbb{t}_2), \ell_2) = \text{false}$*, then for all $((\sigma, \Gamma, h, \mathcal{L}),\ \mathcal{v}) \in \gamma_\mathrm{S}(\mathbb{s})$, for all $t_1, t_2 \in \mathbb{T}$ such that $(t_1, \mathcal{v}) \in \gamma_\mathrm{T}(\mathbb{t}_1)$ and $(t_2, \mathcal{v}) \in \gamma_\mathrm{T}(\mathbb{t}_2)$:*

$$\mathcal{v}(\mathbb{v}_1) \in \llparenthesis t_1 \rrparenthesis_{\mathcal{L},\mathcal{v}} \wedge \mathcal{v}(\mathbb{v}_2) \in \llparenthesis t_2 \rrparenthesis_{\mathcal{L},\mathcal{v}} \implies [\mathcal{v}(\mathbb{v}_1),\ \mathcal{v}(\mathbb{v}_1) + \ell_1[\cap [\mathcal{v}(\mathbb{v}_2),\ \mathcal{v}(\mathbb{v}_2) + \ell_2[= \emptyset$$

Proof. Suppose that **may_alias**$_\mathrm{S}(\mathbb{s}, (\mathbb{v}_1, \mathbb{t}_1), \ell_1, (\mathbb{v}_2, \mathbb{t}_2), \ell_2)$ returns false. Then $t_1 = u_1.(i_1)*$ and $t_2 = u_2.(i_2)*$ for some $u_1, u_2 \in \mathbb{T}$ such that u_1 does not contain u_2 and u_2 does not contain u_1 (follows from the definition of “contains”). In addition, by Theorem 4.4:

$$\mathrm{addr}_\mathcal{L}(u_1) \cap \mathrm{addr}_\mathcal{L}(u_2) = \emptyset$$

But by definition of interpretation (Definition 4.8) and contiguity of the labelling (Definition 4.2) and because $\mathcal{v}(\mathbb{v}_1) \neq 0$ and $\mathcal{v}(\mathbb{v}_2) \neq 0$, $[\mathcal{v}(\mathbb{v}_1),\ \mathcal{v}(\mathbb{v}_1) + \ell_1[\subseteq \mathrm{addr}_\mathcal{L}(u_1)$ and $[\mathcal{v}(\mathbb{v}_2),\ \mathcal{v}(\mathbb{v}_2) + \ell_2[\subseteq \mathrm{addr}_\mathcal{L}(u_2)$, and the result follows. □

5.5 Analysis example

Let us take as an example the call of function `dll_union` at line 38 of Figure 4.1.

The details of translating from C to WHILE-MEMORY are given in Chapter 7, but in this simple case we inline the function call and represent the arguments `x` and `y` of `dll_union` by two WHILE-MEMORY variables x' and y' (to distinguish them from the arguments of `merge`). The result is the following WHILE-MEMORY program:

$$\begin{aligned}
&x' := x + 4;\\
&y' := y + 4;\\
&*_4(*_4 y' + 4) := *_4(x' + 4);\\
&*_4(*_4(x' + 4)) := *_4 y';\\
&*_4(x' + 4) := y';\\
&*_4 y' := x'
\end{aligned}$$

We take as initial state the abstract state of Example 5.4:

$$\begin{array}{ll}
x\ \boxed{\mathbb{v}_0\ :\ \texttt{node}.(0)*} & \mathbb{v}_0 \neq 0\\
y\ \boxed{\mathbb{v}_1\ :\ \texttt{node}.(0)*} & \mathbb{v}_1 \neq 0
\end{array}$$

By rule ADDR$^\#$, variables x' and y' are both attributed type `node`.(4)$*$ and a non-null abstract value. Let us now analyze the third line: first, $*_4 y'$ evaluates to the type $\{x\ :\ \texttt{dll}.(0)* \mid x \neq 0\}$ and to a non-null value —indeed, the value is refined by the predicate “$x \neq 0$”— by rule LOAD$^\#$ and application of “deref”. Thus $*_4 y' + 4$ evaluates to $(\mathbb{a}, \mathbb{t}_a)$ with $\mathbb{a}$ non-null and $\mathbb{t}_a = \texttt{dll}.(4)*$.

By similar reasoning, the right-hand side $*_4(x' + 4)$ evaluates to $(\mathbb{v}, \mathbb{t}_v)$, with $\mathbb{v}$ non-null and $\mathbb{t}_v = \{x\ :\ \texttt{dll}.(0)* \mid x \neq 0\}$. As a consequence, rule STORE$^\#$ applies, since $\mathrm{deref}_{\mathcal{v}^\#}(\mathbb{t}_a, 4) = \mathbb{t}_v$ (where $\mathcal{v}^\#$ represents the constraints accumulated at this point of the analysis) and the abstract type checking

$$\mathbb{v}\ :\ \mathbb{t}_v \overset{\mathcal{v}^\#}{\hookrightarrow} \mathbb{t}_v$$

is verified. Thus the third line leaves the abstract state unchanged.

The abstract semantics can thus be computed for each statement; in the end, no alarm is emitted by the analysis, and the abstract values of x' and y' remain unchanged, yielding the final state:

$$\begin{array}{rl} x & \boxed{\mathbb{v}_0 \ : \ \mathtt{node}.(0)*} \\ y & \boxed{\mathbb{v}_1 \ : \ \mathtt{node}.(0)*} \\ x' & \boxed{\mathbb{v}_2 \ : \ \mathtt{node}.(4)*} \\ y' & \boxed{\mathbb{v}_3 \ : \ \mathtt{node}.(4)*} \end{array} \qquad \begin{array}{l} \mathbb{v}_0 \neq 0 \\ \mathbb{v}_1 \neq 0 \\ \mathbb{v}_2 \neq 0 \\ \mathbb{v}_3 \neq 0 \end{array}$$

Which proves the spatial memory safety of this code fragment, and gives information about the pointed data structures after its execution.

5.6 Conclusion and related work

We used physical types and the notion of well-typed state to develop an abstraction of memory based on the respect of typing invariants, as well as a static analysis to verify that a program preserves these invariants. Let us now compare our abstraction to existing approaches.

Our approach is more precise than the pointer analyses surveyed in Chapter 2 since those cannot compute properties on heap objects that are not directly pointed to by program variables, whereas we do (through typing constraints). The type-based domain $\mathbb{S}^\sharp$ can be seen as a refinement of Ghiya and Hendren [GH96], except that rather than attributing to variables a “shape” among a limit set of possible shapes, we attribute them a type among a rich set of types which finely describe data structures and relations between them (the set of types being a parameter of the analysis), and represent numerical predicates on values using a numerical abstract domain.

However, our analysis cannot verify properties as strong as those verified by full-fledged shape analyses, mainly because it collapses all objects of the same type together. On the other hand, it can natively handle data with unstructured sharing, which is challenging for shape analyses [LRC15].

In addition, the analysis presented so far does not perform the materialization and summarization operations that characterize shape analysis [Cha+20]; however, the next chapter introduces a form of refinement operation *as an independent abstraction.*

The only existing shape analysis that targets programs as low-level as we do, featuring, e.g., unrestricted pointer arithmetic or pointers to non-zero offsets into structures, is Predator [DPV13]. Predator computes shape invariants expressed as tree automata with edges labelled by numerical offsets. Its scope is currently limited to various forms of singly- and doubly-linked lists (possibly nested or cyclic, with data pointers and some sentinel pointers).

The type-based analysis by Diwan et al. [DMM98] is limited to type-safe C programs, i.e., without type casts or arbitrary pointer arithmetic. However, it already observed and took advantage of the fact that objects whose types are not in a subtyping relation do not alias, as we do.

Our type system is inspired by Chandra and Reps's [CR99] and extends it. The analysis in [CR99] uses a subtyping relation similar to $\preceq$, but it is limited as it is only a type inference algorithm, without numerical analysis. As a consequence, it cannot handle pointer arithmetic like we do. Rondon et al.'s analysis [RKJ10] does not need to assume type safety, but rather verifies it, in a type system stronger than that of C. However, it does not handle programs that take advantage of structural subtyping, i.e., programs that use subtyping relations of the form $t.(i)* \preceq u.(j)*$. It is also conservative in the way it chooses when to materialize or summarize heap objects; this is due to the fact that it is mostly a type inference algorithm, in contrast to flow-sensitive abstract interpretation analyses like ours which can make decisions based on the possible values in variables and in memory. In addition, liquid types do not support the structural subtyping of C, whereas it is readily supported by our physical types.

The fact that our abstraction does not represent the heap explicitly is reminiscent of storeless semantics [Jon81; Deu92] in which data is represented by the set of possible access paths, and an equivalence relation on these access paths denoting aliasing. The type-based abstraction can be seen as such a storeless semantics, but expressing the byte-level representation of data; in this view, access paths would be sequences of pointer arithmetics and pointer dereferences. Aliasing relations can be derived from the address subtyping relation, as explained in the previous section. However, we did not pursue the direction of representing memory as pairs of aliased access paths. Rather, we abstract it using the typing constraints. We can analyze programs manipulating unbounded structures, although we cannot express access paths as precise as the symbolic access paths of Deutsch [Deu94]. On the other hand, our abstraction is more lightweight, and our analysis handles non-type-safe programs.

The safety verifications in Cyclone [Mor+02], CCured [Nec+05], or CheckedC [Ell+18] are mostly based on type inference, and thus inherit the limitations of this family of algorithms, already mentioned above [DMM98; CR99; RKJ10], namely the absence of value information that can be used to guide the type abstraction.

Chapter 6

Retained and staged points-to predicates

Outline of the current chapter

The type-based shape abstraction suffers from two important limitations. First, the heap is represented only in a summarized form by the type constraints and there is no way to retain additional information about its contents. Second, all writes to memory must preserve the type invariants—situations where the type invariants are temporarily violated are not handled. We solve both problems by tracking some *points-to predicates* and attaching specific properties to them. We give an informal overview of these points-to predicates in Section 6.1, before describing them in detail in the form of two abstract domains, namely the domain of retained (Section 6.2) and staged (Section 6.3) points-to predicates.

6.1 Informal overview

Consider again the example of Figure 4.1, repeated on page 72. In Chapter 5, we have shown how the `dll_union` function can be proved to preserve typing invariants using our type-based shape domain.

```c
typedef struct uf {
  struct uf* parent;
} uf;

typedef struct dll {
  struct dll *prev; /* != null. */
  struct dll *next; /* != null. */
} dll;

typedef unsigned int node_kind;
typedef struct node {
  node_kind kind; /* kind <= 5. */
  dll dll;
  uf uf;
} node;

uf *uf_find(uf *x) {
  while(x->parent != 0) {
    uf *parent = x->parent;
    if(parent->parent == 0)
      return parent;
    x->parent = parent->parent;
    x = parent->parent;
  }
  return x;
}
```

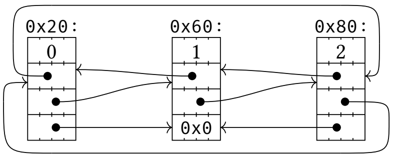
```c
void dll_union(dll *x, dll *y) {
  y->prev->next = x->next;
  x->next->prev = y->prev;
  x->next = y; y->prev = x;
}

void uf_union(uf *x, uf *y) {
  uf *rootx = uf_find(x);
  uf *rooty = uf_find(y);
  if(rootx != rooty)
    rootx->parent = rooty;
}

void merge(node *x, node *y) {
  dll_union(&x->dll, &y->dll);
  uf_union(&x->uf, &y->uf);
}

node *make(node_kind kind) {
  node *n = malloc(sizeof(node));
  n->kind = kind;
  n->dll.next = &n->dll;
  n->dll.prev = &n->dll;
  n->uf.parent = NULL;
  return n;
}
```

Figure 4.1: An algorithm for union-find and listing elements in a partition. (repeated from page 34)



(a) Concrete state.

$$
\begin{aligned}
\mathcal{M}\ :\ \texttt{node\_kind} &\mapsto \{x : \texttt{word}_4 \mid x \leq 5\}\\
\texttt{uf} &\mapsto \texttt{uf}.(0)*\\
\texttt{dll} &\mapsto \{x : \texttt{dll}.(0)* \mid x \neq 0\} \times \{x : \texttt{dll}.(0)* \mid x \neq 0\}\\
\texttt{node} &\mapsto \texttt{node\_kind} \times \texttt{dll} \times \texttt{uf}
\end{aligned}
$$

(b) Physical types.

$$
\begin{array}{rcll}
\multicolumn{4}{l}{\forall \upsilon, \forall(\sigma, h, \mathcal{L}, \Gamma) \text{ well-typed state}, \forall v \text{ value } :}\\
v \in (\!|\texttt{node}.(0)*|\!)_{\mathcal{L},\upsilon} \wedge v \neq 0 & \Longrightarrow & v + 4 \in (\!|\texttt{node}.(4)|\!)_{\mathcal{L},\upsilon} \wedge v + 4 \neq 0 & (1)\\
(\!|\texttt{node}.(4)*|\!)_{\mathcal{L},\upsilon} & \subseteq & (\!|\texttt{dll}.(0)*|\!)_{\mathcal{L},\upsilon} & (2)\\
v \in (\!|\texttt{dll}.(0)*|\!)_{\mathcal{L},\upsilon} \wedge v \neq 0 & \Longrightarrow & h[v..v+4] \in (\!|\{x : \texttt{dll}.(0)* \mid x \neq 0\}|\!)_{\mathcal{L},\upsilon} & (3)\\
(\!|\texttt{uf}.(0)*|\!)_{\mathcal{L},\upsilon} & \cap & (\!|\texttt{dll}.(0)*|\!)_{\mathcal{L},\upsilon} = \{0\} & (4)
\end{array}
$$

(c) Some structural invariants entailed by $\mathcal{M}$

Figure 4.2: Concrete state, physical types and example structural invariants. (repeated from page 35)

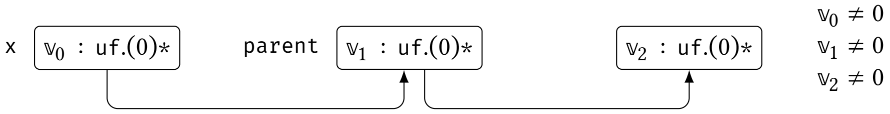


Figure 6.1: Abstract state just before line 22.

However, this approach does not suffice when considering more complex functions, like `uf_find`. First, we remark that this function may run correctly only when the `x` argument is non-null due to the dereference at line 18. Therefore, the verification of this function will require some form of annotation specifying this precondition.

Next, we observe that in order to prove the validity of the access to `parent->parent` at line 20, the analysis needs to establish that `parent` is equal to `x->parent`, which is non-null due to the condition at line 18. Such reasoning cannot be performed solely using a combination of types and numerical predicates, because the type-based invariants cannot attach different information to different heap objects of the same type.

Therefore, we add the possibility to represent additional *points-to predicates.* Such predicates express a points-to relation between two values and can complement pointer types with more precise information about the value pointed. Informally, this corresponds to adding additional boxes to our graphical representation of the abstract state, which correspond to selected heap addresses. Only boxes that are reachable from a variable finite chain of points-to predicates may be retained this way. Figure 6.1 shows the abstract state at line 22 that enables to proves the correctness of the `parent->parent` access. In the following, we call such predicates *retained points-to predicates.*

Such predicates are obtained by retaining information about recent memory writes, reads, or condition tests and need to be dropped as soon as they cannot be proved to be preserved. Indeed, when the analysis encounters a memory write, it drops all such boxes for which the absence of aliasing cannot be established with the current information; some aliasing information (e.g. Equation (4) in Figure 4.2) comes from the partitioning of the heap (see Section 5.4.5). This process will be referred to as *blurring* as it carries some similarity with the blurring encountered in some shape analyses.

Note that retained points-to predicates offer a very lightweight way to keep some memory cells represented precisely, without resorting to *unfolding* or *focusing* as in usual shape abstract domains, which is generally more costly (but also more powerful from the logical point of view), as retaining a heap address or blurring it does not require modifying the summarized heap representation. Physical types coupled with retained points-to predicates allow to verify memory safety and typing invariant preservation for the four functions `dll_union`, `uf_find`, `uf_union` and `merge`.

Finally, we consider the `make` function. For the sake of simplicity, we assume that `malloc` always returns a non-null pointer. We note that variable `n` does not point to a valid node object until the very end of the function, thus attempting to prove it satisfies physical type node.(0) before that point—for instance, just after the call to `malloc`—will fail. In general, some code patterns like memory allocation or byte-per-byte copy temporarily do not preserve the structural invariants described by our types. To alleviate this, we augment our abstraction with a notion of *staged points-to predicates* that represent precisely the effect of sequences of store instructions such as the body of `make`, allowing to delay their abstraction into types at a later point. We give a step-by-step account of the analysis of `make` in Section 6.3.2.

These two extensions, retained points-to predicates and staged points-to predicates, take the form of two independent abstract domains.

6.2 Retained points-to predicates

6.2.1 Abstraction

Points-to predicates are an abstraction that extends another abstract domain $\mathbb{D}^\sharp$:

Notation 6.1 (Parameter domain for retained points-to predicates). *In the rest of this chapter, we denote as $\mathbb{D}^\sharp$ an abstract domain with concretization $\gamma_{\mathrm{D}} : \mathbb{D}^\sharp \to \mathcal{P}(\mathbb{S}_t \times (\mathbb{V}^\sharp \to \mathbb{V}))$. We denote as $\mathbb{V}_{\mathrm{D}}$ the set of abstract values of $\mathbb{D}^\sharp$ (see Section 3.4.1); we assume that $\mathbb{D}^\sharp$ provides a sound abstract semantics of expressions $\mathcal{E}[\![\cdot]\!]^\sharp_{\mathrm{D}}$ and a sound abstract semantics of statements $[\![\cdot]\!]^\sharp_{\mathrm{D}}$, and that $[\![\cdot]\!]^\sharp_{\mathrm{D}}$ preserves the well-typedness of states.*

Example 6.1. $\mathbb{D}^\sharp$ can be instantiated to $\mathbb{S}^\sharp$ (and it is the case in our experiments, see Section 6.4) and $\mathbb{V}_{\mathrm{D}}$ to $\mathbb{V}_{\mathrm{S}} = \mathbb{V}^\sharp \times \mathbb{T}^\sharp$.

We also assume that $\mathbb{D}^\sharp$ possesses an underlying numerical domain, like $\mathbb{S}^\sharp$ does, for instance. In order to simplify notation, we extend our notations $v^\sharp \models p$ (numerical predicate p is verified by the abstract semantics of $v^\sharp$, notation defined in Definition 3.8), to elements $\mathbb{s} \in \mathbb{D}^\sharp$, i.e. $\mathbb{s} \models p$, where p is a formula whose free variables are in $\mathbb{V}_{\mathrm{D}}$. Similarly, we shall let $\mathbb{s}[p]$ denote the addition of a numerical constraint p to the abstract state $\mathbb{s}$.

Intuitively, a points-to predicate constrains an abstract value to point to some abstract region; and an abstract region is a partition of a memory region into contiguous intervals, such that each interval is mapped to an abstract value. Formally:

Definition 6.1 (Abstract region). An *abstract region* $\mathcal{R} : \mathbb{I} \rightharpoonup \mathbb{V}_{\mathrm{D}}$ is a partial mapping from positive integer intervals to abstract values, such that $\bigcup_{i \in \mathrm{dom}(\mathcal{R})} = [0, n[$ for some n, called the *size* of $\mathcal{R}$ and denoted as $\mathrm{size}(\mathcal{R})$; and any two intervals in $\mathrm{dom}(\mathcal{R})$ are disjoint. The set of abstract regions is denoted as $\mathrm{Reg}(\mathbb{D}^\sharp)$.

An abstract region is the abstraction of a set of values, by concatenation of the values it contains. This is expressed by the concretization function:

$$\gamma_{\mathrm{reg}} : \mathrm{Reg}(\mathbb{D}^\sharp) \to \mathcal{P}\big(\mathbb{V} \times (\mathbb{V}^\sharp \to \mathbb{V})\big)$$

$$\gamma_{\mathrm{reg}}(\mathcal{R}) = \left\{ (v, \nu) \;\middle|\; \begin{array}{l} v = \gamma^{\mathrm{V}}_{\mathrm{D}}(\nu, \mathcal{R}(i_1)) :: \cdots :: \gamma^{\mathrm{V}}_{\mathrm{D}}(\nu, \mathcal{R}(i_n)) \\ \text{where } i_1, \ldots, i_n \text{ are the elements of } \mathrm{dom}(\mathcal{R}) \text{ in order.} \end{array} \right\}$$

Given $\mathcal{R} \in \mathrm{Reg}(\mathbb{D}^\sharp)$, $\mathcal{R}([n_1, n_2[)$ is thus the abstract value covering bytes n_1 to $n_2 - 1$ of the region. In a slight abuse of notation, we shall consider $\mathcal{R}([n_1, n_2[)$ to be a valid abstract value even when $[n_1, n_2[$ is not an element of $\mathrm{dom}(\mathcal{R})$, but is nonetheless included in $[0, \mathrm{size}(\mathcal{R})[$. In full rigor, this implies requests to the underlying numerical domain to reconstruct the value from the abstract values in the intervals of $\mathrm{dom}(\mathcal{R})$ covered by $[n_1, n_2[$, but in order to not further complexify this formalization, we consider it an implementation detail. Similarly, we shall write $\mathcal{R}\big[\,[n_1, n_2[\leftarrow \mathbb{v}\,\big]$ to denote the replacement of bytes $[n_1, n_2[$ of $\mathcal{R}$ with $\mathbb{v} \in \mathbb{V}_{\mathrm{D}}$.

Informally, the meaning of a points-to predicate $\mathbb{a} \mapsto \mathcal{R}$ is that the value of size $\mathrm{size}(\mathcal{R})$ stored in the heap at address $\mathbb{a}$ is one of the values abstracted by $\mathcal{R}$. The next two definitions formally define what points-to predicates express relative to the rest of the abstract state.

Points-to predicates are represented by a partial function p mapping abstract values (representing addresses) to abstract regions. We will call such a mapping from abstract values to abstract regions an *abstract points-to map*, and the concrete meaning of such a map is obtained by considering all possible concrete values for each abstract region.

Definition 6.2 (Abstract points-to map). An *abstract points-to map* for domain $\mathbb{D}^\#$, whose set is denoted $\mathbb{P}^\#(\mathbb{D}^\#)$, is a partial function from abstract values to abstract regions:

$$\mathbb{P}^\#(\mathbb{D}^\#) = \mathbb{V}_\mathrm{D} \rightharpoonup \mathrm{Reg}(\mathbb{D}^\#)$$

An abstract points-map is the abstraction of a set of heaps, *via* the concretization:

$$\begin{aligned}
&\gamma_\mathrm{P} \ : \ \mathbb{P}^\#(\mathbb{D}^\#) \to \mathcal{P}\big(\mathbb{H} \times (\mathbb{V}^\# \to \mathbb{V})\big)\\
&\gamma_\mathrm{P}(\mathbb{p}) = \{(h, \upsilon) \mid \forall \mathbb{a} \in \mathbb{V}_\mathrm{D}, \forall v \in \mathbb{V},\ (v, \upsilon) \in \gamma_\mathrm{reg}(\mathbb{p}(\mathbb{a})) \implies h[\gamma_\mathrm{D}^\mathrm{V}(\upsilon, \mathbb{a})..\gamma_\mathrm{D}^\mathrm{V}(\upsilon, \mathbb{a}) + \mathrm{size}(\mathrm{size}_(\ \mathbb{p}(\mathbb{a})))] = v\}
\end{aligned}$$

The retained points-to predicate abstraction consists in associating an abstract points-to map $\mathbb{p}$ to an abstract state of $\mathbb{D}^\#$. The concretization of these predicates is done by a form of intersection semantics between the two domains.

Definition 6.3 (Retained points-to predicate domain). The retained points-to predicate abstract domain for domain $\mathbb{D}^\#$, denoted as $\mathbb{R}^\#(\mathbb{D}^\#)$, is the product set of $\mathbb{D}^\#$ with abstract points-to maps:

$$\mathbb{R}^\#(\mathbb{D}^\#) = \mathbb{D}^\# \times \mathbb{P}^\#(\mathbb{D}^\#)$$

Its concretization is:

$$\begin{aligned}
&\gamma_\mathrm{R} \ : \ \mathbb{R}^\#(\mathbb{D}^\#) \to \mathcal{P}\big(\mathbb{S}_t \times (\mathbb{V}^\# \to \mathbb{V})\big)\\
&\gamma_\mathrm{R}(\mathbb{s}, \mathbb{p}) = \{((\sigma, \Gamma, h, \mathcal{L}),\ \upsilon) \in \gamma_\mathrm{D}(\mathbb{s}) \mid (h, \upsilon) \in \gamma_\mathrm{P}(\mathbb{p})\}
\end{aligned}$$

Its abstract values range over $\mathbb{V}_\mathrm{R} = \mathbb{V}_\mathrm{D}$.

Interestingly, this abstraction is defined independently from any notion of type. In our analyses, we will instantiate $\mathbb{D}^\#$ to $\mathbb{S}^\#$, but the retained points-to predicate domain may be a useful extension to other abstract domains, e.g., one performing an Andersen-style alias analysis (see Section 2.1).

The type-based shape domain $\mathbb{S}^\#$ remembers flow-sensitive information only about the variable store, as the heap is represented only using the type invariants. We use retained points-to predicates $\mathbb{v} \mapsto \mathcal{R}$ to store flow-sensitive information about the heap: they make it possible to attach numerical and type information to values stored in the heap. Retained points-to predicates do not quite achieve the effect of materialization in shape analyses, because they merely retain information about recently accessed memory zones, rather than refining a summarized region into a more precise representation.

Example 6.2. Figure 6.1 graphically represents the abstract state just before the execution of line 22. Here the parameter domain $\mathbb{D}^\#$ is instantiated to $\mathbb{S}^\#$. The abstract state is a pair $(\mathbb{s}, \mathbb{p}) \in \mathbb{R}^\#(\mathbb{S}^\#)$, where $\mathbb{s} \in \mathbb{S}^\#$ is:

$$\mathbb{s} = \big([\mathtt{x} \mapsto \mathbb{v}_0,\ \mathtt{parent} \mapsto \mathbb{v}_1],\ [\mathtt{x} \mapsto \mathtt{uf.(0)*},\ \mathtt{parent} \mapsto \mathtt{uf.(0)*}],\ \upsilon^\#\big)$$

with $\upsilon^\# \in \mathbb{D}_\mathrm{num}$, and

$$\mathbb{p} = [(\mathbb{v}_0, \mathtt{uf.(0)*}) \mapsto \mathcal{R}_1,\ (\mathbb{v}_1, \mathtt{uf.(0)*}) \mapsto \mathcal{R}_2]$$

where $\mathcal{R}_1$ and $\mathcal{R}_2$ are the two abstract regions defined by:

$$\begin{aligned}
\mathcal{R}_1 &= \big[\,[0, 4[\ \mapsto (\mathbb{v}_1, \mathtt{uf.(0)*})\,\big]\\
\mathcal{R}_2 &= \big[\,[0, 4[\ \mapsto (\mathbb{v}_2, \mathtt{uf.(0)*})\,\big].
\end{aligned}$$

6.2.2 Abstract semantics of expressions

The abstract semantics of expressions $\mathcal{E}[\![e]\!]^{\#}_{\mathrm{R}} : \mathbb{R}^{\#}(\mathbb{D}^{\#}) \to \mathbb{V}_{\mathrm{R}} \times \mathbb{R}^{\#}(\mathbb{D}^{\#})$ computes, from an expression and an abstract state, a pair constituted of an abstract value and a new abstract state.

For most expressions, the abstract semantics of $\mathbb{R}^{\#}(\mathbb{D}^{\#})$ does not need to use the abstract points-to map and is the same as the abstract semantics of $\mathbb{D}^{\#}$. The only exception is memory reads. The result of a read of size ℓ at address $\mathbb{a}$ is determined as follows:

- if the read region can be determined to be included in one of the regions retained in the abstract points-to map, then the read is performed from that region.
- Otherwise, the abstract semantics of $\mathbb{D}^{\#}$ is used to perform the read, and the resulting value is added to the abstract points-to map.

There is one additional parameter that must be determined when adding a new region to the abstract points-to map: the choice of the *limits* (i.e. the base and size) of the region to retain. One possible choice is to retain the region read, resulting in the retained points-to predicate $\mathbb{a} \mapsto \mathcal{R}$, with $\mathrm{size}(\mathcal{R}) = \ell$.

In the context of type-based analysis (i.e. when the parameter domain of $\mathbb{R}^{\#}$ is $\mathbb{S}^{\#}$), we make a different choice: when a memory access is made into a region of type t, we retain the region corresponding to *the entire type*. In C, this can correspond to retaining an entire structure when an access is made to a single field of that structure, for instance. This choice of a more relevant region to retain can enhance precision e.g. when manipulating a structure whose fields are related by some constraint (see example in Section 4.4.2). However, in some cases, type information may fail to help identify a relevant region to retain, e.g. if the memory read dereferences an expression whose type is not a pointer.

In order to abstract these considerations from the type-based domain $\mathbb{S}^{\#}$, we require the existence of a function $\mathrm{should\_retain}_{\mathrm{D}} : \mathbb{D}^{\#} \times \mathbb{V}_{\mathrm{D}} \to (\{\mathrm{true}\} \times \mathbb{N} \times \mathbb{N}) \cup \{\mathrm{false}\}$, which takes an abstract value $\mathbb{a}$ and returns whether memory accesses to $\mathbb{a}$ should be retained as points-to predicates; and, if so, what should be the size of the resulting region and the offset of $\mathbb{a}$ in it. For instance, for the type-based domain $\mathbb{S}^{\#}$, $\mathrm{should\_retain}_{\mathrm{S}}$ is defined as follows:

Definition 6.4 (Retaining decision function in $\mathbb{S}^{\#}$). The result of $\mathrm{should\_retain}_{\mathrm{S}}(\mathbb{s}, (\mathbb{a}, \mathbb{t}_a))$ is defined as follows:

- if $\mathbb{t}_a$ is not a pointer type, then $\mathrm{should\_retain}_{\mathrm{S}}(\mathbb{s}, (\mathbb{a}, \mathbb{t}_a)) = \mathrm{false}$.
- Else, if $\mathbb{t}_a = t.(\mathbb{i})\star$ and the offset can be precisely determined, i.e. there exists $i \in \mathbb{N}$ such that $\mathbb{s} \vDash \mathbb{i} = i$, then $\mathrm{should\_retain}_{\mathrm{S}}(\mathbb{s}, (\mathbb{a}, \mathbb{t}_a)) = (\mathrm{true}, i, \mathrm{size}(t))$.
- Else, if $\mathbb{t}_a = t[\mathbb{n}].(\mathbb{i})\star$ and the offset can be precisely determined, i.e. there exists $i, n \in \mathbb{N}$ such that $\mathbb{s} \vDash \mathbb{i} = i \wedge \mathbb{n} = n$, then $\mathrm{should\_retain}_{\mathrm{S}}(\mathbb{s}, (\mathbb{a}, \mathbb{t}_a)) = (\mathrm{true},\ i,\ n \cdot \mathrm{size}(t))$.
- Otherwise, the offset cannot be determined precisely: $\mathrm{should\_retain}_{\mathrm{S}}(\mathbb{s}, (\mathbb{a}, \mathbb{t}_a)) = \mathrm{false}$.

This enables us to define the function "retain", which retains a value in a newly created region of the abstract points-to map, if possible, i.e. if $\mathrm{should\_retain}_{\mathrm{D}}$ returns true.

Definition 6.5 (retain function). The function $\mathrm{retain} : \mathbb{D}^{\#} \times \mathbb{P}^{\#}(\mathbb{D}^{\#}) \times \mathbb{V}_{\mathrm{D}} \times \mathbb{N} \times \mathbb{V}_{\mathrm{D}} \to \mathbb{D}^{\#} \times \mathbb{P}^{\#}(\mathbb{D}^{\#})$ is defined by:

- if $\mathrm{should\_retain}_{\mathrm{D}}(\mathbb{s}, \mathbb{a}) = \mathrm{false}$, then retain does nothing: $\mathrm{retain}(\mathbb{s}, \mathbb{p}, \mathbb{a}, \ell, \mathbb{v}) = (\mathbb{s}, \mathbb{p})$.
- Otherwise, if $\mathrm{should\_retain}_{\mathrm{S}}(\mathbb{s}, \mathbb{a}) = (\mathrm{true}, i, n)$, then let $\mathcal{R}$ be a region of size n containing a fresh, unconstrained value. Let $\mathcal{R}' = \mathcal{R}\big[\,[i, i+\ell[\ \leftarrow \mathbb{v}\,\big]$. Let $\mathbb{s}_1 = \mathbb{s}[\mathbb{a}_0 = \mathbb{a} - i]$, where $\mathbb{a}_0$ is a fresh variable. Then:
$$\mathrm{retain}(\mathbb{s}, \mathbb{p}, \mathbb{a}, \ell, \mathbb{v}) = \big(\mathbb{s}_1,\ \mathbb{p}[\mathbb{a}_0 \leftarrow \mathcal{R}']\big)$$

Figure 6.2 uses this function to define the semantics of memory reads.

$$\frac{\begin{array}{c}\mathcal{E}[\![e]\!]^\sharp_{\mathrm{R}}(\mathbb{s},\mathbb{p}) = \big(\mathbb{a},\ (\mathbb{s}_1,\mathbb{p}_1)\big) \\ \exists \mathbb{a}_0 \in \mathrm{dom}(\mathbb{p}_1),\ \ \mathbb{s}_1 \vDash [\mathbb{a},\mathbb{a}+\ell[\ \subseteq [\mathbb{a}_0,\mathbb{a}_0+\mathrm{size}(\mathbb{p}_1(\mathbb{a}_0))[\qquad \mathcal{R} = \mathbb{p}_1(\mathbb{a}_0)\end{array}}{\mathcal{E}[\![*_\ell e]\!]^\sharp_{\mathrm{R}}(\mathbb{s},\mathbb{p}) = \big(\mathcal{R}([\mathbb{a}-\mathbb{a}_0,\ \mathbb{a}-\mathbb{a}_0+\ell[),\ (\mathbb{s}_1,\mathbb{p}_1)\big)}$$

$$\frac{\begin{array}{c}\mathcal{E}[\![e]\!]^\sharp_{\mathrm{R}}(\mathbb{s},\mathbb{p}) = \big(\mathbb{a},\ (\mathbb{s}_1,\mathbb{p}_1)\big) \\ \forall \mathbb{a}_0 \in \mathrm{dom}(\mathbb{p}_1), \neg(\mathbb{s}_1 \vDash [\mathbb{a},\mathbb{a}+\ell[\ \subseteq [\mathbb{a}_0,\mathbb{a}_0+\mathrm{size}(\mathbb{p}_1(\mathbb{a}_0))[) \qquad \mathcal{E}[\![*_\ell e]\!]^\sharp_{\mathrm{D}}\mathbb{s}_1 = (\mathbb{v},\mathbb{s}_2)\end{array}}{\mathcal{E}[\![*_\ell e]\!]^\sharp_{\mathrm{R}}(\mathbb{s},\mathbb{p}) = \big(\mathbb{v},\ \mathrm{retain}(\mathbb{s}_2,\mathbb{p}_1,\mathbb{a},\ell,\mathbb{v})\big)}$$

Figure 6.2: Abstract semantics of memory reads in $\mathbb{R}^\sharp$.

6.2.3 Abstract semantics of statements

For most statements, the abstract semantics of $\mathbb{R}^\sharp$ ignores the abstract points-to map and are the same as the abstract semantics of $\mathbb{S}^\sharp$. The only exception is memory writes.

Several regions may be retained, yet concretize to the same concrete regions. Consider again the abstract state in Figure 6.1 representing the state just before line 22: although two abstract regions are retained, nothing in the type invariants excludes that `x` and `parent` in fact point to the same memory location. (It is never the case in this program, but this property cannot be expressed in our type system.) Upon a memory write, all regions that may alias with the written region should be updated; for some regions, it may be impossible to precisely determine whether they are affected by the write operation. This is the case for the region pointed to by `parent` upon the operation `x->parent = …` at line 22. To maintain a correct abstract state, those possibly aliasing regions must be dropped before performing the write.

Definition 6.6 (Dropping aliasing regions). The operator drop_alias : $\mathbb{D}^\sharp \times \mathbb{P}^\sharp(\mathbb{D}^\sharp) \times \mathbb{V}_{\mathrm{D}} \times \mathbb{N} \to \mathbb{P}^\sharp(\mathbb{D}^\sharp)$ is defined as follows: drop_alias$(\mathbb{s},\mathbb{p},\mathbb{a},\ell)$ is the abstract points-to map obtained from $\mathbb{p}$ by dropping every points-to predicate $\mathbb{v} \mapsto \mathcal{R}$ such that $\mathbf{may\_alias}_{\mathrm{D}}(\mathbb{s},\mathbb{a},\ell,\mathbb{v},\mathrm{size}(\mathcal{R})) = \mathrm{true}$.

The set of concrete heaps represented by a points-to map is necessarily larger after dropping some mappings.

Lemma 6.1 (Stability of abstract points-to map concretization by drop_alias). *Let* $\mathbb{s} \in \mathbb{D}^\sharp$, $\mathbb{p} \in \mathbb{P}^\sharp(\mathbb{D}^\sharp)$, $\mathbb{a} \in \mathbb{V}_{\mathrm{D}}$ *and* $\ell \in \mathbb{N}$. *Then* $\gamma_{\mathrm{P}}(\mathbb{p}) \subseteq \gamma_{\mathrm{P}}(\text{drop\_alias}(\mathbb{s},\mathbb{p},\mathbb{a},\ell))$.

Proof. From Definitions 6.1 and 6.2: each points-to predicate in $\mathbb{p}$ is a constraint on the concretization $\gamma_{\mathrm{P}}(\mathbb{p})$, therefore removing such predicates removes constraints, and enlarges the concretization. □

The semantics of memory writes is defined by the following rule:

$$\frac{\begin{array}{c}\mathcal{E}[\![e_1]\!]^\sharp_{\mathrm{R}}(\mathbb{s},\mathbb{p}) = (\mathbb{a},\ (\mathbb{s}_1,\mathbb{p}_1)) \\ \mathcal{E}[\![e_2]\!]^\sharp_{\mathrm{R}}(\mathbb{s}_1,\mathbb{p}_1) = (\mathbb{v},\ (\mathbb{s}_2,\mathbb{p}_2)) \qquad \mathbb{p}_3 = \text{drop\_alias}(\mathbb{s}_2,\mathbb{p}_2,\mathbb{a},\ell) \qquad (\mathbb{s}_3,\mathbb{p}_4) = \mathrm{retain}(\mathbb{s}_2,\mathbb{p}_3,\mathbb{a},\ell,\mathbb{v})\end{array}}{[\![*_\ell e_1\ := e_2]\!]^\sharp_{\mathrm{R}}(\mathbb{s},\mathbb{p}) = ([\![*_\ell e_1\ := e_2]\!]^\sharp_{\mathrm{D}}\mathbb{s}_3,\ \mathbb{p}_4)}$$

First, all regions possibly aliasing with the region written are dropped from the abstract points-to map. Then, the memory write is performed *via* the abstract operator provided by $\mathbb{S}^\sharp$, but also retained in the abstract points-to map.

Abstract join

Joining abstract states can be done by conserving points-to predicates with the same source address—the information about which abstract addresses are "the same" being given by the two-way renaming constructed in $\mathbb{D}^\sharp$— and joining their destination regions.

However, we chose to implement a simpler join which works by first dropping the points-to predicates. We observed satisfactory precision (see Chapter 7, Section 7.3) and in particular, in cases where our analysis could not verify the properties of interest, the loss of precision was not at the join level. We therefore preferred to avoid making the implementation more complex for little gain.

Definition 6.7 (Valid two-way renamings in $\mathbb{R}^\sharp(\mathbb{D}^\sharp)$). Given $(\mathbb{s}_1, \mathbb{p}_1)$, $(\mathbb{s}_2, \mathbb{p}_2) \in \mathbb{R}^\sharp(\mathbb{D}^\sharp)$, a valid two-way renaming between them is any valid two-way renaming between $\mathbb{s}_1$ and $\mathbb{s}_2$.

Definition 6.8 (Abstract join in $\mathbb{R}^\sharp(\mathbb{D}^\sharp)$). Let $\mathbb{r}_1 = (\mathbb{s}_1, \mathbb{p}_1)$ and $\mathbb{r}_2 = (\mathbb{s}_2, \mathbb{p}_2)$ be two elements of $\mathbb{R}^\sharp(\mathbb{D}^\sharp)$. Let $\Psi \subseteq (\mathbb{V}^\sharp)^3$ be a valid two-way renaming between $\mathbb{r}_1$ and $\mathbb{r}_2$. The abstract join $\sqcup_{\mathrm{R},\Psi} : \mathbb{R}^\sharp(\mathbb{D}^\sharp) \times \mathbb{R}^\sharp(\mathbb{D}^\sharp) \to \mathbb{R}^\sharp(\mathbb{D}^\sharp)$ is defined as:

$$\mathbb{r}_1 \sqcup_{\mathrm{R},\Psi} \mathbb{r}_2 = (\mathbb{s}_1 \sqcup_{\mathrm{D},\Psi} \mathbb{s}_2,\ [])$$

The widening $\nabla_{\mathrm{R},\Psi}$ is defined similarly to the join:

Definition 6.9 (Widening in $\mathbb{R}^\sharp(\mathbb{D}^\sharp)$). Let $\mathbb{r}_1 = (\mathbb{s}_1, \mathbb{p}_1)$ and $\mathbb{r}_2 = (\mathbb{s}_2, \mathbb{p}_2)$ be two elements of $\mathbb{R}^\sharp(\mathbb{D}^\sharp)$. Let $\Psi \subseteq (\mathbb{V}^\sharp)^3$ be a valid two-way renaming between $\mathbb{r}_1$ and $\mathbb{r}_2$. The widening $\sqcup_{\mathrm{R},\Psi} : \mathbb{R}^\sharp(\mathbb{D}^\sharp) \times \mathbb{R}^\sharp(\mathbb{D}^\sharp) \to \mathbb{R}^\sharp(\mathbb{D}^\sharp)$ is defined as:

$$\mathbb{r}_1 \nabla_{\mathrm{R},\Psi} \mathbb{r}_2 = (\mathbb{s}_1 \nabla_{\mathrm{D},\Psi} \mathbb{s}_2,\ [])$$

This widening is guaranteed not to result in infinite ascending chains, as after one widening step the chains it creates are in bijection with the ascending chains created by ∇_{D}.

Abstract inclusion

The most precise abstract inclusion for this domain would consist, to verify that $\mathbb{r}_1$ is included in $\mathbb{r}_2$, in verifying that every points-to predicate of $\mathbb{r}_2$ corresponds to a predicate in $\mathbb{r}_1$ that has the same source address, and points to an included region.

However, for the same simplicity reasons as explained above for the join, we choose a simpler definition: abstract inclusion can be defined by abstracting away points-to predicates. In particular, $\mathbb{r}_1$ is included in $\mathbb{r}_2$ only if $\mathbb{r}_2$ does not have any points-to predicates.

Definition 6.10 (Valid renamings in $\mathbb{R}^\sharp(\mathbb{D}^\sharp)$). Given $\mathbb{r}_1 = (\mathbb{s}_1, \mathbb{p}_1)$ and $\mathbb{r}_2 = (\mathbb{s}_2, \mathbb{p}_2)$ two elements of $\mathbb{R}^\sharp(\mathbb{D}^\sharp)$, a valid renaming from $\mathbb{r}_2$ to $\mathbb{r}_1$ is any valid renaming from $\mathbb{s}_2$ to $\mathbb{s}_1$.

Definition 6.11 (Abstract inclusion in $\mathbb{R}^\sharp(\mathbb{D}^\sharp)$). The abstract inclusion $\sqsubseteq_{\mathrm{R},\Phi} \subseteq \mathbb{R}^\sharp(\mathbb{D}^\sharp) \times \mathbb{R}^\sharp(\mathbb{D}^\sharp)$ is defined as follows: given $\mathbb{r}_1 = (\mathbb{s}_1, \mathbb{p}_1)$ and $\mathbb{r}_2 = (\mathbb{s}_2, \mathbb{p}_2)$ two elements of $\mathbb{R}^\sharp(\mathbb{D}^\sharp)$ and $\Phi : \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ a valid renaming from $\mathbb{r}_2$ to $\mathbb{r}_1$:

$$\mathbb{r}_1 \sqsubseteq_{\mathrm{R},\Phi} \mathbb{r}_2 \iff \mathbb{p}_2 = [] \ \wedge\ \mathbb{s}_1 \sqsubseteq_{\mathrm{D},\Phi} \mathbb{s}_2$$

6.2.4 Soundness of the abstract semantics

Theorem 6.2 (Soundness of the abstract expression semantics). *Let $\mathbb{r} \in \mathbb{R}^\sharp(\mathbb{D}^\sharp)$ and $e \in$ expr. Let $(\mathbb{v}, \mathbb{r}') = \mathcal{E}[\![e]\!]^\sharp_{\mathrm{R}}\mathbb{r}$. If for all $(s, \upsilon) \in \gamma_{\mathrm{R}}(\mathbb{r})$, s is well typed under υ, then:*

1. *Expression evaluation does not modify the concrete state, therefore all states abstracted by $\mathbb{r}$ are also abstracted by $\mathbb{r}'$:*

$$(\mathrm{untyp} \circ \gamma_{\mathrm{R}})(\mathbb{r}) \subseteq (\mathrm{untyp} \circ \gamma_{\mathrm{R}})(\mathbb{r}')$$

2. *The value component is sound:*

$$\{\mathcal{E}[\![e]\!](\sigma, h) \mid (\sigma, h) \in (\text{untyp} \circ \gamma_{\text{R}})(\mathbb{r})\} \subseteq \{\gamma_{\text{R}}^{\text{V}}(v, \mathbb{v}) \mid (s, v) \in \gamma_{\text{R}}(\mathbb{r}')\}$$

Proof sketch. By induction on the syntax of expressions. Let $(\mathbb{s}, \mathbb{p}) = \mathbb{r}$.

- If e is a binary operator application ($e = e_1 \diamond e_2$): the points-to predicates are not used and not modified, i.e. $\mathcal{E}[\![e_1 \diamond e_2]\!]^{\#}_{\text{R}}(\mathbb{s}, \mathbb{p}) = (\mathcal{E}[\![e_1 \diamond e_2]\!]^{\#}_{\text{D}}\mathbb{s}, \mathbb{p})$. Therefore the induction hypothesis on e_1 and e_2 and the soundness of abstract expression semantics in $\mathbb{D}^{\#}$ yield the result.
- All other kinds of expressions except memory reads can be treated similarly.
- If e is a memory read ($e = *_{\ell} e_{\text{loc}}$): if the region read is included in one of the retained regions, then the read is sound because the concrete states abide by the retained points-to predicates (Definition 6.3); otherwise, the result of the read is sound by soundness of $\mathbb{D}^{\#}$ and the resulting abstract state is sound because the analysis retains a correct points-to predicate.

□

Theorem 6.3 (Soundness of the abstract semantics of $\mathbb{R}^{\#}(\mathbb{D}^{\#})$). *Let* $(\mathbb{s}, \mathbb{p}) \in \mathbb{R}^{\#}(\mathbb{D}^{\#})$ *and* $\text{P} \in$ *stmt. Suppose that for all* $(s, v) \in \gamma_{\text{R}}(\mathbb{s}, \mathbb{p})$*, s is well typed under* v*. Then:*

$$([\![\text{P}]\!] \circ \text{untyp} \circ \gamma_{\text{R}})(\mathbb{s}) \subseteq (\text{untyp} \circ \gamma_{\text{R}} \circ [\![\text{P}]\!]^{\#}_{\text{R}})(\mathbb{s}, \mathbb{p}).$$

In addition, for all $(s', v') \in (\gamma_{\text{R}} \circ [\![\text{P}]\!]^{\#}_{\text{R}})(\mathbb{s}, \mathbb{p})$*, s' is well typed under* v'*.*

Proof sketch. By induction on the syntax of statements. The proof is trivial for all statements types except memory writes. Regarding memory writes, the soundness derives from Lemma 6.1 and the fact that we retain a correct points-to predicate. Finally, clearly we have $(\gamma_{\text{R}} \circ [\![\text{P}]\!]^{\#}_{\text{R}})(\mathbb{s}, \mathbb{p}) \subseteq (\gamma_{\text{D}} \circ [\![\text{P}]\!]^{\#}_{\text{D}})\mathbb{s}$. Therefore $[\![\cdot]\!]^{\#}_{\text{R}}$ preserves well-typedness of states, as $[\![\cdot]\!]^{\#}_{\text{D}}$ does. □

6.3 Staged points-to predicates

6.3.1 Abstraction

Staged points-to predicates are an abstraction distinct from retained points-to predicates, although in practice, we will use them together. Unlike retained points-to predicates, staged points-to predicates represent delayed writes, and therefore they take precedence over the heap representation of the parameter domain. We express this using the notion of asymmetric merging of concrete heaps:

Definition 6.12 (Asymmetric heap merging). Given $h, h' \in \mathbb{H}$, the heap $h \triangleright h' : \text{dom}(h) \cup \text{dom}(h') \to \mathbb{V}_1$ is defined by:

$$(h \triangleright h')(a) = \begin{cases} h'(a) & \text{if } a \in \text{dom}(h') \\ h(a) & \text{otherwise} \end{cases}$$

Definition 6.13 (Staged points-to predicate domain). The staged points-to predicate domain, denoted as $\mathbb{St}^{\#}(\mathbb{D}^{\#})$, is the product of $\mathbb{D}^{\#}$ with an abstract points-to map:

$$\mathbb{St}^{\#}(\mathbb{D}^{\#}) = \mathbb{D}^{\#} \times \mathbb{P}^{\#}(\mathbb{D}^{\#})$$

Its concretization is:

$$\begin{aligned} &\gamma_{\text{St}} : \mathbb{St}^{\#}(\mathbb{D}^{\#}) \to \mathcal{P}\big(\mathbb{S}_t \times (\mathbb{V}^{\#} \to \mathbb{V})\big) \\ &\gamma_{\text{St}}(\mathbb{s}, \mathbb{p}) = \{((\sigma, \Gamma, h \triangleright h', \mathcal{L}),\ v) \mid ((\sigma, \Gamma, h, \mathcal{L}),\ v) \in \gamma_{\text{D}}(\mathbb{s}) \text{ and } (h', v) \in \gamma_{\text{P}}(\mathbb{p})\} \end{aligned}$$

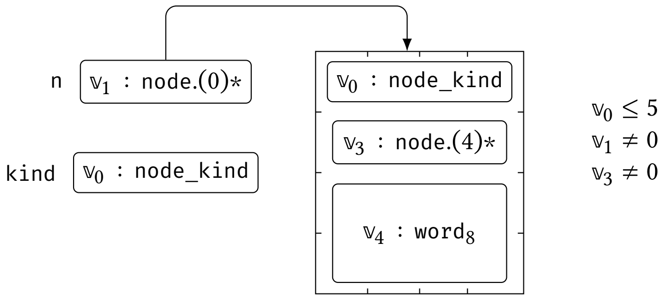

n
$\mathbb{v}_1$: node.(0)∗
kind
$\mathbb{v}_0$: node_kind
$\mathbb{v}_0$: node_kind
$\mathbb{v}_3$: node.(4)∗
$\mathbb{v}_4$: word$_8$
$\mathbb{v}_0 \leq 5$
$\mathbb{v}_1 \neq 0$
$\mathbb{v}_3 \neq 0$

Figure 6.3: Example of abstract state in $\mathbb{St}^\#(\mathbb{D}^\#)$.

Example 6.3. Figure 6.3 shows an example abstract state $\mathbb{r} = (\mathbb{s}, \mathbb{p}) \in \mathbb{St}^\#(\mathbb{S}^\#)$, where $\mathbb{p}$ is an abstract points-to map with a single points-to predicate:

$$\mathbb{s} = ([\texttt{n} \mapsto \mathbb{v}_1,\ \texttt{kind} \mapsto \mathbb{v}_0],\ [\texttt{n} \mapsto \texttt{node}.(0)*,\ \texttt{kind} \mapsto \texttt{node\_kind}])$$
$$\mathbb{p} = [\mathbb{v}_1 \mapsto \mathcal{R}]$$

where

$$\mathcal{R} = \Big[[0,4[\ \mapsto (\mathbb{v}_0, \texttt{node\_kind}),\quad [4,8[\ \mapsto (\mathbb{v}_3, \texttt{node}.(4)*),\quad [8,16[\ \mapsto (\mathbb{v}_4, \texttt{word}_8)\Big]$$

The staged points-to predicate takes precedence over the memory invariants abstracted by $\mathbb{s}$, therefore n points to a region that is not consistent with type node (because bytes 8 to 12 do not meet the necessary constraints), even though n is labelled with type node.(0)∗ in $\mathbb{s}$; and therefore some states in the concretization of $\mathbb{r}$ are ill-typed.

6.3.2 Example analysis using staged points-to predicates

Let us first describe informally the analysis of the make function from our running example (Figure 6.4a) using $\mathbb{St}^\#(\mathbb{S}^\#)$. For this analysis we can consider that $\mathbb{X}$ contains only two variables names: $\mathbb{X} = \{\texttt{n}, \texttt{kind}\}$. The analysis starts from an initial state compliant with the precondition expressed of make, i.e., kind contains a value of type node_kind which is less than or equal to 5, and n contains any value of any type. Figure 6.4b shows the abstract state just after line 43, or equivalently, after applying the abstract semantics of the $n := \textbf{malloc}_{\texttt{node}}(16)$ statement: the staged points-to predicate $\texttt{n} \mapsto \mathcal{R}$ has been added, with $\mathcal{R}$ a region containing a single, 16-byte, unconstrained value.

All subsequent writes are performed to the staged region (Figures 6.4c and 6.4d). To understand why the two middle abstract values have type node.(4)∗, recall that the C expression &n->dll translates as simply n + 4 in While-memory. After line 47, the abstract region pointed by n is consistent with type node, which can be verified in our analyzer by making sure to “commit” the entirety of the abstract points-to map, so that the analysis ends with no staged points-to predicates left, as in Figure 6.4e.

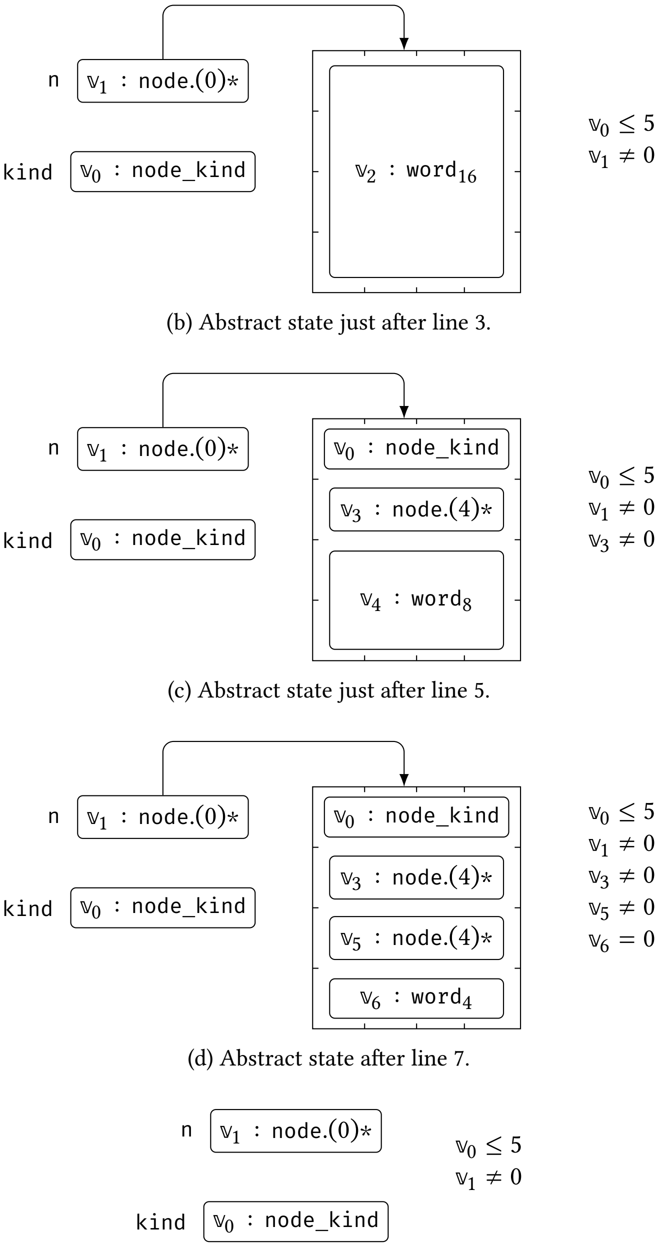
6.3.3 Abstract semantics of expressions

We now describe the abstract semantics of staged points-to predicates in detail. For all expressions except memory reads, the abstract semantics of $\mathbb{St}^\#(\mathbb{D}^\#)$ is the same as the abstract semantics of $\mathbb{D}^\#$ and does not use the abstract points-to map. The result of a read of size ℓ at address $\mathbb{a}$ is determined as follows:

- if the read region can be determined to be included in one of the regions retained in the abstract points-to map, then the read is performed from that region.

```
1  node *make(node_kind kind) {
2    node *n =
3      malloc(sizeof(node));
4    n->kind = kind;
5    n->dll.next = &n->dll;
6    n->dll.prev = &n->dll;
7    n->uf.parent = NULL;
8    return n;
9  }
```

(a) Code of fuction make.



(b) Abstract state just after line 3.

(c) Abstract state just after line 5.

(d) Abstract state after line 7.

(e) Abstract state after (successfully) committing the staged points-to predicate.

Figure 6.4: Abstract states at different points in the analysis of function make.

$$\frac{\begin{array}{c}\mathcal{E}[\![e]\!]^\sharp_{\mathrm{St}}(\mathbb{s},\mathbb{p}) = \big(\mathbb{a},\ (\mathbb{s}_1,\mathbb{p}_1)\big)\\ \exists \mathbb{a}_0 \in \mathrm{dom}(\mathbb{p}_1),\ \mathbb{s}_1 \models [\mathbb{a}, \mathbb{a}+\ell[\ \subseteq [\mathbb{a}_0, \mathbb{a}_0+\mathrm{size}(\mathbb{p}_1(\mathbb{a}_0))[\qquad \mathcal{R} = \mathbb{p}_1(\mathbb{a}_0)\end{array}}{\mathcal{E}[\![*_\ell e]\!]^\sharp_{\mathrm{St}}(\mathbb{s},\mathbb{p}) = \big(\mathcal{R}([\mathbb{a}-\mathbb{a}_0,\ \mathbb{a}-\mathbb{a}_0+\ell[),\ (\mathbb{s}_1,\mathbb{p}_1)\big)}$$

$$\frac{\begin{array}{c}\mathcal{E}[\![e]\!]^\sharp_{\mathrm{St}}(\mathbb{s},\mathbb{p}) = \big(\mathbb{a},\ (\mathbb{s}_1,\mathbb{p}_1)\big) \qquad \forall \mathbb{a}_0 \in \mathrm{dom}(\mathbb{p}_1), \neg\big(\mathbb{s}_1 \models [\mathbb{a}, \mathbb{a}+\ell[\ \subseteq [\mathbb{a}_0, \mathbb{a}_0+\mathrm{size}(\mathbb{p}(\mathbb{a}_0))[\big)\\ (\mathbb{s}_2,\mathbb{p}_2) = \mathrm{commit\_alias}((\mathbb{s}_1,\mathbb{p}_1),\mathbb{a},\ell) \qquad \mathcal{E}[\![*_\ell e]\!]^\sharp_{\mathrm{D}}\mathbb{s}_2 = (\mathbb{v},\mathbb{s}_3)\end{array}}{\mathcal{E}[\![*_\ell e]\!]^\sharp_{\mathrm{St}}(\mathbb{s},\mathbb{p}) = \big(\mathbb{v},\ (\mathbb{s}_3,\ \mathbb{p}_2)\big)}$$

Figure 6.5: Abstract semantics of memory reads in $\mathbb{St}^\sharp(\mathbb{D}^\sharp)$.

- Otherwise, all staged points-to predicates that possibly alias with the region read are "committed" to memory, i.e., the delayed writes that they represent are performed using the abstract semantics of the subdomain $\mathbb{D}^\sharp$; the memory read is also performed using the abstract semantics of $\mathbb{D}^\sharp$.

The action of "committing" delayed writes to memory is defined through the function commit_alias : $\mathbb{St}^\sharp(\mathbb{D}^\sharp) \times \mathbb{V}_{\mathrm{D}} \times \mathbb{N} \to \mathbb{St}^\sharp(\mathbb{D}^\sharp)$ which commits to memory all the delayed writes that may alias with a given region:

Definition 6.14 (commit_alias function). The function commit_alias is defined by Algorithm 6.1.

Algorithm 6.1 Algorithm of function commit_alias.

```
function commit_alias(($\mathbb{s}$, $\mathbb{p}$), $\mathbb{a}$, $\ell$)
    $\mathbb{s}' \leftarrow \mathbb{s}$
    $\mathbb{p}' \leftarrow \mathbb{p}$
    for all mappings $\mathbb{v} \mapsto \mathcal{R}$ in $\mathbb{p}$ do
        if may_alias$_{\mathrm{D}}(\mathbb{s}, \mathbb{a}, \ell, \mathbb{v}, \mathrm{size}(\mathcal{R}))$ then
            $\mathbb{s}' \leftarrow$ result of writing $\mathcal{R}([0, \mathrm{size}(\mathcal{R})[)$ to $\mathbb{s}'$ at address $\mathbb{v}$ in the semantics of $\mathbb{D}^\sharp$
            $\mathbb{p}' \leftarrow \mathbb{p}'[\mathbb{v} \leftarrow \bot]$
    return $(\mathbb{s}', \mathbb{p}')$
```

The detailed semantics of memory reads is given in Figure 6.5.

6.3.4 Abstract semantics of statements

We now define the abstract semantics of statements, $[\![s]\!]^\sharp_{\mathrm{St}} : \mathbb{St}^\sharp(\mathbb{D}^\sharp) \to \mathbb{St}^\sharp(\mathbb{D}^\sharp)$. This abstract semantics extends $[\![\cdot]\!]^\sharp_{\mathrm{D}}$, the abstract semantics of $\mathbb{D}^\sharp$, with an abstract points-to map, while remaining sound with regard to the concrete semantics $[\![\cdot]\!]$.

For most statements, the abstract semantics $[\![s]\!]^\sharp_{\mathrm{St}}$ is identical to the semantics of $\mathbb{D}^\sharp$ and does not modify the abstract points-to map, that is, $[\![s]\!]^\sharp_{\mathrm{St}}(\mathbb{s},\mathbb{p}) = ([\![s]\!]^\sharp_{\mathrm{D}}\mathbb{s},\ \mathbb{p})$. The two exceptions are memory write and memory allocation.

Memory writes Memory writes are performed as follows:

$$\frac{\begin{array}{c}\mathcal{E}\llbracket e_1 \rrbracket^\sharp_{\mathrm{St}}(\mathbb{s}, \mathbb{p}) = \big(\mathbb{a},\ (\mathbb{s}_1, \mathbb{p}_1)\big) \\ \mathcal{E}\llbracket e_2 \rrbracket^\sharp_{\mathrm{St}}(\mathbb{s}_1, \mathbb{p}_1) = \big(\mathbb{v},\ (\mathbb{s}_2, \mathbb{p}_2)\big) \qquad \exists \mathbb{a}_0 \in \mathrm{dom}(\mathbb{p}_2),\ \mathbb{s}_2 \vDash [\mathbb{a}, \mathbb{a}{+}\ell[\ \subseteq [\mathbb{a}_0, \mathbb{a}_0{+}\mathrm{size}(\mathbb{p}_2(\mathbb{a}_0))[\\ \mathcal{R} = \mathbb{p}_2(\mathbb{a}_0) \qquad \mathcal{R}' = \mathcal{R}\big[\,[\mathbb{a}{-}\mathbb{a}_0,\ \mathbb{a}{-}\mathbb{a}_0{+}\ell[\ \leftarrow \mathbb{v}\,\big]\end{array}}{\llbracket *_\ell e_1 \ := e_2 \rrbracket^\sharp_{\mathrm{St}}(\mathbb{s}, \mathbb{p}) = \big(\mathbb{s}_2,\ \mathbb{p}_2\big[\mathbb{a}_0 \leftarrow \mathcal{R}'\big]\big)}$$

$$\frac{\begin{array}{c}\mathcal{E}\llbracket e_1 \rrbracket^\sharp_{\mathrm{St}}(\mathbb{s}, \mathbb{p}) = \big(\mathbb{a},\ (\mathbb{s}_1, \mathbb{p}_1)\big) \\ \mathcal{E}\llbracket e_2 \rrbracket^\sharp_{\mathrm{St}}(\mathbb{s}_1, \mathbb{p}_1) = \big(\mathbb{v},\ (\mathbb{s}_2, \mathbb{p}_2)\big) \qquad \forall \mathbb{a}_0 \in \mathrm{dom}(\mathbb{p}_2), \neg\big(\mathbb{s}_2 \vDash [\mathbb{a}, \mathbb{a}{+}\ell[\ \subseteq [\mathbb{a}_0, \mathbb{a}_0{+}\mathrm{size}(\mathbb{p}_2(\mathbb{a}_0))[\big) \\ (\mathbb{s}_3, \mathbb{p}_3) = \mathrm{commit\_alias}((\mathbb{s}_2, \mathbb{p}_2), \mathbb{a}, \ell) \qquad \mathrm{should\_retain}_{\mathrm{D}}(\mathbb{s}_3, \mathbb{a}) = (\mathrm{true}, i, n)\end{array}}{\llbracket *_\ell e_1 \ := e_2 \rrbracket^\sharp_{\mathrm{St}}(\mathbb{s}, \mathbb{p}) = \mathrm{retain}(\mathbb{s}_2, \mathbb{p}_2, \mathbb{a}, \ell, \mathbb{v})}$$

$$\frac{\begin{array}{c}\mathcal{E}\llbracket e_1 \rrbracket^\sharp_{\mathrm{St}}(\mathbb{s}, \mathbb{p}) = \big(\mathbb{a},\ (\mathbb{s}_1, \mathbb{p}_1)\big) \\ \mathcal{E}\llbracket e_2 \rrbracket^\sharp_{\mathrm{St}}(\mathbb{s}_1, \mathbb{p}_1) = \big(\mathbb{v},\ (\mathbb{s}_2, \mathbb{p}_2)\big) \qquad \forall \mathbb{a}_0 \in \mathrm{dom}(\mathbb{p}_2), \neg\big(\mathbb{s}_2 \vDash [\mathbb{a}, \mathbb{a}{+}\ell[\ \subseteq [\mathbb{a}_0, \mathbb{a}_0{+}\mathrm{size}(\mathbb{p}_2(\mathbb{a}_0))[\big) \\ (\mathbb{s}_3, \mathbb{p}_3) = \mathrm{commit\_alias}((\mathbb{s}_2, \mathbb{p}_2), \mathbb{a}, \ell) \qquad \mathrm{should\_retain}_{\mathrm{D}}(\mathbb{s}_3, \mathbb{a}) = \mathrm{false}\end{array}}{\llbracket *_\ell e_1 \ := e_2 \rrbracket^\sharp_{\mathrm{St}}(\mathbb{s}, \mathbb{p}) = \Big(\llbracket *_\ell e_1 \ := e_2 \rrbracket^\sharp_{\mathrm{D}} \mathbb{s}_3,\ \mathbb{p}_2\Big)}$$

Figure 6.6: Abstract semantics of memory writes in $\mathbb{St}^\sharp(\mathbb{D}^\sharp)$.

- if the region written is contained in one of the destination regions of the staged points-to predicates, then the write is performed in that region.
- Otherwise, all staged points-to predicates that may alias with the region written are committed to memory, and then a new points-to predicate is added to represent the write, if possible (that is, if $\mathrm{should\_retain}_{\mathrm{D}}$ returns true). Otherwise, the write is performed in the parameter domain $\mathbb{D}^\sharp$.

These rules are expressed in full in Figure 6.6.

Memory allocation The abstract operator $\llbracket x := \mathbf{malloc}_t(e) \rrbracket^\sharp_{\mathrm{St}}$ for memory allocation is defined as follows. It uses the allocation semantics of the subdomain $\mathbb{D}^\sharp$, but in addition, creates a new region in the abstract points-to map and fills it with a fresh, unconstrained abstract value. Formally, it is defined by the following rule:

$$\frac{\begin{array}{c}\mathcal{E}\llbracket e \rrbracket^\sharp_{\mathrm{St}} \mathbb{s} = (\mathbb{v}_\ell,\ (\mathbb{s}_1, \mathbb{p}_1)) \qquad \exists \ell \in \mathbb{N},\ \mathbb{s}_1 \vDash \mathbb{v}_\ell = \ell \\ \llbracket x \ := \mathbf{malloc}_t(e) \rrbracket^\sharp_{\mathrm{D}} \mathbb{s}_1 = \mathbb{s}_2 \qquad \mathcal{E}\llbracket x \rrbracket^\sharp_{\mathrm{D}} \mathbb{s}_2 = (\mathbb{a}, \mathbb{s}_3) \qquad \mathcal{R} = \big[\,[0, \ell[\ \mapsto \mathbb{v}\,\big] \qquad \mathbb{v}\ \mathrm{fresh}\end{array}}{\llbracket x \ := \mathbf{malloc}_t(e) \rrbracket^\sharp_{\mathrm{St}}(\mathbb{s}, \mathbb{p}) = (\mathbb{s}_3,\ \mathbb{p}_1[\mathbb{a} \leftarrow \mathcal{R}])}$$

Memory allocation when $\mathbb{D}^\sharp = \mathbb{S}^\sharp$ When the parameter domain is $\mathbb{S}^\sharp$, we slightly change its abstract semantics of memory allocation in order to gain precision: we no longer require that the allocated type be compatible with any value. This check was necessary for the abstract semantics of $\mathbb{S}^\sharp$ to preserve typing, as discussed in Example 5.9. But in $\mathbb{St}^\sharp(\mathbb{S}^\sharp)$, it becomes redundant with the added points-to predicate; the type checking is not suppressed, but delayed until the staged predicate is committed. This can be seen in Section 6.3.2 above: `n` has type `node`.(0)∗, even though it points to an unconstrained value (i.e. a value that is not necessarily in the interpretation of type `node`). While this makes the state ill-typed, it is still sound with respect to the concrete semantics.

Abstract join

As with retained points-to predicates, staged points-to predicates are approximated away before join. Only, in this case, the approximation consists in committing them, instead of dropping them. For this purpose, we define an auxiliary function "commit_all" that commits all points-to predicates to the underlying state.

Definition 6.15 (commit_all function). The function commit_all : $\mathbb{St}^\sharp(\mathbb{D}^\sharp) \to \mathbb{D}^\sharp$ is defined as follows: the abstract state commit_all($\mathbb{s}, \mathbb{p}$) is the result of the following algorithm: for every points-to predicate $\mathbb{v} \mapsto \mathcal{R}$ in $\mathbb{p}$, write $\mathbb{v}$ to $\mathbb{s}$ according to the abstract semantics of memory writes in $\mathbb{D}^\sharp$, and drop that points-to predicate from $\mathbb{p}$.

Definition 6.16 (Valid two-way renamings in $\mathbb{St}^\sharp(\mathbb{D}^\sharp)$). Given $(\mathbb{s}_1, \mathbb{p}_1)$, $(\mathbb{s}_2, \mathbb{p}_2) \in \mathbb{St}^\sharp(\mathbb{D}^\sharp)$, a valid two-way renaming between them is any valid two-way renaming between $\mathbb{s}_1$ and $\mathbb{s}_2$.

Definition 6.17 (Abstract join in $\mathbb{St}^\sharp(\mathbb{D}^\sharp)$). Let $\mathbb{x}_1 = (\mathbb{s}_1, \mathbb{p}_1)$ and $\mathbb{x}_2 = (\mathbb{s}_2, \mathbb{p}_2)$ be two elements of $\mathbb{St}^\sharp(\mathbb{D}^\sharp)$. Let $\Psi \subseteq (\mathbb{V}^\sharp)^3$ be a valid two-way renaming between $\mathbb{x}_1$ and $\mathbb{x}_2$. The abstract join $\sqcup_{\mathrm{St},\Psi}$: $\mathbb{St}^\sharp(\mathbb{D}^\sharp) \times \mathbb{St}^\sharp(\mathbb{D}^\sharp) \to \mathbb{St}^\sharp(\mathbb{D}^\sharp)$ is defined as:

$$\mathbb{x}_1 \sqcup_{\mathrm{St},\Psi} \mathbb{x}_2 = (\text{commit\_all}(\mathbb{x}_1) \sqcup_{\mathrm{St},\Psi} \text{commit\_all}(\mathbb{x}_2),\ [])$$

The widening is defined similarly:

Definition 6.18 (Widening in $\mathbb{St}^\sharp(\mathbb{D}^\sharp)$). Let $\mathbb{x}_1 = (\mathbb{s}_1, \mathbb{p}_1)$ and $\mathbb{x}_2 = (\mathbb{s}_2, \mathbb{p}_2)$ be two elements of $\mathbb{St}^\sharp(\mathbb{D}^\sharp)$. Let $\Psi \subseteq (\mathbb{V}^\sharp)^3$ a valid two-way renaming between $\mathbb{x}_1$ and $\mathbb{x}_2$. The widening $\sqcup_{\mathbb{St}^\sharp,\Psi}$: $\mathbb{St}^\sharp(\mathbb{D}^\sharp) \times \mathbb{St}^\sharp(\mathbb{D}^\sharp) \to \mathbb{St}^\sharp(\mathbb{D}^\sharp)$ is defined as:

$$\mathbb{x}_1 \triangledown_{\mathrm{R},\Psi} \mathbb{x}_2 = (\text{commit\_all}(\mathbb{s}_1, \mathbb{p}_1) \triangledown_{\mathrm{D},\Psi} \text{commit\_all}(\mathbb{s}_2, \mathbb{p}_2),\ [])$$

This widening does not result in infinite ascending chains, which we can justify with an argument similar to the one used for $\triangledown_{\mathrm{R}}$.

Abstract inclusion

Definition 6.19 (Valid renamings in $\mathbb{St}^\sharp(\mathbb{D}^\sharp)$). Given $\mathbb{x}_1 = (\mathbb{s}_1, \mathbb{p}_1)$ and $\mathbb{x}_2 = (\mathbb{s}_2, \mathbb{p}_2)$ two elements of $\mathbb{St}^\sharp(\mathbb{D}^\sharp)$, a valid renaming from $\mathbb{x}_2$ to $\mathbb{x}_1$ is any valid renaming from $\mathbb{s}_2$ to $\mathbb{s}_1$.

Definition 6.20 (Abstract inclusion in $\mathbb{St}^\sharp(\mathbb{D}^\sharp)$). The abstract inclusion $\sqsubseteq_{\mathrm{St},\Phi} \subseteq \mathbb{St}^\sharp(\mathbb{D}^\sharp) \times \mathbb{St}^\sharp(\mathbb{D}^\sharp)$ is defined as follows: given $\mathbb{x}_1 = (\mathbb{s}_1, \mathbb{p}_1)$ and $\mathbb{x}_2 = (\mathbb{s}_2, \mathbb{p}_2)$ two elements of $\mathbb{St}^\sharp(\mathbb{D}^\sharp)$ and $\Phi : \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ a valid renaming from $\mathbb{x}_2$ to $\mathbb{x}_1$:

$$\mathbb{x}_1 \sqsubseteq_{\mathrm{St},\Phi} \mathbb{x}_2 \iff \mathbb{p}_2 = [] \wedge \text{commit\_all}(\mathbb{s}_1, \mathbb{p}_1) \sqsubseteq_{\mathrm{D},\Phi} \mathbb{s}_2$$

As with retained predicates, the join and inclusion operations could be more precise. In fact, the join and inclusion on points-to predicates described in Section 6.2.3 for *retained* points-to predicates can be applied to *staged* points-to predicates as well. However, as with retained points-to predicates, we preferred to keep our implementation simple and more conservative as the precision gains did not seem to justify the added complication to the analysis (see experiments Section 7.3).

6.3.5 Soundness of the abstract semantics

In essence, the staged points-to predicate abstract domain $\mathbb{St}^\sharp(\mathbb{D}^\sharp)$ only *delays* memory writes, as long as aliasing relations permit.

For this reason, its abstract semantics is sound, but well-typedness is preserved only in abstract states where the staged predicates have been committed, i.e., states whose points-to map is empty.

Theorem 6.4 (Soundness of the abstract semantics of $\mathbb{St}^\sharp(\mathbb{D}^\sharp)$). *Let* $(\mathbb{s}, \mathbb{p}) \in \mathbb{St}^\sharp(\mathbb{D}^\sharp)$ *and* $\mathrm{P} \in$ *stmt. Suppose that for all* $(s, v) \in \gamma_{\mathrm{St}}(\mathbb{s}, \mathbb{p})$*, s is well typed under v. Let* $(\mathbb{s}', \mathbb{p}') = [\![\mathrm{P}]\!]^\sharp_{\mathrm{St}}(\mathbb{s}, \mathbb{p})$*. Then:*

$$([\![\mathrm{P}]\!] \circ \mathrm{untyp} \circ \gamma_{\mathrm{St}})(\mathbb{s}) \subseteq (\mathrm{untyp} \circ \gamma_{\mathrm{St}})(\mathbb{s}', \mathbb{p}').$$

In addition, if $\mathrm{dom}(\mathbb{p}') = \emptyset$*, then for all* $(s', v') \in \gamma_{\mathrm{St}}(\mathbb{s}', \mathbb{p}')$*, s' is well typed under v'.*

The fact that well-typedness is preserved (when the abstract points-to map is empty) is analogous to preservation theorems in type systems, which state that an evaluation step preserves the well-typed nature of a term.

Since only the abstract states with an empty points-to maps are sound abstractions of the concrete states, our analyzer makes sure to commit any remaining staged points-to predicate after computing the abstract semantics of a program. This way, if the analysis terminates without any alarm, it verifies the spatial memory safety and preservation of type invariants.

6.4 Combining retained and staged points-to predicates

In practice, we shall use the retained and staged points-to predicates, as well as the type-based shape domain:

Definition 6.21 (Type-based shape domain with points-to predicates). We let $\mathbb{F}^\sharp$ denote the abstract domain consisting in $\mathbb{S}^\sharp$, used as the parameter of $\mathbb{R}^\sharp$, in turn used as the parameter of $\mathbb{St}^\sharp$:

$$\mathbb{F}^\sharp = \mathbb{St}^\sharp(\mathbb{R}^\sharp(\mathbb{S}^\sharp))$$

The set of abstract values of $\mathbb{F}^\sharp$ is the same as $\mathbb{S}^\sharp$: $\mathbb{V}_{\mathrm{F}} = \mathbb{V}_{\mathrm{St}} = \mathbb{V}^\sharp \times \mathbb{T}^\sharp$.

6.5 Conclusion

After discussing the limitations of the type-based shape domain regarding the need to retain non type-related predicates and to allow temporary violations of typing invariants, we proposed two new abstract domains to mitigate them. These abstract domains consist in parameterized abstractions which use points-to predicates to either *refine* (in the case of retained points-to predicates) or *modify* (in the case of staged points-to predicates) the concrete memory states represented by an abstract state.

In the context of machine code analysis, Balakrishnan and Reps introduced the *recency abstraction* [BR06]. Similarly to some context-sensitive pointer analyses, memory regions allocated dynamically are abstracted by their allocation site. However, this abstraction is made more precise by distinguishing the most recently allocated region from other regions allocated at the same site.

The idea of distinguishing the most recently allocated regions is in a way similar to the points-to predicates, except that (retained and staged) points-to predicates distinguish the most recently *accessed* memory regions, whether for reading, writing, or allocation, while the recency abstraction only distinguishes the most recently allocated one. Our abstractions could be parameterized by a domain that abstracts heap objects by their allocation sites (rather than their types, as $\mathbb{S}^\sharp$ does), resulting in an abstraction similar to the recency abstraction.

For example, consider a program allocating and initializing several `node` objects by calling `make` in a loop, and then calling `merge` on some of them (see code Figure 4.1). Balakrishnan and Reps's recency abstraction, like ours, would be able to verify that `make` initializes `node`s in a way compliant with typing invariants; however, it would fail to verify that the memory access `parent->parent` at line 20 is safe, because `*x` is abstracted by the same abstract object as all other `nodes` (since all share the same allocation site) and therefore the effect of the condition `x->parent != 0` cannot be represented.

Chapter 7

Practical analysis of C and machine code programs

Outline of the current chapter

So far, we have defined our analysis on the simple WHILE-MEMORY language. We now turn to real-world programs. In this thesis, we focus on C and machine code. We describe in Section 7.1 how we construct our analysis of C programs. We then turn to machine code programs, and define a semantics for them (Section 7.2.1), assuming an intermediate representation of assembly instructions in WHILE-MEMORY augmented with control flow jumps. Section 7.2 then describes the analysis of machine code, in particular the inference of control flow. We then describe the experiments we carried out to evaluate the efficiency and effectiveness of our analysis, on both C and machine code programs, in Section 7.3. Finally, Section 7.4 covers existing works on static analysis of low-level code.

7.1 Analysis of C programs

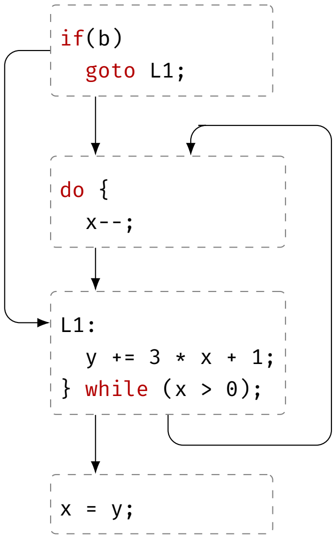
Our static analysis is implemented as part of Codex, an abstract interpretation library developed at CEA. Codex is written in the OCaml programming language, and provides various numerical and memory abstractions. The implementation of our full static analysis using physical types and retained and staged points-to predicates amounts to 3,300 source lines of code (SLOC). It is common to both C and machine code analysis tools. It exports an interface that consists essentially of the operators of the abstract semantics of WHILE-MEMORY. We implemented the abstract domains $\mathbb{S}^\sharp$ and $\mathbb{F}^\sharp$, parameterized by the choice of a numerical abstract domain.

The C analysis is implemented as a module of the Frama-C tool, that uses the Codex library. We call the resulting tool Frama-C/Codex. Frama-C is an open-source platform dedicated to the analysis of C programs. It facilitates the development of program analyses by handling the parsing of C code and proposing a library to iterate on the resulting AST. Its modular architecture allows to develop analyses as independent plugins.

The Frama-C module exploiting the Codex library to analyze C programs consists of 2,817 SLOC. Conceptually, it translates the input C program into a WHILE-MEMORY program, and executes the abstract semantics of that program.

The following adaptations are required for the translation from C to WHILE-MEMORY:

- *Function calls* are inlined (full context-sensitivity); thus we do not handle recursive functions. Function calls could be more finely managed by introducing a stack abstraction, but this is not relevant to our goal of verifying memory-related properties on sufficiently small programs without recursive functions.
- *Local variables* could be translated into elements of $\mathbb{X}$, in such a way that two distinct variables are associated to distinct variables. While this method is the simplest, it does not allow local variables to have an address—e.g., the expression `&x` should evaluate to the address of variable `x`. For this reason, we consider local variables to be stored in the heap, but each at a set of special addresses, e.g., $\mathbb{A}_{\mathtt{x}} \subseteq \mathbb{A}$ is the set of (contiguous) addresses holding variable `x`. If `x` and `y` are distinct variables, then $\mathbb{A}_{\mathtt{x}} \cap \mathbb{A}_{\mathtt{y}} = \emptyset$. Scope-local variables (e.g., in a function) are allocated at the beginning of their scope and freed at the end.
- *Floating-point computations* could be handled by using a numerical abstraction with floating-point support, but since we had no floating point case studies and numerical analyses are an independent issue, our implementation over-approximates floating-point operations by fully unknown values.
- *Data structures* are handled in specific ways: accesses in `struct` and `union` lvalues are translated to the corresponding pointer arithmetic; `enum` types are translated to integers. Note that this translation of access paths to pointer arithmetic only depends on the declared C types, as is the case in the compilation process of a usual compiler; the translation to WHILE-MEMORY does not involve our system of physical types at any point; those only matter at the time of the analysis (see below Section 7.1.3).
- Structure fields with a size that is not a multiple of one byte, which the C standard calls *bit-fields*, are currently not supported because the granularity of address type offsets in physical types is one byte. In principle, however, pointer offsets could easily be made bit-precise, and the implementation could be improved accordingly.
- *Dynamic memory allocation* is handled by translating calls to the `malloc` standard function into the **malloc** WHILE-MEMORY statement. Calls to `free` are translated as no-ops. Calls to `calloc` or `realloc` can be translated in terms of `malloc`.



```
if(b)
  goto L1;

do {
  x--;

L1:
  y += 3 * x + 1;
} while (x > 0);

x = y;
```

Figure 7.1: CFG not directly translatable to While-memory.

7.1.1 Semantics of programs with arbitrary control flow

In C, the control flow may be more complex than in While-memory. Constructs like `switch` statements, `for` loops or `do { ...} while` loops can be easily translated into While-memory control flow commands, but this is not true when `goto` or `break` statements are involved, as in the program example of Figure 7.1.

For this reason, in reality our tool does not translate the C program to a single While-memory statement, but to a control flow graph (CFG). In the context of this thesis, a CFG is a graph whose nodes are program locations and whose edges are labelled with abstract semantic operators:

Definition 7.1 (Control flow graph). We denote as $\mathcal{L}$ the set of all *program locations* in the C program P. Then, $\text{cfg}[\text{P}] \in \mathcal{P}\big(\mathcal{L} \times (\mathbb{F}^\sharp \to \mathbb{F}^\sharp) \times \mathcal{L}\big)$ gives the edges of the control flow graph of P as a set of triples $(\ell_1, \text{F}^\sharp, \ell_2)$, where ℓ_1 and ℓ_2 are program locations and $\text{F}^\sharp : \mathbb{F}^\sharp \to \mathbb{F}^\sharp$ is the abstract semantics of the program between locations ℓ_1 and ℓ_2. There is at most one edge between two locations.

This CFG can be obtained by a simple process: first, transforming the C program into a graph of C statements, the edges between instructions being optionally labeled with a condition; then, each C instruction is translated to a While-memory statement. The CFG, in the sense of Definition 7.1, is obtained by labelling each edge from ℓ_1 to ℓ_2 with the semantics of the statement at ℓ_1, possibly refined by a condition.

We associate to each program location ℓ an abstract state $\mathbb{s}_\ell \in \mathbb{F}^\sharp$. Additionally, we assume that the graph cfg[P] possesses an initial and a final node, corresponding to locations ℓ_0 and ℓ_f, respectively. Finally, we assume $\mathbb{s}_0$ to be a sound abstraction of the initial states at ℓ_0. The abstract semantics of a program P is then defined by the equation system:

$$\forall \ell \in \mathcal{L},\ \mathbb{s}_\ell = \begin{cases} \mathbb{s}_0 & \text{if } \ell = \ell_0 \\ \displaystyle\bigsqcup_{(i, \text{F}^\sharp, \ell)} \text{F}^\sharp(\mathbb{s}_i) & \text{otherwise} \end{cases}$$

The solution of this equation system can be over-approximated by the standard process [CC77] of computing a post-fixpoint using the abstract join and widening operators of the abstract domain (Section 3.4.3). This method of computing abstract semantics has been present since the beginning of abstract interpretation, introduced by Cousot and Cousot [CC77]. In a separate publication [Cou77], P. Cousot introduced an efficient method of computation by chaotic iterations. This method has been further developed by Bourdoncle [Bou93] who introduced methods to chose suitable widening strategies so as to ensure termination while preserving precision, on arbitrary control flow graphs. A didactic account of the solving method is given in [Min17, Section 3.5.3].

7.1.2 Under-specified behaviors

The official specification of the C language [IsoC18] does not entirely specify the behavior of programs: some cases are left unspecified, either because they depend on the architecture, or because compiler optimizations may favor some choices in order to gain efficiency.

The list of not fully specified behaviors [IsoC18, Annex J] distinguishes four kinds of incomplete specifications:

- *Undefined behaviors* are situations in which the standard does not impose any requirement about what should happen. Examples include null pointer dereference, or conversion of an integer value that is outside of the range that can be represented by the destination type, or pointer conversion producing a pointer with an incorrect alignment. Since an undefined behavior situation gives absolutely no guarantees about the behavior of the program, they are considered to be errors, except when additional assumptions can be made on the compiler and hardware.
- *Unspecified behaviors* are cases where the standard provides several possibilities but does not impose one of them. The order of evaluation of binary operator operands is an example of unspecified behavior, or the value of padding bits.
- *Implementation-defined behaviors* are unspecified behaviors that should be documented by each implementation of the language, such as the accuracy of floating-point operations, or the result of converting a pointer into an integer.
- Finally, *locale-specific behaviors* are behaviors dependent on linguistic conventions, mostly about output format or character sets.

Our static analysis is a *refinement* of the standard, in the sense that:

- It implements everything the standard mandates, or documents it when it is not the case;
- It makes one of the available choices for each unspecified behavior, and documents that choice.

For example, by nature, translating physical types from C types requires assumptions about the architecture word size and endianness, and the number of padding bytes between fields.

Finally, the static analysis will warn about undefined behaviors, with two exceptions. Since temporal memory safety is out of our scope, accessing an object outside of its lifetime is not detected; and some pointer comparisons that standardly result in undefined behavior are not always detected. Indeed, the behavior is undefined when “Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators” [IsoC18, Annex J.2]. The abstract domain $\mathbb{F}^\sharp$ is very imprecise in determining whether two pointers point to the same object, because it mostly groups heap objects by type. For this reason, we make an additional assumption on the language implementation in this case, namely that execution continues normally, although the result of the comparison is unspecified.

7.1.3 Manual annotations required by the type-based shape domain

To use the type-based shape domain, one must instantiate the type mapping $\mathcal{M} : \mathcal{N} \to \mathbb{T}$ from type names to physical types described in Chapter 4. Then, there are exactly two ways to introduce physical types in the analysis of a program. The first is by specifying the types of global and local variables in the initial state; the second is by annotating calls to `malloc` with a physical type.

Note that the set of physical types will usually be a refinement of the declared C types, but this is not necessary: the analysis will be sound either way.

Annotation of `malloc` calls with physical types

As described in Section 5.4.3, in While-memory, memory allocation can be annotated with a type hint, specifying what type the allocated region should be.

When translating C to While-memory, most of the time this annotation can be automatically inferred from the C type of the lvalue the `malloc` is assigned to, if this C type is associated to a physical type in the parameters of the analysis.

Example 7.1 (Automatic addition of a type hint to a memory allocation). Consider the following statement:

```
struct node *x = malloc(sizeof(struct node));
```

The analysis can be parameterized to map the C type `struct node` to the physical type `node` from Figure 4.2. In that case, the above statement will be translated into e.g.:

$$x := \mathbf{malloc}_{\texttt{node}}(16)$$

The analysis can also be parameterized to attribute the most generic types to the allocated region, namely $\texttt{word}_{16}$ in that case:

$$x := \mathbf{malloc}_{\texttt{word}_{16}}(16)$$

Note that *type annotations do not influence the soundness of the analysis*, only its precision.

7.2 Analysis of executables

Our analysis is not limited to C programs, but also handles machine code (also called binary code). The machine code analysis is implemented as a module of the Binsec [Dav+16] tool: we name the resulting analyzer Binsec/Codex. Binsec is a machine code analysis platform that provides a set of utilities and libraries to parse executable files, and disassemble the machine code they contain into an intermediate representation. This lifting to an intermediate representation allows our analysis to operate independently from the destination architecture of the executable: in our experiments, we were able to analyze executables destined for the 32-bit x86 and ARM architectures (see Section 7.3 and Chapter 10); Binsec also supports the RISC-V architecture. Binsec's intermediate representation is very close to While-memory, except for the fact that it permits *control flow jumps* (or simply *jumps*), both static —i.e., jumps to a static code address— and dynamic, i.e., jumps whose destination is computed at runtime. We first define a concrete semantics of machine code in Section 7.2.1, and then present how we deal with dynamic jumps using a control flow abstraction in Section 7.2.2.

As in our C analyzer, function calls are inlined; however, since the notion of function is not part of the semantics of machine code, we resort to a (sound) heuristic, detailed in Section 7.2.3.

The state of the system during the execution of a machine code program is abstracted as follows: $\mathbb{X}$ is defined as the set of CPU registers; and the memory abstraction is a product between the type-based shape domain and a more classical memory abstraction: we give more details below in Section 7.2.4.

$$\begin{array}{lcll} \mathit{jump} & ::= & \mathbf{stop} & \\ & \mid & \mathbf{goto}\ e & (e \in \mathit{expr}) \\ & \mid & \mathbf{if}\ e\ \mathbf{then\ goto}\ e_1\ \mathbf{else\ goto}\ e_2 & (e, e_1, e_2 \in \mathit{expr}) \end{array}$$

Figure 7.2: Grammar of dynamic jumps.

7.2.1 A semantics of machine code

We first define the semantics of assembly instructions. For that purpose, we assume that all instructions are representable in WHILE-MEMORY. For this to be true, we need to extend the language with control flow jumps.

Definition 7.2 (WHILE-MEMORY statements with dynamic jumps). To account for dynamic jumps, we define a new type of statement with the *jump* grammar in Figure 7.2. Further, we assume that every assembly instruction can be decoded into a WHILE-MEMORY statement, followed by a jump, i.e., an element of $\mathit{stmt} \times \mathit{jump}$. We denote as $\mathcal{I} \subseteq \mathit{stmt} \times \mathit{jump}$ the set of instructions of the considered architecture.

This representation is suitable to translate all assembly instructions. For example, consider the x86 instruction `je 0x000000bf`, or "jump if equal", whose semantics is the following: in x86, the flag `ZF` is set if the last comparison instruction concluded to an equality, or if the last subtraction result was zero. This instruction jumps to the instruction located at a relative offset of `0x000000bf` to the current location if the `ZF` flag is set; otherwise, it jumps to the next instruction. This would result in the following element of $\mathit{stmt} \times \mathit{jump}$ (we separate the *stmt* element and the *jump* element by a semicolon):

$$\begin{array}{l} \mathbf{skip}; \\ \mathbf{if}\ \mathtt{ZF} = 1\ \mathbf{then\ goto}\ \mathtt{eip} + \mathtt{0xbf}\ \mathbf{else\ goto}\ \mathtt{eip} + 6 \end{array}$$

The x86 "result is zero" flag, `ZF`, and the program counter `eip`, are both translated as WHILE-MEMORY variables. When the jump condition is not met, the operand of **goto** is `eip`+6, that is, the current program counter plus 6, the size of the current instruction. In other words, when the condition is not met, the jump target is the instruction immediately following the current one.

The dynamic jump that reads a location in memory at address `0xdeadbeef` and jumps to that address, would be translated as:

$$\begin{array}{l} \mathbf{skip}; \\ \mathbf{goto}\ *_4\mathtt{0xdeadbeef} \end{array}$$

Note that this representation of assembly instructions allows for complex semantics, including loops. Indeed, the semantics of some instructions can involve loops, such as `rep movsd` in x86 assembly which writes contiguous bytes in memory a specified number of times, or until the `ZF` flag is no longer set.

Instructions requiring a certain privilege level can also be expressed in WHILE-MEMORY, by storing the current privilege level in a variable. Trying to execute an instruction without the required privilege results in a transition to the error state, Ω. Unauthorized memory accesses and other kinds of run-time errors, such as divisions by zero, also cause a transition to Ω. This is an abstraction of the real semantics of most hardware architectures, in which such faults trigger a specific interrupt. However, we do not need to model them more precisely, since our analysis verifies the absence of such run-times errors and emits an alarm, should it fail.

We now define formally the semantics of such instructions:

Notation 7.1 (Program counter in machine code analysis). *When analyzing machine code, we model the program counter by a variable* `pc` *which contains the address of the current instruction. For brevity,* *given $s = (\sigma, h) \in \mathbb{S}$, we denote as $s.\mathtt{pc}$ the value $\sigma(\mathtt{pc})$, and as $s[\mathtt{pc} \leftarrow v]$ the state with* pc *updated to v: $s[\mathtt{pc} \leftarrow v] = (\sigma[\mathtt{pc} \leftarrow v], h)$.*

We define the semantics of an assembly instruction as the set of states reachable through that instruction:

Definition 7.3 (Semantics of assembly instructions). The semantics of assembly instructions $[\![\cdot]\!]_{\mathcal{I}} : \mathcal{I} \times \mathbb{S} \to \mathcal{P}(\mathbb{S})$ are defined as follows:

$$[\![\mathrm{P}, \textbf{stop}]\!]_{\mathcal{I}} \mathrm{S} = \emptyset$$

$$\begin{aligned}[\![\mathrm{P}, \textbf{if } e \textbf{ then goto } e_1 \textbf{ else goto } e_2]\!]_{\mathcal{I}} \mathrm{S} = &\{s' \mid \exists s \in \mathcal{C}[\![e]\!]([\![\mathrm{P}]\!]\mathrm{S}),\ s' = s[\mathtt{pc} \leftarrow \mathcal{E}[\![e_1]\!]s]\} \\ \cup &\{s' \mid \exists s \in \mathcal{C}[\![\neg e]\!]([\![\mathrm{P}]\!]\mathrm{S}),\ s' = s[\mathtt{pc} \leftarrow \mathcal{E}[\![e_2]\!]s]\}\end{aligned}$$

The distinguishing feature of machine code is that it is stored in memory like data. We assume the existence of a partial function decode $: \mathbb{S} \times \mathbb{A} \rightharpoonup$ *stmt* × *jump* that, given a state of the system and a memory address, decodes the instruction at that location, if it is well-formed.

A machine code executable can be considered to be a partial mapping from addresses to data, i.e., an element of $\mathbb{A} \rightharpoonup \mathbb{V}_1$. Before running an executable, its contents are stored in memory. This operation can be done by the operating system for user programs, or by a bootloader if the code runs directly on the hardware. Once the executable contents are in memory, its execution is entirely defined by the semantics of machine code. In the case of user programs running on an operating system, the execution is generally interleaved with the execution of the kernel and of other programs, but that does not affect the semantics of the program (except for input and output, which are an independent issue out of the scope of this thesis).

Definition 7.4 (Semantics of machine code). The semantics of machine code is given by the transition relation $\to \subseteq \mathbb{S} \times (\mathbb{S} \cup \{\Omega\})$ defined as:

$$s \to s' \iff \exists i \in \mathcal{I},\ \mathrm{decode}(s, s.\mathtt{pc}) = i \wedge s' \in [\![i]\!]_{\mathcal{I}}\{s\}$$

Definition 7.5 (Reachable states semantics in machine code). The set of states reachable from a set of initial states S_0 in machine code is denoted as $[\![\mathrm{S}_0]\!]_{\mathrm{bin}}$ and is:

$$[\![\mathrm{S}_0]\!]_{\mathrm{bin}} = \{s \in \mathbb{S} \mid \exists s_0 \in \mathrm{S}_0,\ s_0 \to^* s\}$$

where $\to^*$ is the transitive and reflexive closure of →.

7.2.2 Incremental inference of control flow in the presence of dynamic jumps

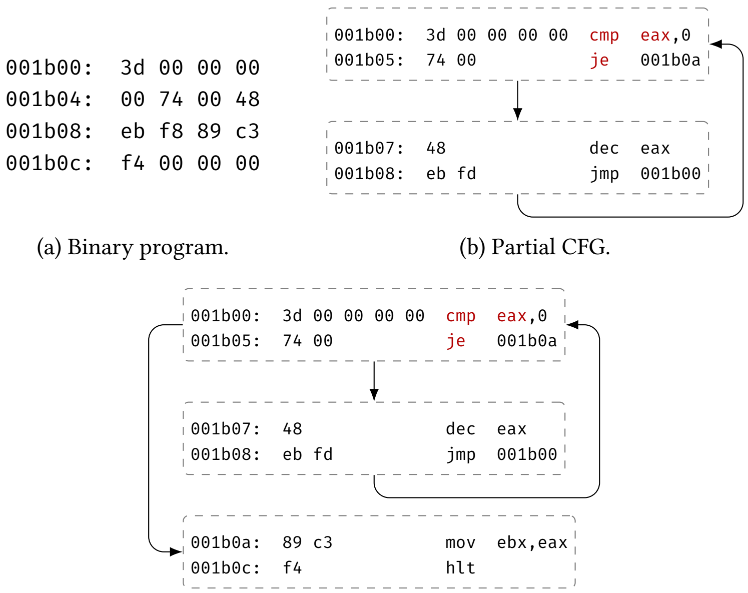
Control flow and data computation are strongly interdependent at the machine code level, since the targets of dynamic jumps depend on the contents of registers and memory. As a consequence, no control flow graph can be known in advance for machine code programs. We therefore designed an analysis to simultaneously compute abstractions of the CFG and of possible states at each program location.

For the simplicity of this presentation, we will for now consider that program locations are simply *code addresses*:

Definition 7.6 (Program locations (simple)). In machine code analysis, the set of program locations, denoted by $\mathcal{L}$, is the set of memory addresses:

$$\mathcal{L} = \mathbb{A}.$$

Our machine code analysis computes two objects: a CFG, and an abstract state of $\mathbb{F}^\sharp$ at each program location in that CFG.



```
001b00:  3d 00 00 00
001b04:  00 74 00 48
001b08:  eb f8 89 c3
001b0c:  f4 00 00 00
```

(a) Binary program.

```
001b00:  3d 00 00 00 00  cmp  eax,0
001b05:  74 00           je   001b0a

001b07:  48              dec  eax
001b08:  eb fd           jmp  001b00
```

(b) Partial CFG.

```
001b00:  3d 00 00 00 00  cmp  eax,0
001b05:  74 00           je   001b0a

001b07:  48              dec  eax
001b08:  eb fd           jmp  001b00

001b0a:  89 c3           mov  ebx,eax
001b0c:  f4              hlt
```

(c) Fixpoint CFG.

Figure 7.3: Incremental CFG inference.

Definition 7.7 (Control flow graph, and control flow invariant domain). Let $\mathbb{G}$ be the set of control flow graphs:

$$\mathbb{G} = \mathcal{P}\big(\mathcal{P}\big(\mathcal{L} \times (\mathbb{F}^\sharp \to \mathbb{F}^\sharp) \times \mathcal{L}\big)\big)$$

and $\mathbb{C}$ be the set of partial functions from program locations to abstract states:

$$\mathbb{C} = \mathcal{L} \rightharpoonup \mathbb{F}^\sharp.$$

An element of $\mathbb{C}$ represents a global invariant on the program. Our machine code analysis computes an element of $\mathbb{G} \times \mathbb{C}$. The set $\mathbb{G} \times \mathbb{C}$ is an abstraction of a set of states via the concretization:

$$\gamma_{\text{flow}}(\mathcal{G}, \xi) = \bigcup_{(\ell, \mathrm{F}^\sharp, \ell') \in \mathcal{G}} \{s \in \mathbb{S} \mid \exists \mathcal{v} : \mathbb{V}^\sharp \to \mathbb{V},\ (s, \mathcal{v}) \in \gamma_{\mathrm{E}}(\xi(\ell))\}$$

The computation of an invariant is a fixpoint computation, performed iteratively. Let us first describe it informally:

1. The analysis starts with an empty CFG.
2. If the CFG is empty, then the global invariant only maps ℓ_0 (the initial location) to $\mathbb{s}_0$ (the initial abstract state).
3. Otherwise, then a global mapping ξ, mapping each program location to an abstract state, is computed by chaotic iterations based on the current CFG $\mathcal{G}$.
4. At each program location ℓ, the abstract state $\xi(\ell)$ is used to compute the possible outgoing edges from ℓ. New discovered edges are added to the CFG.
5. Steps 3 and 4 are repeated until reaching a fixpoint for the CFG.

Consider the example binary program in Figure 7.3a, the initial location being `0x1b00` and the initial state being [`eax` ↦ {10}, `ebx` ↦ $[-2^{32},\ 2^{32}-1]$, `pc` ↦ {`0x1b00`}]. The inference of both control flow and reachable states proceeds as follows. The first instruction is decoded, it is a comparison between the value held by `eax` and zero (`cmp eax,0`). The successor instruction is a conditional jump (`je 001b0a`). If the comparison resulted in an equality, the next instruction is `0x1b0a`; otherwise, it is the successor of the jump instruction, in that case `0x1b07`. Here, since the equality is false, the analysis considers that the instruction has only one successor (`0x1b07`). The two next instructions are a decrementation of `eax` and a jump back to the initial location `0x1b00`.

After these first steps, the graph is as shown in Figure 7.3b. To improve readability, we group instruction into basic blocks and do not represent edges from one instruction to the next inside a basic block. At this point, the set of possible values for register `eax` at location `0x1b00` is computed as being in the interval $[0, 10]$. As a consequence, the conditional jump can now be taken and a new edge is added to the CFG with destination `0x1b0a`. Figure 7.3c shows the final graph. This graph is a fixpoint of the edge discovery process, since there are no edges left to add (the halting instruction `hlt` has no successor edges). Note that for simplicity's sake, this example is a small loop coded using a conditional jump, but the principle is the same for computed jumps, whose outgoing edges are determined by the value analysis.

We now formalize this process.

We first abstract the instruction-decoding function into $\mathrm{decode}^{\#} : \mathbb{F}^{\#} \times \mathcal{L} \to \mathcal{P}(\mathit{stmt} \times \mathit{jump}) \uplus \{\top\}$, such that $\mathrm{decode}^{\#}(\mathbb{s}, \ell)$ returns the decodings of ℓ in all the concrete states abstracted by $\mathbb{s}$; if in one state this decoding is undefined, then $\mathrm{decode}^{\#}$ returns $\top$:

$$\mathrm{decode}^{\#}(\mathbb{s}, \ell) = \begin{cases} \top & \text{if } \exists (s, v) \in \gamma_{\mathrm{F}}(\mathbb{s}),\ \mathrm{decode}(s, \ell) \text{ is undefined} \\ \{\mathrm{decode}(s, \ell) \mid (s, v) \in \gamma_{\mathrm{F}}(\mathbb{s})\} & \text{otherwise} \end{cases}$$

Finally, the operator $\mathfrak{jumps} : \mathbb{F}^{\#} \times \mathit{stmt} \times \mathit{jump} \to \mathcal{P}\big((\mathbb{F}^{\#} \to \mathbb{F}^{\#}) \times \mathcal{L}\big)$ computes the possible outgoing edges from an instruction, given an abstract state:

$$\mathfrak{jumps}(\mathbb{s}, \mathrm{P}, \mathbf{stop}) = \emptyset$$

$$\begin{aligned} \mathfrak{jumps}(\mathbb{s}, \mathrm{P}, \mathbf{if}\ e\ \mathbf{then\ goto}\ e_1\ \mathbf{else\ goto}\ e_2) = {} & \{(\mathrm{F}^{\#}, \ell) \mid \mathrm{F}^{\#} \overset{\text{def}}{=} \mathbf{guard}_{\mathrm{F}} \circ \mathcal{E}[\![e]\!]_{\mathrm{F}}^{\#} \circ [\![\mathrm{P}]\!]_{\mathrm{F}}^{\#},\ \ell \in \gamma_{\mathrm{F}}^{\mathrm{V}}(\mathcal{E}[\![e_1]\!]^{\#}(\mathrm{F}^{\#}(\mathbb{s})))\} \\ & \cup \{(\mathrm{F}^{\#}, \ell) \mid \mathrm{F}^{\#} \overset{\text{def}}{=} \mathbf{guard}_{\mathrm{F}} \circ \mathcal{E}[\![\neg e]\!]_{\mathrm{F}}^{\#} \circ [\![\mathrm{P}]\!]_{\mathrm{F}}^{\#},\ \ell \in \gamma_{\mathrm{F}}^{\mathrm{V}}(\mathcal{E}[\![e_2]\!]^{\#}(\mathrm{F}^{\#}(\mathbb{s})))\} \end{aligned}$$

Then, given a CFG $\mathcal{G}$ and a function $\xi \in \mathbb{C}$ mapping every node ℓ in $\mathcal{G}$ to an abstract state, the set of outgoing edges from ℓ is:

$$\{(\ell, \mathrm{F}^{\#}, \ell') \mid \exists (\mathrm{P}, \mathrm{J}) \in \mathrm{decode}^{\#}(\xi(\ell), \ell),\ (\mathrm{F}^{\#}, \ell') \in \mathfrak{jumps}(\xi(\ell), \mathrm{P}, \mathrm{J})\}$$

All edges that were not present in $\mathcal{G}$ are added. If for any $\ell \in \mathrm{dom}(\xi)$, the result of $\mathrm{decode}^{\#}(\xi(\ell), \ell)$ is $\top$, then the analysis emits an error: the analyzed machine code may fail attempting to decode an ill-formed instruction. This may also be caused by an imprecision in the analysis.

We are now able to define the abstract operator that grows the abstract CFG (steps 3 and 4). As stated above (Section 7.1.1), step 3, namely inferring a sound abstract state at each program location given a CFG, is standard. We denote as $\mathfrak{analyze} : \mathbb{G} \times \mathbb{F}^{\#} \to \mathbb{C}$ the function realizing this operation from a CFG and an initial state.

Definition 7.8 (CFG iteration operator). The operator $\mathcal{F} : \mathbb{G} \times \mathbb{C} \to \mathbb{G} \times \mathbb{C}$ is defined as follows:

- if there exists $\ell \in \mathrm{dom}(\xi)$ such that $\mathrm{decode}^{\#}(\xi(\ell), \ell) = \top$, then the machine code may fail due to an ill-formed instruction (also called "invalid opcode" error); the new CFG and state invariant are completely imprecise:
$$\mathcal{F}(\mathcal{G}, \xi) = (\top_{\mathbb{G}}, \top_{\mathbb{C}})$$

(In practice, the analysis emits an alarm indicating a possibly invalid opcode and stops.)

- Otherwise,

$$\mathcal{F}(\mathcal{G}, \xi) = (\mathcal{G}', \xi')$$

where

$$\begin{aligned}\xi' &= \mathfrak{analyze}(\mathcal{G}, \xi(\ell_0))\\ \mathcal{G}' &= \mathcal{G} \cup \{(\ell, \mathrm{F}^\#, \ell') \mid \exists(\mathrm{P}, \mathrm{J}) \in \mathrm{decode}^\#(\xi'(\ell), \ell),\ (\mathrm{F}^\#, \ell') \in \mathfrak{jumps}(\xi'(\ell), \mathrm{P}, \mathrm{J})\}\end{aligned}$$

Definition 7.9 (Abstract semantics of the control flow domain). Given an abstract state $\mathbb{s}_0 \in \mathbb{F}^\#$, the abstract semantics of machine code starting from $\mathbb{s}_0$ in $\mathbb{G} \times \mathbb{C}$, denoted as $[\![\mathbb{s}_0]\!]^\#_{\mathrm{bin}}$, is the fixpoint of operator $\mathcal{F}$, starting from the element $(\emptyset,\ [\ell_0 \mapsto \mathbb{s}_0])$. The order on the elements of $\mathbb{G} \times \mathbb{C}$ is defined as:

$$(\mathcal{G}_1, \xi_1) \sqsubseteq_{\mathrm{flow}} (\mathcal{G}_2, \xi_2) \iff \mathcal{G}_1 \subseteq \mathcal{G}_2$$

Note that the CFG thus inferred has at most one edge between two locations.

This approach is sound, in the sense that it infers a CFG that contains all possible reachable states:

Theorem 7.1 (Soundness of $[\![\cdot]\!]^\#_{\mathrm{bin}}$). *Given* $\mathbb{s}_0 \in \mathbb{F}^\#$*:*

$$[\![\gamma_{\mathrm{F}}(\mathbb{s}_0)]\!]_{\mathrm{bin}} \subseteq \gamma_{\mathrm{F}}([\![\mathbb{s}_0]\!]^\#_{\mathrm{bin}})$$

Proof sketch. This fact can be proved *ab absurdum*: suppose that there exists a concrete execution trace, i.e., a sequence of CFG edges, that is not contained in the CFG $\mathcal{G}$ resulting from the fixpoint computation described above, and consider the first edge of the trace not in $\mathcal{G}$. This edge is necessarily in the result of $\mathfrak{jumps}$. Indeed, it can be shown that $\mathfrak{jumps}$ always returns a superset of the possible outgoing edges in a given (partial) CFG, since the abstract semantics of the state domain $\mathbb{F}^\#$ is sound. Therefore $\mathcal{G}$ is not a fixpoint, which contradicts our hypothesis. □

The CFG computation always terminates because the set of program locations is finite (and there is at most one edge between two program locations), but it may take a very long time. However, we did not find it necessary to design convergence acceleration techniques for the CFG inference, since in our experiments reaching the control flow fixpoint never was a performance bottleneck of the analysis.

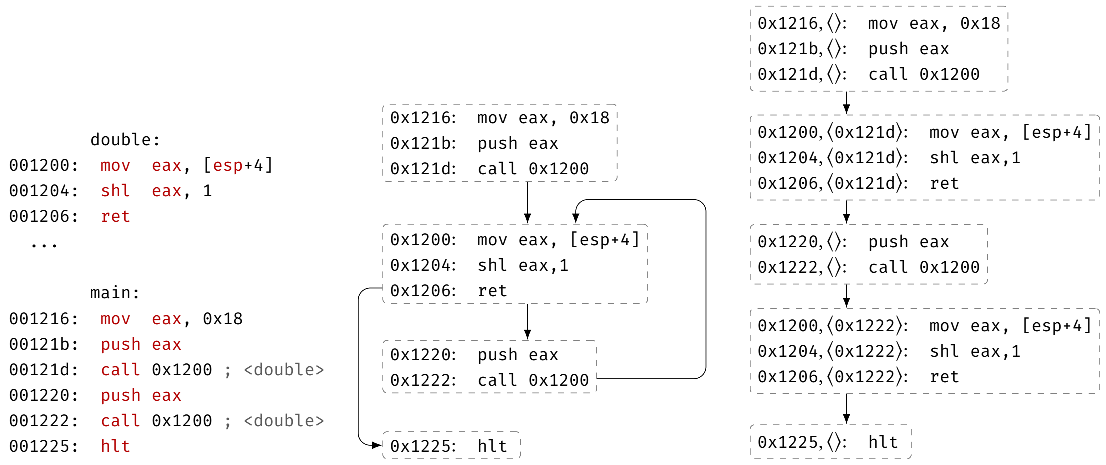
7.2.3 Delineation of functions

Another difficulty specific to machine code is the absence of a well-defined notion of function: function calls are nothing more than control flow jumps[1].

Yet, detecting function calls is crucial to perform a context-sensitive analysis, i.e., an analysis that analyzes the code of functions in their calling context. We perform a context-sensitive analysis for precision reasons. Consider the function `double` in Figure 7.4a which returns its arguments multiplied by 2 and is called twice by the `main` function. If the analysis is not context-sensitive, then the CFG resembles that of a loop (Figure 7.4b). As a consequence, our analysis will merge the possible stacks, and, since the argument of `double` is pushed onto the stack, the analysis will be imprecise. On the contrary, if each call is analyzed in context (Figure 7.4c), the analysis will be more precise on the function argument, and thus on the program as a whole.

We therefore enrich program locations with a *call stack* (see Figure 7.4c), represented as a finite sequence of code addresses:

[1] x86 assembly has dedicated `call` and `ret` instructions, but there is no guarantee that functions calls and function returns will not be compiled into other jump instructions; and in fact it is often not the case due to call stack optimizations.



```
        double:
001200:  mov  eax, [esp+4]
001204:  shl  eax, 1
001206:  ret
  ...

        main:
001216:  mov  eax, 0x18
00121b:  push eax
00121d:  call 0x1200 ; <double>
001220:  push eax
001222:  call 0x1200 ; <double>
001225:  hlt
```

(a) Assembly program.

```
0x1216:  mov eax, 0x18
0x121b:  push eax
0x121d:  call 0x1200

0x1200:  mov eax, [esp+4]
0x1204:  shl eax,1
0x1206:  ret

0x1220:  push eax
0x1222:  call 0x1200

0x1225:  hlt
```

(b) CFG without call stacks: the two function calls are indistinguishable from a loop.

```
0x1216,⟨⟩:  mov eax, 0x18
0x121b,⟨⟩:  push eax
0x121d,⟨⟩:  call 0x1200

0x1200,⟨0x121d⟩:  mov eax, [esp+4]
0x1204,⟨0x121d⟩:  shl eax,1
0x1206,⟨0x121d⟩:  ret

0x1220,⟨⟩:  push eax
0x1222,⟨⟩:  call 0x1200

0x1200,⟨0x1222⟩:  mov eax, [esp+4]
0x1204,⟨0x1222⟩:  shl eax,1
0x1206,⟨0x1222⟩:  ret

0x1225,⟨⟩:  hlt
```

(c) CFG with call stacks: the two function calls are identified and can be analyzed in context.

Figure 7.4: Illustration of the inferred CFGs with and without including a call stack in program locations.

Definition 7.10 (Context-sensitive program locations). In machine code analysis, a *program location* is a pair consisting of:

- a *code address* in $\mathbb{A}$;
- and a *call stack*, i.e., a finite sequence of code addresses.

The set of program locations is denoted by $\mathcal{L}$:

$$\mathcal{L} = \mathbb{A} \times \mathbb{A}^*.$$

During the analysis, a function call causes the calling address to be added to the code stack; on the contrary, returning from a function causes the topmost element of the call stack to be popped. This enables the analysis to be *fully context-sensitive.*

The question remains of how function calls and returns are identified. When compiled to assembly code, function calls are translated as jumps that are not easy to distinguish from e.g., jumps in a loop.

We resort to a simple, yet efficient heuristic: function boundaries are identified using the symbol table present in the executable, if any. We model it by a set of function names $\mathcal{N}_{\text{fun}}$ and a function $\text{sym} : \mathbb{A} \rightharpoonup \mathcal{N}_{\text{fun}}$ representing the symbol associated to a code address.

Definition 7.11 (Machine code semantics with call stacks). The transition relation $\rightarrow' \subseteq (\mathbb{S}\times\mathcal{L})\times(\mathbb{S}\times\mathcal{L})$ is defined as follows. Given $s, s' \in \mathbb{S}$, given $(\rho, \sigma), (\rho', \sigma') \in \mathcal{L}$, the relation $(s, (\rho, \sigma)) \rightarrow' (s', (\rho', \sigma'))$ holds if and only if $s \rightarrow s'$ and $\rho = s.\text{pc}$ and $\rho' = s'.\text{pc}$, and one of the following holds:

- $\text{sym}(\rho) = \text{sym}(\rho')$ and $\sigma = \sigma'$ (the destination location has the same call stack as the source location).
- $\text{sym}(\rho) \neq \text{sym}(\rho')$ and $\sigma = a_1, a_2, \dots, a_k, \rho, a_{k+1}, \dots$ with $a_1, \dots, a_k \neq \rho'$, and $\sigma' = \rho, a_{k+1}, \dots$, i.e., if the destination address can be found in the call stack of the source location, then the jump is considered to be a return from one or more functions[2], and the corresponding addresses are popped from the call stack.

[2] Due to compiler optimizations, one instruction may return from several instructions at once (sibling call optimization).

- $\mathrm{sym}(\rho) \neq \mathrm{sym}(\rho')$, and $\rho' \notin \sigma$ and $\sigma' = \rho, \sigma$, i.e., the jump is considered to be a function call, and the calling address is added to the call stack.

The semantics $\rightarrow'$ corresponds to $\rightarrow$ with all functions (in the sense of the symbol table) inlined. Indeed, the two semantics are identical up to call stacks:

Theorem 7.2. *Given $s, s' \in \mathbb{S}$:*

$$s \rightarrow s' \iff \exists \sigma, \sigma',\ (s, (s.\mathsf{pc}, \sigma)) \rightarrow' (s', (s'.\mathsf{pc}, \sigma'))$$

Proof. The implication from right to left is true by definition. To prove the converse implication, take any $\sigma \in \mathbb{A}^*$. Depending on whether $\mathrm{sym}(s.\mathsf{pc})$ equals $\mathrm{sym}(s'.\mathsf{pc})$ or not, it is easy to construct σ' such that $(s, (\rho, \sigma)) \rightarrow' (s', (\rho', \sigma')$. □

The CFG construction described in Section 7.2.2 does not change: the $\mathfrak{jumps}$ function is easily adapted to work with call stacks. The analysis still terminates as the call stack size remains bounded (since we assume the absence of recursive calls).

While symbol tables are not guaranteed to be correctly generated by the compiler, our experiments find that this method is effective at delineating function contexts (Section 7.3 and Chapter 10). Note also that this heuristic influences only the precision of the analysis, and not its soundness. In the absence of a symbol table, another way to delineate functions may have to be devised if the precision is not satisfactory, possibly via manual annotations or automated heuristic [ASB17].

7.2.4 Product with an “array of bytes” memory abstraction

In machine code, unlike in C code, the concept of *variables* is absent: except for data that is stored in registers, every data access involves dereferencing an address. For example, C compilers typically translate accesses to local variables as memory accesses to the stack, the stack being simply a region in the program address space; and global variables are stored in another such memory region.

The principle behind the type-based shape domain is to abstract the memory by a type invariant. However, this abstraction is not well-suited to represent the stack and global variables, for the following reasons: firstly, on the stack, one memory location will typically hold various, unrelated values over time; these values cannot be precisely abstracted by one type. Secondly, our type-based abstraction is flow-insensitive, whereas a precise analysis of local variables require a flow-sensitive representation of the stack.

Finally, the stack and global variables are stored in a bounded, statically-known memory region, and therefore practical to represent by a non-summarizing abstraction. For all these reasons, we use a different abstraction for the stack and global variables than the type-based abstraction used for the rest of the heap. This abstraction represents memory as an “array of bytes”, that is, it fully enumerates all memory cells.

We denote as $\mathbb{A}_\mathrm{F}$ the set of addresses holding the stack and global variables, and $\mathbb{A}_\mathrm{T} = \mathbb{A} \setminus \mathbb{A}_\mathrm{F}$. (F stands for “fixed” and T stands for “typed”.)

In addition, we modify the definitions of physical types so that only $\mathbb{A}_\mathrm{T}$ is labelled with types, and pointer types to point into $\mathbb{A}_\mathrm{T}$. Table 7.1 shows the modified definitions.

The fully enumerative memory abstraction (also called “flat model” abstraction) is defined thus:

Definition 7.12 (“Flat model” memory domain). The abstract domain $\mathbb{M}^\sharp$ is defined as:

$$\mathbb{M}^\sharp = \mathbb{A}_\mathrm{F} \rightarrow \mathbb{V}_\mathrm{F}$$

Object defined	New definition
Labellings (Definition 4.2)	$\mathbb{L} = \mathbb{A}_\mathrm{T} \to \mathbb{T}_\mathrm{A}$
Addresses covered by a type (Definition 4.7)	$\mathrm{addr}_\mathcal{L}(t) = \bigcup_{i=0}^{\mathrm{size}(t)-1} \{a \in \mathbb{A}_\mathrm{T} \mid \mathcal{L}(a) \preceq t.(i)\}$
Interpretation of pointer types (Definition 4.8)	$(\!\| t.(k)* \|\!)_{\mathcal{L},\mathcal{v}} = \{0\} \cup \{a \in \mathbb{A}_\mathrm{T} \mid \mathcal{L}(a) \preceq t.(k)\}$
Well-typed state (Definition 4.10)	$\forall a \in \mathbb{A}_\mathrm{T}, (\exists t \in \mathbb{T}, \mathcal{L}(a) = t.(0))$ $\implies \quad h[a..a+\mathrm{size}(t)] \in (\!\| t \|\!)_{\mathcal{L},\mathcal{v}}$

Table 7.1: Definitions modified in order to restrict the type-based abstraction of memory to $\mathbb{A}_\mathrm{T}$ (modifications shown in red).

with concretization:

$$\begin{aligned} \gamma_\mathrm{M} &: \mathbb{M}^\sharp \to \mathcal{P}\big((\mathbb{A}_\mathrm{F} \to \mathbb{V}_1) \times (\mathbb{V}^\sharp \to \mathbb{V})\big) \\ \gamma_\mathrm{M}(\mathbb{h}_\mathrm{F}) &= \{(h_\mathrm{F}, \mathcal{v}) \mid \forall a \in \mathbb{A}_\mathrm{F},\ h_\mathrm{F}(a) = \mathcal{v}(\mathbb{h}_\mathrm{F}(a))\} \end{aligned}$$

This abstraction is finite, since $\mathbb{A}_\mathrm{F}$ is. In practice, since representing each memory byte separately is inefficient, it uses a stratified representation of memory caching multi-bytes loads and stores, like in [Min06]; and it does not track memory cells whose contents are not constrained by any predicate.

Definition 7.13 (Combined shape–fixed abstract domain). The combined shape–fixed abstract domain $\mathbb{E}^\sharp$ is defined as:

$$\mathbb{E}^\sharp = \mathbb{F}^\sharp \times \mathbb{M}^\sharp$$

with concretization:

$$\begin{aligned} \gamma_\mathrm{E} &: \mathbb{E}^\sharp \to \mathcal{P}\big(\mathbb{S}_t \times (\mathbb{V}^\sharp \to \mathbb{V})\big) \\ \gamma_\mathrm{E}(\mathbb{f}, \mathbb{m}) &= \left\{ ((\sigma, \Gamma, h, \mathcal{L}),\ \mathcal{v}) \;\middle|\; \begin{array}{l} \exists h_\mathrm{T} \in \mathbb{H},\ ((\sigma, \Gamma, h_\mathrm{T}, \mathcal{L}), \mathcal{v}) \in \gamma_\mathrm{F}(\mathbb{f}), \\ \exists h_\mathrm{F} \in \mathbb{H},\ (h_\mathrm{F}, \mathcal{v}) \in \gamma_\mathrm{M}(\mathbb{m}), \\ \forall a \in \mathbb{A},\ h(a) = \begin{cases} h_\mathrm{F}(a) & \text{if } a \in \mathbb{A}_\mathrm{F} \\ h_\mathrm{T}(a) & \text{otherwise} \end{cases} \end{array} \right\} \end{aligned}$$

The set of *abstract values* of $\mathbb{E}^\sharp$ (see Section 3.4.1) is the same as the set of abstract values of $\mathbb{F}^\sharp$:

$$\mathbb{V}_\mathrm{E} = \mathbb{V}_\mathrm{F} = \mathbb{V}_\mathrm{S} = \mathbb{V}^\sharp \times \mathbb{T}^\sharp$$

We do not give the full detail of the transfer functions of $\mathbb{E}^\sharp$; they are mostly trivial, except for the fact that the abstract semantics of memory reads and writes use the semantic operators of either $\mathbb{F}^\sharp$ or $\mathbb{M}^\sharp$, depending on which part of the heap is affected.

For example, storing the abstract value $(\mathbb{v}, \mathbb{t})$ at address $(\mathbb{a}, \mathbb{t}_a)$ proceeds as follows:

- if $\mathbb{t}_a$ is a pointer type, then the memory write is performed using the abstract memory write operator of domain $\mathbb{F}^\sharp$.
- Otherwise, if $\mathbb{a} \in \mathbb{A}_\mathrm{F}$, then the write is performed using the abstract memory write operator of domain $\mathbb{M}^\sharp$.
- Otherwise, the resulting abstract state is completely imprecise (or, in practice, the analysis emits an alarm).

7.2.5 Numerical abstraction

The numerical abstract domain that we use in our analyses (i.e., the instance of $\mathbb{D}_{\text{num}}$ that we use) consists mainly in non-relational intervals with congruence information (sometimes called strided intervals [Bal07]), in product with a *bitwise* abstraction, i.e., an abstraction that attempts to retain the effect of bit-level operations by representing a value as a sequence of bits, where each bit is abstracted by an element of $\{0, 1, ?\}$, where "?" means "unknown".

The abstraction we use is in fact able to retain a few relational predicates: for any two symbolic variables $\mathbbm{v}_1$ and $\mathbbm{v}_2$, it is able to retain the predicate $\mathbbm{v}_1 \leq \mathbbm{v}_2$ (recall that symbolic variables represent immutable values). This is useful for bound checking on arrays whose length is symbolic. However, it is not able to derive new inequalities, such as $\mathbbm{v}_1 + 1 < \mathbbm{v}_2 + 1$.

The abstraction of pointer offsets is slightly richer: a pointer offset is represented as a triple $(i, f, n) \in \mathbb{V}^\# \times \mathbb{V}^\# \times \mathbb{N}$, where i represents the index in the pointed array, f represents the same index, but *counting from the end* of the array, and n represents the offset in the array element. The concrete offset o represented by this triple is such that these two equations hold:

$$\begin{cases} o = i \cdot \text{size}(t) + n \\ f = i - s \end{cases}$$

for an offset in an array of type $t[s]$. Non-array types are viewed as arrays of length 1, in which case $i = 0$ and $f = -1$.

This representation allows to check array bounds, even using non-relational numerical abstractions: a pointer offset is in the bounds of the array if $i \geq 0$ and $f < 0$. This abstraction was sufficiently precise on our use cases (see Section 7.3 and Chapter 10) that we did not need to use a fully relational numerical abstract domain.

7.3 Experimental evaluation

We evaluated the efficiency and the precision of our analysis, and the parametrization effort, by analyzing a set of benchmark programs manipulating various data structures, some including complex sharing patterns. We perform these analyses both on C source code and on compiled executables. Those structures include singly- and doubly-linked lists, AVL and red-black trees, directed graphs, and splay trees. These experiments were presented, along with our type-based shape domain and the staged and retained points-to predicates, in an article published at the VMCAI 2022 conference [NLR22].

These experiments evaluate the ability of our abstractions to verify *spatial memory safety* and *type-based structural invariants.* While spatial memory safety is clearly a desirable property, the usefulness of verifying type-based structural invariants has not been established yet. This will be the object of our kernel verification experiments, described at length in Part II.

7.3.1 Research questions

RQ1 (Is our analysis is effective?) *Is the type-based shape domain with retained and staged points-to predicates effective to analyze real-world data-structure libraries?* Our criterion will be the number of programs analyzed without false alarms. An analysis without alarms means that we have successfully verified 1. *spatial memory safety*, i.e., that all memory accesses are performed within the bounds of allocated objects; and 2. *preservation of structural invariants* as defined by the well-typedness property of states (Definition 4.10). While spatial memory safety is a property that can be defined independently from physical types, structural invariants are relative to a *set of types* (more precisely type mapping, see Section 4.2) which is *a parameter of the analysis.*

RQ2 (Are retained and staged points-to predicates necessary?) *Are retained and staged points-to predicates necessary to keep sufficient precision?* We evaluate their usefulness by running the analysis with and without points-to predicates. Our implementation reunites retained and staged points-to predicates in a single abstract domain. For this reason, they are either both enabled or both disabled.

RQ3 (Is our analysis efficient?) *Does our type-based shape analysis scale well in practice?* In particular, does it scale better than shape analyses based on separation logic, which express stronger invariants at the cost of handling disjunctions?

RQ4 (Is the annotation effort low?) *What is the amount of manual annotation required to perform a successful analysis?* What we call *annotations* in the context of this analysis is the *manually written definitions of physical types*. Since in practice for C programs, the physical types are a refinement of the C types, an initial set of types can be automatically generated from the C source. We implemented this automated generation. However, the structural invariants entailed by the automatically generated set of types may not be strong enough for the analysis to terminate without alarms; in that case, the types must be *refined* by e.g., adding some refinement predicates. For this reason, in our evaluation we distinguish between the number of lines of annotations *generated* and the number of lines that *added or modified* for the analysis to emit fewer or no false alarms.

7.3.2 Methodology

We ran our C analyzer, Frama-C/Codex, on all the C benchmarks from two shape analysis publications; moreover, we compiled them using GCC 10.3.0 with different levels of optimizations, namely `-O0`, `-O1`, `-O2`, and `-O3`, and analyzed the resulting executables with Binsec/Codex.

These benchmarks are challenging: the `graph-*` benchmarks from Li et al. [LRC15] were used to verify unstructured sharing patterns; to complete this evaluation we add the functions `uf_find`, `merge` and `make` from our running example of Figure 4.1 (benchmarks `uf-*`). The other benchmarks are from Li et al. [Li+17] and were used to demonstrate scalability issues faced by shape analyzers. Both sets of benchmarks were created to demonstrate shape analysis, which is a more precise abstraction than the one we propose. They are thus suitable to evaluate ability to handle complex sharing patterns and precision. We also use them to compare how our approach scales compared to separation logic-based shape analysis, either standard or improved with the semantic-directed clumping from Li et al. [Li+17].

All analyses have been conducted on a standard laptop Intel Xeon E3-1505M 3 Ghz running Linux 5.10.81, with a 32 GB RAM. We took the mean values between 10 runs, and report the mean (all standard deviations were below 4%).

7.3.3 Results

Table 7.2 provides the results of the evaluation. Each benchmark consists in the analysis of one function, possibly calling other functions. The benchmarks are grouped by the data structure they operate on; we report the number of lines describing physical types (**gen**erated from existing types information, or manually **ed**ited) shared by a group. Finally, the **pre.** column gives the number of lines of **pre**-conditions for the verified function (e.g., that a pointer argument must not be null). The **pre.** annotations consist in providing the types of the arguments (if any) of the function analyzed. The **LOC** column is the number of lines of code of each function, excluding comments, blank lines and subroutines. The ratio of lines of manual annotations per line of code for a group, goes from 0% to 7.8%, with a mean of 3.2% and median of 2.7%.

The next columns in the table provide the time taken by the full analysis (in s), the number of alarms of the full analysis (↦ column) and the analysis without the retained and staged points-to predicates ($\not\mapsto$

Benchmark	Annotations			LOC	C			O0			O1			O2			O3		
	gen./ed./pre.				Time / ↦ / ↦̸			Time/↦/↦̸			Time/↦/↦̸			Time/↦/↦̸			Time/↦/↦̸		
sll-delmin	11	0	1	25	0.27	0	0	0.13	0	0	0.15	0	0	0.15	0	0	0.13	0	0
sll-delminmax			1	49	0.30	0	0	0.19	0	0	0.17	0	0	0.17	0	0	0.16	0	0
psll-bsort	10	0	0	25	0.30	0	22	0.41	0	3	0.25	0	3	0.26	0	3	0.29	0	3
psll-reverse			0	11	0.28	0	2	0.10	0	1	0.13	0	1	0.10	0	1	0.10	0	1
psll-isort			0	20	0.29	0	2	0.34	0	1	0.34	0	1	0.32	0	1	0.33	0	1
bstree-find	12	0	1	26	0.27	0	0	0.14	0	0	0.13	0	0	0.15	0	0	0.16	0	0
gdll-findmin	25	5	1	49	0.50	0	0	0.41	0	0	0.39	0	0	0.41	0	0	0.42	0	0
gdll-findmax			1	58	0.55	0	0	0.33	0	0	0.22	0	0	0.21	0	0	0.20	0	0
gdll-find			1	78	0.56	0	0	0.15	0	0	0.15	0	0	0.14	0	0	0.14	0	0
gdll-index			1	55	0.53	0	0	0.32	0	0	0.33	0	0	0.30	0	0	0.29	0	0
gdll-delete			1	107	0.57	0	2	0.16	0	0	0.14	0	0	0.13	0	0	0.13	0	0
javl-find	45	12	2	25	0.35	0	0	0.23	0	0	0.28	0	0	0.18	0	0	0.19	0	0
javl-free			1	27	0.35	0	4	0.11	0	3	0.12	0	0	0.10	0	0	0.11	0	0
javl-insert			2	95	0.53	6	56	0.52	12	20	0.39	30	34	0.43	29	34	0.43	29	34
javl-insert-32×			2	95	16.68	192	1792	28.28	14	20	33.14	34	34	32.00	32	34	40.01	32	34
gbstree-find	23	5	1	53	0.58	0	0	0.38	0	0	0.40	0	0	0.56	0	0	0.59	0	0
gbstree-delete			1	165	0.81	0	0	0.90	0	0	0.72	0	0	0.67	0	0	0.66	0	0
gbstree-insert			1	133	0.55	0	7	0.26	0	0	0.21	0	0	0.23	0	0	0.24	0	0
brbtree-find	24	3	2	29	0.32	0	0	0.17	0	0	0.19	0	0	0.23	0	0	0.23	0	0
brbtree-delete			2	329	0.79	103	127	1.15	44	73	1.23	46	53	0.85	58	63	0.84	58	63
brbtree-insert			2	177	0.61	24	47	0.90	11	23	0.47	16	17	1.22	21	17	0.97	21	17
bsplay-find	22	1	1	81	0.53	0	18	0.25	0	7	0.23	0	7	0.23	0	7	0.23	0	7
bsplay-delete			1	95	0.72	0	38	0.45	0	11	0.44	0	10	0.44	0	10	0.44	0	10
bsplay-insert			1	101	0.57	0	18	0.25	0	7	0.25	0	7	0.25	0	7	0.25	0	7
graph-nodelisttrav	23	0	1	12	0.20	0	0	0.10	0	0	0.10	0	0	0.10	0	0	0.11	0	0
graph-path			1	19	0.21	0	14	0.15	0	5	0.16	0	0	0.14	0	0	0.16	0	0
graph-pathrand			1	25	0.22	0	10	0.13	0	0	0.21	0	0	0.12	0	0	0.11	0	0
graph-edgeadd			1	15	0.27	0	2	0.12	0	1	0.11	0	1	0.10	0	1	0.10	0	1
graph-nodeadd			1	15	0.26	0	2	0.10	0	1	0.08	0	1	0.09	0	1	0.10	0	1
graph-edgedelete			1	11	0.20	0	2	0.10	0	1	0.10	0	0	0.10	0	0	0.11	0	0
graph-edgeiter			1	22	0.23	0	0	0.13	0	0	0.11	0	0	0.12	0	0	0.12	0	0
uf-find	33	3	1	11	0.31	0	24	0.07	0	6	0.09	0	0	0.08	0	0	0.07	0	0
uf-merge			1	17	0.34	0	50	0.13	0	7	0.18	0	0	0.18	0	0	0.15	0	0
uf-make			0	9	0.31	0	4	0.05	0	3	0.06	0	3	0.07	0	3	0.06	0	3
Total verified						30	13		30	16		30	21		30	21		30	21

Table 7.2: Results of the evaluation

column), for the C version of the code and the various binaries produced by GCC. For brevity, we have omitted the time taken by the $\not\mapsto$ analysis in the benchmarks; on average this analysis takes 1.5% less time for the C, and 20% less for binary code (maximum: 45%). The number of alarms is counted differently in C (one possible alarm each time the analyzer evaluates a statement) and in binary (where each alarm type is counted at most once per instruction), but in both, 0 alarms means that the analyzer verified type-safety. We observe that the full analyzer succeeds in verifying 30 benchmarks (out of 34), both in C and binary code. Removing the points-to predicates makes the analysis significantly less precise, as only 13 benchmarks are verified in C, and between 16 (for `-O0`) and 21 (for `-O1`,`-O2`,`-O3`) in binary code.

Annotation example

The C type declarations for the `javl-*` set of benchmarks are as follows:

```
typedef struct jsw_avlnode {
  int  balance;               /* Balance factor */
  int *data;                  /* User-defined content */
  struct jsw_avlnode *link[2]; /* Left (0) and right (1) links */
} jsw_avlnode_t;

typedef struct jsw_avltree {
  jsw_avlnode_t* root;   /* Top of the tree */
  size_t         size;   /* Number of items (user-defined) */
} jsw_avltree_t;

typedef struct jsw_avltrav {
  jsw_avltree_t *tree;     /* Paired tree */
  jsw_avlnode_t *it;       /* Current node */
  jsw_avlnode_t *path[64]; /* Traversal path */
  size_t         top;      /* Top of stack */
} jsw_avltrav_t;
```

The mapping $\mathcal{M}$ from type names to physical types automatically generated from those type declarations is the following:

$$\mathcal{M} = \begin{bmatrix} \texttt{int} \mapsto \texttt{word}_4 \\ \texttt{avlnode} \mapsto \texttt{int} \times \texttt{int}.(0)\ast \times \big(\texttt{avlnode}.(0)\ast\big)[2] \\ \texttt{avltree} \mapsto \texttt{avlnode}.(0)\ast \times \texttt{int} \\ \texttt{avltrav} \mapsto \texttt{avltree}.(0)\ast \times \texttt{avlnode}.(0)\ast \times \big(\texttt{avlnode}.(0)\ast\big)[64] \times \texttt{int} \end{bmatrix}$$

We modified it in the following way: we created an additional type name `node_data`, bound to type `int`:

$$\mathcal{M} : \texttt{node\_data} \mapsto \texttt{int}$$

and replaced $\texttt{int}.(0)\ast$ in `avlnode` with type $\{x : \texttt{node\_data}.(0)\ast \mid x \neq 0\}$:

$$\mathcal{M} : \texttt{avlnode} \mapsto \texttt{int} \times \{x : \texttt{node\_data}.(0)\ast \mid x \neq 0\} \times \big(\texttt{avlnode}.(0)\ast\big)[2]$$

to improve precision. Indeed, this way, writing to an address of type $\texttt{node\_data}.(0)\ast$ can only modify the data field of an AVL node; whereas writing to an address of type $\texttt{int}.(0)$ potentially modifies regions of type `avlnode` and `avltree`, because both have a field of type `int`. This happened during the analysis and caused some retained points-to predicates to be dropped, resulting in imprecisions later in the analysis. Introducing the type `node_data` fixed that problem.

Note that the modification we just described corresponds to the 12 **ed**ited lines for the `javl-*` benchmarks (Table 7.2). (Our type description language is more verbose than the notations used in this thesis, especially to define new types such as `node_data`.)

7.3.4 Discussion and conclusions

RQ1 (Is our analysis effective?) Our combination of domains is effective at verifying preservation of structural invariants—i.e., the invariants expressed by our type-based abstract domain (which entail spatial memory safety), distinct from the stronger invariants that define some data structures like linked lists, trees, etc.— on C and binary code, even for benchmarks that have complex sharing patterns, with a low amount of annotations.

RQ2 (Are retained and staged points-to predicates necessary?) The points-to predicates are very important for precision, as otherwise the number of false alarms increases significantly. The analysis with points-to predicates runs just as well on binary programs and on C programs, despite the complex code patterns that the C compiler may produce. Note that without points-to predicates, more binary codes are verified than in C: indeed, in some cases the compiler performs a register promotion of a heap value, which removes the need for a points-to predicate.

RQ3 (Is our analysis efficient?) The analysis performs evenly well on all benchmarks, and scales well on `javl-insert-32×`, which consists in an AVL tree insertion composed 32 times with itself. This benchmark produces a CFG 32 times as large as `javl-insert`, and is challenging for shape analyses due to the increasingly complex abstract states [Li+17]. On this benchmark, our analysis, without any adaptations, scales comparably to the shape analysis with guided clumping (a sophisticated technique to keep the size of the abstract state under control) described in [Li+17]. This suggests that type-based invariants allow to keep predictable analysis times. Note that, although the analysis is not precise enough to verify `javl-insert-32×`, it can still be used to measure performance: the emission of false alarms is independent from the fact that the size of the abstract points-to map is bounded by the number of memory accesses, and therefore so is the complexity of the abstract operators of $\mathbb{R}^\sharp$ and $\mathbb{S}\mathbb{t}^\sharp$.

RQ4 (Is the annotation effort low?) The annotation effort varies between 0 and 12 lines of modifications (on average 3.2 lines) to the automatically generated physical types. The precondition annotations, which would be required by any analysis proving the same properties as ours, vary between 0 and 2 lines. On these benchmarks centered around memory safety, the vast majority of manual annotations consist in specifying that a pointer field should not be null by means of a refinement predicate. The only exception is the data field of AVL tree nodes (detailed in the annotation example above).

7.4 Related work on static analysis of low-level code

7.4.1 Analysis of machine code

Kinder et al. [KZV09] propose an abstract domain for control flow reconstruction, similar to our CFG-based domain $\mathbb{G} \times \mathbb{C}$. Our computation of new edges in the CFG resembles their *resolve* operator. Unlike us, Kinder et al. frame the problem of computing a sound abstract state at each node of the partial CFG as a data flow problem, which they solve with a worklist-based algorithm. This leads to suboptimal behavior in the presence of loops: if the state abstract domain has infinite increasing chains, then they must either sacrifice termination or replace all their joins by widenings, which is costly in terms of precision. In contrast, we frame the problem of computing abstract states in a partial CFG as the computation of the abstract semantics of a flowchart program, which allows us to reuse standard techniques for such computations, including the choice of a small set of widening points [Bou93], while still enforcing termination.

CodeSurfer/x86 is a tool to analyze machine code executables [Bal07; Bal+05], designed to be used in conjunction with the commercial disassembler IDAPro. It takes as input the CFG, function boundaries and call graph produced by IDAPro as the starting point of the analysis, and performs an abstract interpretation on the given CFG to infer over-approximations of possible states at each location, like we do. Although the CFG given by IDAPro may be incorrect or incomplete, CodeSurfer/x86 completes it into an over-approximation of the concrete CFG using a fixpoint computation similar to ours, or emits an alarm if it fails to do so. Unlike us, CodeSurfer/x86 abstracts sets of heap regions by their allocation site, in the manner of pointer analyses. We therefore expect it to struggle with code manipulating large or dynamically-allocated data structure, since many heap objects will be abstracted into a single summary node, regardless of the structure of data. To regain some precision, Balakrishnan and Reps developed the recency abstraction [BR06], which we mentioned in Section 6.5 and which notably enabled CodeSurfer/x86 to gain precision in analysis of code involving virtual function tables.

7.4.2 Analysis of low-level C

Miné [Min06] proposes a way to use a classical numerical abstract domain —i.e. a numerical domain that concretizes to $\mathcal{P}(\mathcal{V} \to \mathbb{V})$, where $\mathcal{V}$ is a finite set of variables and $\mathbb{V}$ the set of values— to analyze low-level C code that performs byte-level access to values, possibly involving unions, pointer casts, and pointer arithmetic. To achieve this, the numerical domain is used not directly on the set of variables but on a collection of abstract memory cells, which are in turn mapped to the contents of C variables. The abstract cells play a role similar to our symbolic variables, except that in [Min06] the set of cells is finite. In contrast, in our framework the set of symbolic variables is infinite, which always permits to use hitherto unused variables to represent, e.g., memory allocation.

As already mentioned in Chapter 2, the shape analysis tool Predator has been extended to support low-level C programming patterns such as unrestricted pointer arithmetic, pointers to a non-zero offset in a structure, manipulation of pointers to invalid targets, or block memory operations. No other shape analysis to our knowledge deals with such low-level constructs, even though some abstractions have been adapted to deal with some forms of pointer arithmetics [LCR10; KSV10].

Part II

End-to-end Verification of Embedded Kernels

Chapter 8

Kernel semantics and implicit properties

Outline of the current chapter

In order to present our methodology of OS kernel verification, in this chapter we define the semantics of a system managed by a kernel, at the machine code level (Section 8.1). After defining state properties (Section 8.2) and absence of run-time errors, we formally define absence of privilege escalation as a state property in Section 8.3, and show that it is also an implicit property (Section 8.4), that is, a property that does not require writing a specification tailored to each OS kernel.

8.1 System loop

We consider a computer system consisting in hardware running two kinds of code in succession: a *kernel runtime*, which is security-critical, and *user code*. The kernel runtime starts executing whenever an *interrupt* occurs, either a software interrupt (e.g. an application performed a system call) or a hardware interrupt (e.g. due to a timer firing, or an illegal memory access). We call *system loop* this alternation of kernel code and user code. In the context of kernel verification, the user code is unknown. Our goal is to ensure that the only code running as privileged is the uncompromised, security-critical code. We model the *privileged* nature of a system state by a predicate `privileged` $: \mathbb{S} \to \mathbb{B}$ on machine states.

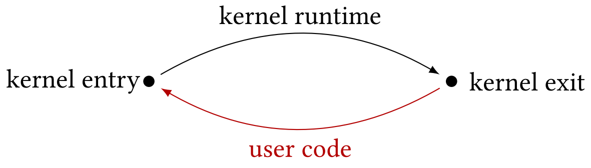


Figure 8.1: Alternated execution of kernel and user code.

```
typedef struct {
  Int8 pc;
  Int8 sp;
  Int8 flags;
} Context;

typedef struct {
  Int8 base;
  Int8 size_and_rights;
} Segment;

typedef struct {
  Segment code;
  Segment data;
} Memory_Table;

typedef struct Thread {
  Memory_Table *mt;
  Context ctx;
  Thread *next;
} Thread;

typedef struct {
  Thread *threads;
  Int8 threads_length;
} Interface;

register Int8 sp′, pc′, flags′,
              mpu1, mpu2;
Thread *cur;
Context *ctx;
extern Interface *itf;
```

```
void kernel() {
  switch( interrupt_number() ) {
    case RESET:
      init();
      load_mt();
      load_context();
    case YIELD_SYSCALL:
    case TIMER_INTERRUPT:
      save_context();
      schedule();
      load_mt();
      load_context();
    case ... : ...
  }
}

void save_context() {
  ctx->pc = pc′;
  ctx->sp = sp′;
  ctx->flags = flags′;
}
void schedule() {
  cur = cur->next;
  ctx = &cur->ctx;
}
void init() {
  cur = &itf.threads[0];
  ctx = &cur->ctx;
}
void load_mt() {
  mpu1= &cur->mt->code;
  mpu2= &cur->mt->data;
}
void load_context() {
  pc′= ctx->pc;
  sp′= ctx->sp;
  flags′= ctx->flags;
}
```

Figure 8.2: Code of a simple embedded OS kernel.

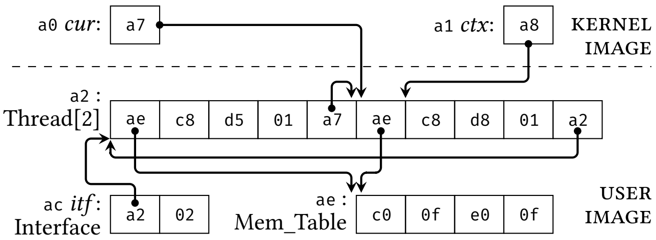


Figure 8.3: Example of a memory dump of the system.

In a typical OS kernel, this predicate corresponds to the value of a special register —or a bit in a flags register— containing the hardware privilege level.

8.1.1 Attacker model and trusted components

The attacker's goal is to escalate privilege, either by running untrusted software with privilege or by injecting code into the kernel runtime. The attacker controls the user code and user data—loaded with the kernel before boot—and can perform any software-based attack, such as modifying user task code and memory at runtime; but cannot make the hardware deviate from its specification, and thus cannot perform physical attacks nor exploit hardware backdoors or glitches. We trust only a minimal number of components:

- the software used to load the kernel and user tasks to memory (bootloader, EEPROM flasher, etc.);
- and the tools used to perform the formal verification.

8.1.2 Example kernel

Consider the code in Figure 8.2 which implements a simple kernel; for simplicity's sake, this kernel executes on fictitious 8-bit hardware; and we show it as a C program, although what we analyze is the kernel executable. The user code executes using three registers `pc′`, `sp′` and `flags′`, distinct (e.g. banked) from those used for the kernel execution. The kernel executes whenever an interrupt occurs. If the interrupt corresponds to a preemption (e.g. user code calls a "yield" system call, or the kernel receives a timer interrupt), it saves the values of the registers of the preempted thread ("`save_context`"), determines the next thread to be executed ("`schedule`", here a simple round-robin scheduler), sets up the memory protection to limit the memory accessible by this thread ("`load_mt`"), restores the previous register values of the thread ("`load_context`"), and transitions to user code. In this example, memory protection is done using two Memory Protection Unit (MPU) registers ($\texttt{mpu}_1$ or $\texttt{mpu}_2$): user code can only access the memory addresses allowed by one of the `mpu` registers, each giving access to one segment (i.e. interval of addresses). In addition, unsetting the *hardware privilege* flag (`PRIVILEGED` bit in the `flags′` register) forbids user code to change the values of $\texttt{mpu}_1$ and $\texttt{mpu}_2$.

A special interrupt (RESET) corresponds to the system *boot*, where the kernel initializes the memory ("`init`"). Additional interrupts (the other `case`s) would perform additional actions, like other system calls or interfacing with hardware.

Memory layout and parameterization

Let us now look at the memory layout of the kernel (Figure 8.3). The kernel is *parameterized*, i.e. independent from the *user tasks* running on it: both the kernel and user tasks can be put in separate executable images and linked at either compile-time or boot-time. This separation is necessary for closed-source kernel vendors, and also allows certifying the kernel image independently to reuse this certification in several applications.

For instance, in Figure 8.3, addresses `a0..a1` come from the kernel executable (the *kernel image*), while addresses `a2..b2` come from the user tasks executable (the *user image*). While Figure 8.3 represents a system composed of two threads sharing the same memory table, the same kernel image can be linked to another user image to run any number of threads, each with different memory rights.

A consequence of this parameterization is that the addresses of many system objects (e.g. of type `Thread` or `Memory_Table`) vary and are not statically known in the kernel. It makes the code much harder to analyze and explains why in existing automated verification approaches [Dam+13; Nel+19; Nor20], these objects must be statically allocated in the kernel, hardcoding a fixed limit on the number of `Threads`, for instance.

Interface and precondition

The kernel and user images agree on a shared *interface* to work together. In our example, this interface consists in having the variable `itf` at an address known by both images; a common alternative is to use the bootloader to share such information [FB]. Also, the system is not expected to work for any user task image with which the kernel would link. For instance the system would misbehave, if `itf->threads` pointed to `ctx`, to the kernel code or to the stack. Thus, system correctness depends on some *precondition* on the provided user image.

8.2 State properties

We model the system consisting of the kernel and the tasks as a transition system $\langle \mathbb{S}, S_0, \rightarrow \rangle$, where S_0 is the set of initial states, and → corresponds to the transition from one instruction to the next defined in Section 7.2.1. In all states of S_0, the memory contains the code of the kernel at the addresses specified by the kernel executable; this comes from the fact that we trust the bootloader.

Definition 8.1 (State property). A *state property* P is a set of states $P \subseteq \mathbb{S}$. The transition system $\langle \mathbb{S}, S_0, \rightarrow \rangle$ satisfies P if and only if all states reachable from S_0 belong to P, i.e., if $[\![S_0]\!]_{\text{bin}} \subseteq P$.

8.2.1 Absence of run-time errors

The definition of what is a run-time error varies depending on programming languages and verification requirements. In C, for instance, floating-point computations can produce infinities or NaN values, which is not an error; however, in some verification contexts where infinities and NaNs are not desirable, the production of such values is considered to be a run-time error.

Similarly, in machine code, as part of the hardware specification, some operations can trigger software interrupts, namely illegal opcodes, illegal memory accesses, and division by zero. However, these events are usually unwanted. We make the choice to consider the occurrence of such events as run-time errors, and model them by a transition to the error state Ω. The definition of absence of runtime errors is thus straightforward:

Definition 8.2 (Absence of run-time errors). The transition system $\langle \mathbb{S}, S_0, \rightarrow \rangle$ satisfies *absence of runtime errors (ARTE)* if and only if $\Omega \notin [\![S_0]\!]_{\text{bin}}$.

ARTE is thus a state property.

Our analysis proves ARTE by verifying that the absence of transitions to Ω.

	unprivileged	privileged
kernel-controlled	✓	✓
attacker-controlled	✓	privilege escalation

Figure 8.4: Privilege escalation happens when the system reaches a privileged state controlled by the attacker.

8.3 Absence of privilege escalation as a state property

8.3.1 Definition

The semantics of some assembly instructions depends on the privilege level. For instance on usual processors, *system registers* cannot be modified when in an unprivileged state.

Two entities are sharing their use of the system, called the *kernel* and the *attacker*. In the kernels that we consider, a state is kernel-controlled if the next instruction that it executes comes from the kernel executable file and was never modified; all the other states are considered attacker-controlled.

Notation 8.1. *Given a state $s \in \mathbb{S}$, we let* A-controlled(s) *denote the fact that s is attacker-controlled.*

In the context of the embedded kernels that we study, A-controlled(s) is false if and only if s.`pc` belongs to the region of kernel code, and that region was never modified.

Definition 8.3 (Absence of privilege escalation). We define privilege escalation as reaching a state that is both privileged and attacker-controlled (Figure 8.4). On the contrary, a transition system $\langle \mathbb{S}, \mathrm{S}_0, \rightarrow \rangle$ enjoys *absence of privilege escalation (APE)* if no such state is reachable.

Thus, an attacker can escalate its privilege by either gaining control over privileged kernel code (e.g. by code injection), or by leading the kernel into granting privilege to user code (e.g. by corrupting the `flags′` register).

8.3.2 A semantics suitable for parameterized verification

The semantics of the system depends on the user code. But we want to verify the kernel independently from any particular user code that it may run.

In the rest of this thesis, we call *parameterized verification of the kernel* a verification that does not depend on user code.

Definition 8.3 cannot be used directly for parameterized verification, because the execution of the whole system depends on the user code. We solve this problem by defining a new semantics for machine code which is independent from the attacker's execution.

Regular and interrupt transitions

We partition the transition relation into *regular transitions* and *interrupt transitions*. Regular transitions either preserve the current privilege level or go from a privileged state to an unprivileged one, but *cannot* evolve from an unprivileged state to a privileged one. *Interrupt transitions* are the only way to evolve from unprivileged to privileged. In OS kernels, it corresponds to the reception of hardware or software interrupts.

Empowering the attacker

In this section, we will define a new, approximated transition system in which the attacker is more powerful.

We first define relation $\overset{A}{\leadsto}$ which over-approximates the transitions that an attacker can effectively perform (i.e., we make the attacker more powerful). Instead of only being able to execute the next instruction under the program counter, the attacker will now be able to execute sequences of arbitrary instructions:

Definition 8.4 (Empowered transition relation). We call *empowered transition* a transition corresponding to the execution of any instruction. Recall that $\mathcal{I} \subseteq \mathit{stmt} \times \mathit{jump}$ denotes the set of assembly instructions supported by the hardware. The empowered transition relation $\overset{A}{\leadsto} \subseteq \mathbb{S} \times \mathbb{S}$ is defined as:

$$s \overset{A}{\leadsto} s' \iff \exists i \in \mathcal{I},\ s' \in [\![i]\!]_{\mathcal{I}}\{s\}$$

The reflexive and transitive closure of $\overset{A}{\leadsto}$ is denoted as $\overset{A}{\leadsto}^*$.

We now define the new transition system $\langle \mathbb{S}, \mathbb{S}_0, \leadsto \rangle$ with a new transition relation $\leadsto$. The $\leadsto$ relation restricts the ability to execute arbitrary instructions to attacker-controlled states; when a state is kernel-controlled, the normal transition relation $\rightarrow$ applies. In addition, interrupt transitions are also possible in attacker-controlled states.

Definition 8.5 (Attacker-abstracted transition relation). The attacker-abstracted transition relation $\leadsto \subseteq \mathbb{S} \times \mathbb{S}$ is defined by:

$$s \leadsto s' \iff s \rightarrow s' \ \vee\ (\text{A-controlled}(s) \wedge s \overset{A}{\leadsto}^* s')$$

The $\leadsto$ relation still restricts the ability of the attacker to execute arbitrary instructions, as hardware restrictions related to privilege and memory protections still apply. In addition, interrupt transitions are possible in attacker-controlled states.

Lemma 8.1. *Every state reachable in the transition system $\langle \mathbb{S}, \mathbb{S}_0, \rightarrow \rangle$ is also reachable in $\langle \mathbb{S}, \mathbb{S}_0, \leadsto \rangle$.*

Proof. Easy from the fact that $\rightarrow$ is included in $\leadsto$ by definition (Definition 8.5). □

Now, to verify APE although the user code is not known to the analyzer, we rely on the following assumption:

Assumption 8.1. *Running an arbitrary sequence of privileged instructions allows to reach any state of $\mathbb{S}$.*

This assumption is reasonable; it amounts to assuming that an arbitrary sequence of instructions with hardware privilege allows changing any register or memory location, thus reaching any state. However, depending on the hardware, some registers may hold only a restricted set of values; or some memory locations tied to hardware devices may not be modifiable. In that case it is enough to modify the definition of the set of states $\mathbb{S}$ to only include reachable states.

From this assumption, we can prove the following theorems:

Lemma 8.2. *If a transition system $\langle \mathbb{S}, \mathbb{S}_0, \leadsto \rangle$ is vulnerable to privilege escalation, then the only satisfiable state property in the system is the* trivial state property $\top$*, true for every state.*

Proof. If a privilege escalation vulnerability exists, by definition there is a reachable state that is both attacker-controlled and privileged (Definition 8.3), therefore the transition system contains all executions of arbitrary sequences of instructions from that state (Definition 8.5) and thus, by Assumption 8.1, any state of $\mathbb{S}$ is reachable. □

Theorem 8.3. *If a transition system $\langle \mathbb{S}, S_0, \rightsquigarrow \rangle$ satisfies a non-trivial state property, then it also satisfies APE.*

Proof. This theorem is the contrapositive of Lemma 8.2. □

Theorem 8.3 has two crucial practical implications:

- if privilege escalation is possible, the only state property that holds in the system is the trivial one, making it impossible to prove definitively any other property. Thus, *proving absence of privilege escalation is a necessary first step* for any formal verification of an OS kernel;
- The proof of *any* non-trivial state property implies as a byproduct the existence of a piece of code able to protect itself from the attacker, i.e. a kernel with protected privileges. In particular, we can prove absence of privilege escalation automatically, by successfully inferring *any* non-trivial state property with a sound static analyzer.

8.4 Implicit properties

We call implicit properties program properties whose definition only depends on the semantics considered, and not on any particular program. ARTE, or the preservation of typing that we defined in Chapter 4, are examples of implicit properties, as opposed to e.g. loop invariants, user assertions or function contracts. Their advantage is that they can be verified more easily, as they do not require a tailored specification to be written for each program.

Definition 8.6 (Implicit poperty). An implicit property is a property that does not depend on a particular program.

Theorem 8.4. *ARTE is an implicit property.*

Proof. We defined it without reference to specific programs. □

In the following theorem, we state formally that preservation of structural invariants is an implicit property. Since the typed semantics $[\![\cdot]\!]_t$ excludes ill-typed executions, programs that preserve structural invariants are characterized by the fact that their typed semantics contains the same states as their untyped semantics (up to type erasure).

Theorem 8.5. *Preservation of structural invariants is an implicit property. More precisely, given a mapping $\mathcal{M}$ from type names to types and a set of initial well-typed states S_0, the property $\rho(P)$, that holds if and only if* $(\text{untyp} \circ [\![P]\!]_t)S_0 = ([\![P]\!] \circ \text{untyp})S_0$*, is an implicit property.*

Proof. The property ρ applies to any P and its definition only depends on a mapping $\mathcal{M}$ and a set of well-typed states. □

Theorem 8.6. *Absence of privilege escalation is an implicit property.*

Proof. By Theorem 8.3, absence of privilege escalation is equivalent to the existence of a non-trivial state property in the semantics $\rightsquigarrow$. □

Note that the definition of $\rightsquigarrow$ depends on the A-controlled predicate, whose definition depends on the location of the kernel code in memory in the initial state. However, we argue that it is still reasonable to call APE implicit, as the A-controlled predicate is a notion applicable to any embedded kernel, and can be derived automatically from the kernel image.

Recasting APE as an implicit property allows to vastly lighten the verification effort compared to e.g. a verification using assisted program proof, which requires expliciting invariants tailored to the kernel being verified [Kle+09; WKP80; Bev89; Ric10; Gu+15; Xu+16; Alk+10; YH11; Vas+16; Fer+17], either at the level of the main system loop, or at the level of smaller loops in the code.

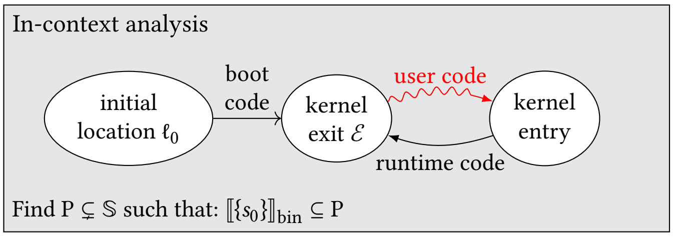

In-context analysis
initial location ℓ_0
boot code
kernel exit $\mathcal{E}$
user code
kernel entry
runtime code
Find $\mathrm{P} \subsetneq \mathbb{S}$ such that: $[\![\{s_0\}]\!]_{\text{bin}} \subseteq \mathrm{P}$

Figure 8.5: Fully automated, in-context analysis of the system.

8.5 In-context verification of kernels

We now describe how to use an analysis like the one described in Section 7.2 to verify an OS kernel *in context*, that is, for a given interface.

As Theorem 8.3 states, it is enough to verify any non-trivial state property on the whole kernel–applications system to establish APE. One therefore needs to use a static analysis to over-approximate the set of reachable states $[\![\{s_0\}]\!]_{\text{bin}}$ (see Definition 7.5), and hopefully find a set distinct from $\mathbb{S}$.

To do that, one can use a "flat model" memory domain similar to $\mathbb{M}^\sharp$ (Section 7.2.4) to represent the entire memory; the type-based shape domain is not necessary, since the interface is known at the time of the analysis. Since all data manipulated is located at known addresses, this domain simply represents the memory as a mapping from addresses to abstract values:

Definition 8.7 (Fully enumerative state abstract domain)**.** The fully enumerative state abstract domain is defined as:

$$\mathbb{S}^\sharp_{\text{flat}} = (\mathbb{X} \to \mathbb{V}^\sharp) \times (\mathbb{A} \rightharpoonup \mathbb{V}^\sharp) \times \mathbb{D}_{\text{num}}$$

with concretization:

$$\begin{aligned} \gamma_{\text{flat}} &: \mathbb{S}^\sharp_{\text{flat}} \to \mathcal{P}\big(\mathbb{S} \times (\mathbb{V}^\sharp \to \mathbb{V})\big) \\ \gamma_{\text{flat}}(\sigma^\sharp, h^\sharp, \upsilon^\sharp) &= \left\{ ((\sigma, h),\ \upsilon) \mid \begin{array}{l} \forall x \in \mathbb{X},\ \sigma(x) = \upsilon(\sigma^\sharp(x)) \\ \forall a \in \mathbb{A},\ h(a) = \upsilon(h^\sharp(a)) \end{array} \right\} \end{aligned}$$

The abstract semantics of $\mathbb{S}^\sharp_{\text{flat}}$ is the obvious, point-wise lifting of the abstract operators of $\mathbb{D}_{\text{num}}$.

Like $\mathbb{M}^\sharp$ in Section 7.2.4, the domain $\mathbb{S}^\sharp_{\text{flat}}$ is implemented using optimizations that are standard.

8.5.1 Abstracting the attacker-controlled transition

In order to overapproximate $\rightsquigarrow$ in our analysis, we abstract the attacker-controlled transition as follows: whenever control jumps to an instruction that is not in the code of the kernel, i.e. whenever the next state is possibly attacker-controlled, two cases are possible:

- if that stage is privileged, then the analysis reports a possible privilege escalation, and stops.
- Otherwise, the analysis drops all knowledge about non-system registers, i.e., registers that can be modified by unprivileged code, and about the parts of memory that are *not* protected against writing by the memory protections currently in place. The analysis then continues from the beginning of the kernel runtime —e.g. from the beginning of the `kernel()` function in the example kernel of Figure 8.2.

For instance, in our example kernel, the abstract attacker-controlled transition is triggered at the end of the function `load_context()`, since the next instruction to execute is an unknown, not kernel-controlled instruction. This effectively results in analyzing the system loop, where the user code is replaced by the abstract attacker-controlled transition (Figure 8.5). As for all loops, the analysis continues until a fixpoint is reached.

8.5.2 Illustration on the example kernel

We illustrate our method on the example kernel of Figure 8.2.

Abstract attacker-controlled transition

The real attacker-controlled transition (which does not appear in the example kernel code) consists in executing the code that starts at `pc′` after the end of `load_context()`. This transition can be abstracted by assigning unknown, unconstrained values to:

- the user registers `pc′`, `sp′`, and `flags′`;
- the memory locations that are not protected by either $\texttt{mpu}_1$ or $\texttt{mpu}_2$, if `flags′` has the `PRIVILEGED` bit unset;
- every memory location, if `PRIVILEGED` is set in `flags′`.

Note that the implementation of this transition is not specific to the kernel that we analyze (it is not “hardcoded”): it is only specific to the hardware on which the kernel is implemented.

Computed state property

Using our abstract interpreter for machine code on the system composed of the kernel code with the special user code transition, starting with the initial state of Figure 8.3, our analysis computes a state property which implies[1] that, at all times in the system loop (i.e. after boot):

- Addresses `a0` (variable `cur`), `a6` and `ab` hold either `a2` or `a7`;
- address `a1` (variable `ctx`) can hold values `a3` or `a8`;
- Register $\texttt{mpu}_1$ always holds `ae` and $\texttt{mpu}_2$ always holds `b0`;
- Addresses `a2` and `a7` always hold `ae`;
- Addresses `a5` and `aa` and the `flags′` register can hold any value with the *PRIVILEGED* bit unset;
- Addresses `a3`, `a4`, `a8`, `a9`, and all the other registers can have any value;
- Addresses in the range [`c0`, `cf`] and [`e0`, `ef`], (made accessible by the `mpu` registers) can have any value;

This automatically computed invariant must be manually written in prior automated methods [DGN13; Nel+19; Nor20].

Verifying implicit properties

The state property above implies that when the user code executes, the system is set up such that execution is not `PRIVILEGED`, and the memory protection tables prevent the user code from modifying kernel data. In fact, the mere presence of a non-trivial invariant implies that APE is impossible. Verifying ARTE (i.e. no faulty execution of an instruction) is simple as our invariant incorporates the CFG (i.e. the set of all instructions executable by the kernel) and computes a superset of values for every operand of every instruction.

[1] The property computed consists in a set of states for every possible program location.

In other words, an in-context analysis enables us to prove APE and ARTE on this particular system, *automatically, without any annotations*. And indeed we show in our experiments (Section 10.6) that our tool is able to prove APE on some simple kernels by an in-context analysis.

However, the system is only verified for a given, known interface. In the next chapter, we make this method more general and perform *parameterized* verification.

Chapter 9

Parameterized verification of OS kernels

Outline of the current chapter

In the previous chapter, we presented a method for verifying ARTE and APE on computer systems consisting in a kernel image (containing kernel code and initial data) and a user image (containing application code and initial data). More precisely, what is needed is only the kernel image and the *interface*, i.e. a part of the user image whose location is a convention between kernel and application developers. We called this method *in-context verification* because it requires an instance of the interface.

In-context verification does not satisfy our objective of verifying a kernel in a *parameterized* way, that is, independently from any user image. In-context verification also has other limitations, namely the lack of robustness and scalability of the analysis. We discuss these points in Section 9.1. We then propose a method of parameterized kernel verification. Section 9.2 gives a high-level view of this method, and we illustrate it in Section 9.3 on our example kernel. Finally, we remark that this method is insufficient for kernels with a complex initialization procedure; Section 9.4 proposes a *differentiated handling of boot and runtime code* to verify such kernels.

9.1 Shortcomings of in-context verification

The fully automated verification works in practice for small, simple kernels, but has some shortcomings. The root of the problem lies in the *flat memory model* (also used by all the previous binary-level automated methods [Dam+13; Nel+19; Nor20]), i.e. viewing the memory as a single large array of bytes. This abstraction poses three distinct problems:

- *Robustness*, i.e., the ability to recover from an imprecision. In the in-context analysis, imprecisions can cascade so much that the analysis can no longer compute an invariant. This would happen for instance if, in our example kernel, the set of possible values for variable `cur`, namely $\{\mathtt{a2}, \mathtt{a7}\}$, was over-approximated by the interval $[\mathtt{a2}, \mathtt{a7}]$. The root cause of this lack of robustness is that a *numerical* abstraction of addresses must be very precise to preserve the *structure of data* in memory.
- *Scalability*, because of the need to enumerate large sets. For instance, on a system running 1,000 tasks, the values for $\mathtt{mpu}_1$ may point to 1,000 different arbitrary addresses, which must be precisely enumerated if we want to be robust;
- *Over-specialization*: it is impossible to analyze programs without knowing the precise memory layout of all the data. In particular, the example kernel cannot be analyzed independently from the user image, even though this kernel has been designed to be independent from the user image.

We propose to use the type-based shape domain to lift those three limitations, at the expense of writing a small number of annotations.

9.2 Method overview

Our method extends that of Section 8.5 with the following key points:

Abstraction of the interface using physical types Instead of representing the interface contents using the flat memory model, we abstract it into its *structure*, expressed using physical types. Specifically, we use the structural invariants provided by types to precisely analyze the parts of kernel code that manipulate the interface; and we verify that the kernel preserves the well-typedness of the interface. This solves the robustness and scalability issues of the flat model and allows *parameterized analysis*, i.e. analysis of the kernel independently from a specific user image. The kernel memory, which mainly consists of the kernel's stack and global (mutable and immutable) data, on the other hand, is still represented numerically using the flat model, which allows to verify the kernel using its raw binary.

Differentiated handling of boot and runtime code We propose to handle kernel runtime and boot differently. On the one hand, the kernel runtime should *preserve* existing structural invariants that exist on the interface, while the boot code *establishes* those invariants by initializing the structures. This makes the runtime suitable for verification by our type-based shape domain, which aims at verifying the preservation of memory invariants. On the other hand, the boot code *establishes* those invariants by initializing the data structures present in the interface; and our experiments (see Section 10.4) show that our type-based domain is not precise enough on such initialization code. We therefore make use of the fact that the boot code is mostly deterministic (only small sources of non-determinism remain: multicore handling, device initialization, etc.) to analyze it robustly using in-context analysis (Section 8.5). Based on these observations we propose a method (Section 9.4) where boot code is verified using the in-context analysis, and the runtime code is verified using the parameterized static analysis.

Summary: a method in three steps We propose a three-step method that relies on two above key ingredients.

- *Step 1*: *Lightweight annotation of the interface types.* The structure of the interface should be known, as it is part of the convention between kernel and application developers; yet a number of additional annotations may be necessary for the verification to succeed (see example in Section 9.3.1 and cases studies of Chapter 10).
- *Step 2*: *Parameterized static analysis of the kernel.* The result of this step is a state property P holding on all executions, under the precondition that the initial state is well typed. If P is the trivial state property $\mathbb{S}$ (i.e. if the analysis is not precise enough), then go back to Step 1 to refine the interface types.
- *Step 3*: *Base case checking* consists in verifying that the initial state is indeed well typed.

9.3 Illustration on the example kernel

We illustrate how the 3-step method is applied on the example kernel of Figure 8.2 (repeated on page 122).

9.3.1 Lightweight type annotation

First, we define a set of physical types that describe the structure of the interface between the kernel and the applications, i.e. all data accessible through the global `itf`. Since knowing the structure of the interface is necessary in order to develop applications for the kernel, we expect this structure to be documented. Physical types can be automatically generated from C types or from debug annotations, if available. However, it may be necessary to refine those types, as in our experiments on ordinary non-kernel programs in Section 7.3.

In the kernels that we have studied, these refining annotations mostly consist in indicating the size of arrays (Figure 9.1, type `Interface`), and which pointers may be null. We also specify that the memory zone accessible to a task is disjoint from the kernel address space (Figure 9.1, type `Segment`), and that saved user flags are unprivileged (Figure 9.1, type `Flags`). Note that these manual annotations are smaller (and simpler) than the full loop invariant (Section 8.5.2); this is even more true on larger kernels.

Concretely, what we call annotations consists in a set of types like in Figure 9.1. These types must be written in a separate file and provided to our tool along with the kernel executable. No binary or source code is annotated; instead, our tool only requires to know the address and the type of the interface entry point (type `Interface` at address `ac` in the example), i.e. the memory location that the kernel should use to access any element of the interface. The set of types is necessarily provided by the kernel (and bootloader) developers as part of the software development kit; the refinement predicates on types `Flags`, `Segment` and `Interface`, required to successfully complete the verification, can also be provided as part of the kernel documentation; or finding them can be part of the verification work. We give practical details about this in Chapter 10.

9.3.2 Parameterized static analysis of the kernel

We then analyze the kernel code using the abstract domain $\mathbb{F}^\sharp$ (Section 6.4), starting from an initial abstract state $\mathbb{s}_0$ that is such that the global `itf` holds an address of type `Interface`.(0)∗.

The analysis then computes a state property $\mathrm{P} \subseteq \mathbb{S}$ such that $[\![\gamma_{\mathrm{F}}(\mathbb{s}_0)]\!]_{\mathrm{bin}} \subseteq \mathrm{P}$, i.e. starting from an initial state abstracted by $\mathbb{s}_0$, every reachable state is in P. On this simple kernel, the analysis will easily compute a non-trivial P (i.e. $\mathrm{P} \neq \mathbb{S}$) which implies in particular the following:

- Address `a0` (variable `cur`) holds an admissible `Thread*` value, i.e., `cur` $\in$ $(\!|$`Thread`.(0)∗$|\!)_{\mathcal{L},\upsilon}$;

```
typedef struct {
  Int8 pc;
  Int8 sp;
  Int8 flags;
} Context;

typedef struct {
  Int8 base;
  Int8 size_and_rights;
} Segment;

typedef struct {
  Segment code;
  Segment data;
} Memory_Table;

typedef struct Thread {
  Memory_Table *mt;
  Context ctx;
  Thread *next;
} Thread;

typedef struct {
  Thread *threads;
  Int8 threads_length;
} Interface;

register Int8 sp′, pc′, flags′,
       mpu_1, mpu_2;
Thread *cur;
Context *ctx;
extern Interface *itf;
void kernel() {
  switch( interrupt_number() ) {
    case RESET:
      init();
      load_mt();
      load_context();
    case YIELD_SYSCALL:
    case TIMER_INTERRUPT:
      save_context();
      schedule();
      load_mt();
      load_context();
    case ... : ...
  }
}

void save_context() {
  ctx->pc = pc′;
  ctx->sp = sp′;
  ctx->flags = flags′;
}
void schedule() {
  cur = cur->next;
  ctx = &cur->ctx;
}
void init() {
  cur = &itf.threads[0];
  ctx = &cur->ctx;
}
void load_mt() {
  mpu_1= &cur->mt->code;
  mpu_2= &cur->mt->data;
}
void load_context() {
  pc′= ctx->pc;
  sp′= ctx->sp;
  flags′= ctx->flags;
}
```

Figure 8.2: Code of a simple embedded OS kernel for a fictitious 8-bit hardware. (repeated from page 110)

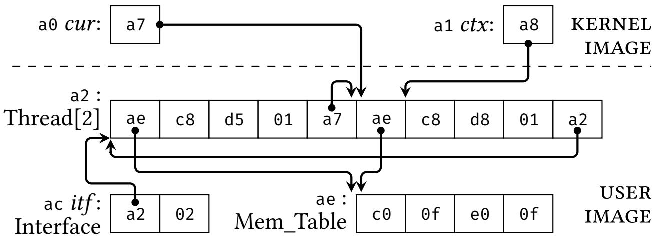


Figure 8.3: Example of a memory dump of the system. (repeated from page 111)

$$\mathcal{M} = \left[\begin{array}{rl}
\texttt{Flags} \mapsto & \{x : \texttt{word}_1 \mid x \mathbin{\&} \texttt{PRIVILEGED} = 0\} \\
\texttt{Context} \mapsto & \texttt{word}_1 \times \texttt{word}_1 \times \texttt{Flags} \\
\texttt{Segment} \mapsto & \{x : \texttt{word}_1 \mid x \geq \mathbb{e}_{\text{kernel}}\} \times \texttt{word}_1 \\
\texttt{Memory\_Table} \mapsto & \texttt{Segment} \times \texttt{Segment} \\
\texttt{Task} \mapsto & \texttt{Memory\_Table}.(0){*}_{\neq 0} \times \texttt{Context} \times \texttt{Task}.(0){*}_{\neq 0} \\
\texttt{Interface} \mapsto & \texttt{Task}[\mathbb{m}].(0){*}_{\neq 0} \times \{x : \texttt{word}_1 \mid x = \mathbb{m} \wedge x \geq 1\}
\end{array}\right]$$

Figure 9.1: Physical types representing the interface structure. The symbolic variables $\mathbb{e}_{\text{kernel}}$ and $\mathbb{m}$ represent the greatest address of the kernel code region and the number of tasks, respectively.

- Address a1 (variable ctx) holds an admissible Context* value, i.e., $\texttt{ctx} \in (\!|\,\texttt{Context}.(0){*}\,|\!)_{\mathcal{L},v}$;
- Registers $\texttt{mpu}_1$ and $\texttt{mpu}_2$ hold admissible Segment values;
- Register flags′ holds an admissible Flags value;
- The memory region holding the interface matches the types of Figure 9.1, i.e., the state is well typed.

For example, the example interface in Figure 8.3 verifies the above invariant. E.g. the set of admissible values for the type $\texttt{Thread}.(0){*}$ in that case is $\{\texttt{a2}, \texttt{a7}\}$. However, note that instead of describing the invariant for one given user image, this new invariant describes what happens for *all* (well-typed) user images, hence it is parameterized.

9.3.3 Base case checking

The previous step has established a non-trivial state property (and therefore APE) for all executions starting from a state in $\gamma_{\mathrm{F}}(\mathbb{s}_0)$. And $\mathbb{s}_0$ represents the states such that the interface is well typed; therefore, we have succeeded in verifying the kernel under this *precondition.*

All that remains to do is to make sure that we can decide whether a particular *instance* of the system, that is, an executable containing the kernel, the applications, and the interface, verifies this precondition. For this, we need an algorithm that takes a memory h, an address a and a valuation v and answers the following question: is there a labelling $\mathcal{L}$ such that h is consistent with $\mathcal{L}$ and $h[a..a+\mathrm{size}(\texttt{Interface})] \in (\!|\,\texttt{Interface}\,|\!)_{\mathcal{L},v}$? This algorithm is defined in Section 9.4.3.

9.3.4 Discussion

The whole process (parameterized analysis of the kernel and base case checking) is summarized graphically in Figure 9.2. Note that the technique is no longer fully automated, as the user has to provide type annotations (even if most of it comes from the existing types of the interface). But let us observe that

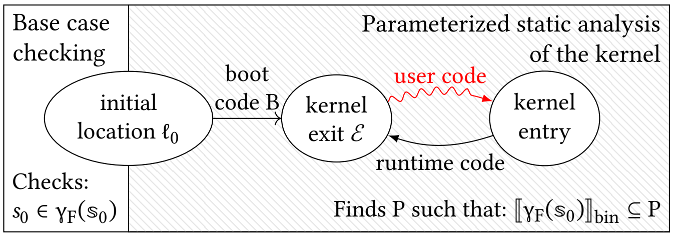

Base case checking
Parameterized static analysis of the kernel
initial location ℓ_0
boot code B
kernel exit $\mathcal{E}$
user code
kernel entry
runtime code
Checks: $s_0 \in \gamma_F(\mathbb{s}_0)$
Finds P such that: $[\![\gamma_F(\mathbb{s}_0)]\!]_{\text{bin}} \subseteq P$

Figure 9.2: Parameterized verification when user memory starts initialized.

the example kernel works correctly only when linked with a well-typed user image (according to Figure 9.1), and thus *parameterized verification of this kernel is impossible* without limiting, using a manual *precondition*, the admissible user image. Thus in general, *parameterized kernel verification is impossible without user-supplied annotations*, given here using the type annotations.

9.4 Differentiating boot and runtime code

9.4.1 Difficulties with the verification of the initialization code

In the example system, all the data structures in the user image are already initialized and thus are initially well typed for the types of Figure 9.1. In other systems this might not be the case: for instance, the values of the `next` field in `Threads` could be uninitialized, and thus the boot code would have to create the circular list; or the `Memory_Table` table could be dynamically allocated and filled during the boot, as in our case studies.

This poses a difficulty for our verification method: our experiments (Chapter 10) show that the type-based shape domain performs well to verify the *preservation* of structural invariants, but not so well to verify the *establishment* of such invariants.

One of the reasons for that is that our type-based domain is designed to work with a fixed labelling of memory; whereas in the context of an initialization procedure, the analysis would need to start with generic types and refine those types, depending on the operations of the program. Instead, our abstractions can only represent initialization in a limited way, through staged points-to predicates. However, this is insufficient if the kernel e.g. allocates and partly initializes several structures, and then iterates through them again to complete their initialization.

9.4.2 Principle of the differentiated verification

To handle these cases, we propose to perform the base case checking when the kernel has finished booting and enters its main loop (Figure 9.3), rather than at the beginning of the execution (Figure 9.2). More precisely, we:

- Perform the parameterized analysis —i.e., using the abstract domain $\mathbb{F}^\sharp$— of the boot code, starting from an initial state $\mathbb{s}_0$ where the interface is initialized (i.e., well typed). *If the analysis emits alarms when analyzing the boot code, we ignore them.* Ignoring alarms is equivalent to ignoring erroneous executions. From this we deduce an abstract state $\mathbb{s}_{\text{booted}}$. The idea is that we make the temporary hypothesis that this abstract state is, in fact, a sound abstraction of all post-boot states. However, since alarms can be emitted during the analysis, we have no guarantee of that.
- Perform the parameterized analysis of the kernel runtime code, taking $\mathbb{s}_{\text{booted}}$ as the initial abstract state to obtain a state property P. If there are any alarms, or equivalently if $P = \mathbb{S}$, stop here.

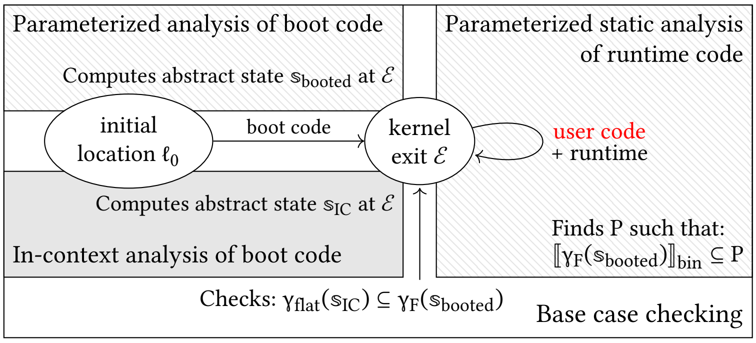

Parameterized analysis of boot code
Computes abstract state $\mathbb{s}_{\text{booted}}$ at $\mathcal{E}$
Parameterized static analysis of runtime code
initial location ℓ_0
boot code
kernel exit $\mathcal{E}$
user code
+ runtime
Computes abstract state $\mathbb{s}_{\text{IC}}$ at $\mathcal{E}$
In-context analysis of boot code
Finds P such that:
$[\![\gamma_{\text{F}}(\mathbb{s}_{\text{booted}})]\!]_{\text{bin}} \subseteq \text{P}$
Checks: $\gamma_{\text{flat}}(\mathbb{s}_{\text{IC}}) \subseteq \gamma_{\text{F}}(\mathbb{s}_{\text{booted}})$
Base case checking

Figure 9.3: Parameterized verification when user memory needs initialization.

If $\text{P} \neq \mathbb{S}$, then we have a proof of APE on the kernel *under the precondition* that the state at the end of the boot code is in $\gamma_{\text{F}}(\mathbb{s}_{\text{booted}})$. Given an instance of the system, this precondition can be verified as follows:

- Perform a fully automated, in-context analysis (i.e., using a flat model memory abstraction) of the boot code with the given interface to get an abstract state $\mathbb{s}_{\text{IC}} \in \mathbb{S}^{\sharp}_{\text{flat}}$ at the end of boot code.
- Check that all the "in-context" states at the end of the boot match the parameterized invariant, i.e. check that $\gamma_{\text{flat}}(\mathbb{s}_{\text{IC}}) \subseteq \gamma_{\text{F}}(\mathbb{s}_{\text{booted}})$. We detail how this check is performed in Section 9.4.3.

Once this is done, the full system is verified: the combination of both analyses imply the existence of a non-trivial state property on the whole system, and thus APE. This process is summarized graphically in Figure 9.3. Finally, this method allows to verify in addition the absence of run-time errors, both in the boot code and the runtime.

This method uses two characteristics of the boot code:

- First, one goal of the boot code is to initialize the data structures so that they are well-typed according to Figure 9.1. The initial value for these types is unimportant as they will be overwritten, so the code will also work if these data structures are initialized.
- The execution of the boot code is almost deterministic (except for hardware device handling and multicore execution) once the interface contents are known, thus the in-context analysis in this case is both robust and scalable (as the abstract values that are propagated abstract mostly singletons).

Note that this method avoids the need to specify (and annotate) the code with the actual precondition on the user image; here, the precondition that we verify is that "the user image should be such that its initialization will make it well-typed", using the fact that it is easier to describe the runtime types than the initial types.

9.4.3 Base case checking

We now detail how to perform the second step of base case checking, namely checking that all the "in-context" states at the end of the boot match the parameterized invariant, i.e. check that $\gamma_{\text{flat}}(\mathbb{s}_{\text{IC}}) \subseteq \gamma_{\text{F}}(\mathbb{s}_{\text{booted}})$.

This second step is performed as follows:

1. first, the contents of the interface is checked for compatibility with type `Interface`;

2. if that is the case, then $\mathbb{s}_{\mathrm{IC}}$ is abstracted into an element of $\mathbb{F}^\#$, denoted as $\mathbb{s}'_{\mathrm{IC}}$, by abstracting away the interface and all the rest of non-kernel memory, and simply dropping the address range $\overline{\mathbb{A}} \setminus \mathbb{A}$ from the abstract state.
3. Finally, the tool checks whether $\mathbb{s}'_{\mathrm{IC}} \sqsubseteq_{\mathbb{F}^\#} \mathbb{s}_{\mathrm{booted}}$.

We now detail how the first step, i.e. checking the compatibility between a memory region and a type, is performed.

Typechecking of memory regions

This algorithm decides whether a region is well typed, i.e. whether the value it holds is in $(\!| t |\!)_{\mathcal{L},\upsilon}$ for some type t.

More precisely, the algorithm takes a memory heap h, a valuation υ, an address a_0 and a type t; it either returns a labelling $\mathcal{L}$ such that $h[a..a + \mathrm{size}(t)] \in (\!| t |\!)_{\mathcal{L},\upsilon}$, or fails.

It works by constructing the labelling $\mathcal{L}$ by following pointers recursively, if possible. In the same process, it checks the satisfaction of refinement predicates.

The valuation parameter υ serves to assign concrete values to the symbolic variables present in the refinement predicates and array sizes of the type t; e.g. the symbolic variables $\mathbb{e}_{\mathrm{kernel}}$ and $\mathbb{n}$ for the example types of Figure 9.1. This instantiation of symbolic variables can be provided manually, but in many cases it can be derived automatically from the system image; it was the case for the kernels that we verified (Chapter 10).

This algorithm works by recursively updating a mapping $\mathcal{L} : \mathbb{A} \rightharpoonup \mathbb{T}_{\mathrm{A}}$, which is initially empty. Then the procedure $\mathtt{check}(a_0, t)$ proceeds as follows:

1. For all $a \in [a_0,\ a_0 + \mathrm{size}(t)[$,
   - if $a \notin \mathrm{dom}(\mathcal{L})$, then replace $\mathcal{L}$ with $\mathcal{L}[a \leftarrow t.(a - a_0)]$.
   - Otherwise, if $\mathcal{L}(a) \preceq t.(a - a_0)$ or $t.(a - a_0) \preceq \mathcal{L}(a)$, then the most precise of the two address types (in the $\preceq$ sense) becomes the new $\mathcal{L}(a)$.
   - Otherwise, the two types are incompatible, and the procedure fails.
2. Then,
   - if $t = \mathtt{word}_n$, then return.
   - If $t = \mathtt{n}$ (where $\mathtt{n}$ is a type name), then perform $\mathtt{check}(a_0, \mathcal{M}(\mathtt{n}))$.
   - If $t = u.(k)*$, then let $v = h[a_0 - k..a_0 - k + \mathcal{W}]$. If $v = 0$, then return; otherwise perform $\mathtt{check}(v, u)$.
   - If $t = t_1 \times t_2$ then perform $\mathtt{check}(a_0, t_1)$ and then $\mathtt{check}(a_0 + \mathrm{size}(t_1),\ t_2)$.
   - If $t = \{x : u \mid p(x)\}$, and predicate p is satisfied, i.e. $\mathrm{eval}_\upsilon(p, h[a_0..a_0 + \mathrm{size}(t)]) = \mathrm{true}$, then perform $\mathtt{check}(a_0, u)$; otherwise the procedure fails.
   - If $t = u[s]$, then for each $i \in [0, s[$, perform $\mathtt{check}(a_0 + i \cdot \mathrm{size}(u),\ u)$.

If the procedure does not fail, then $\mathcal{L}$ can be completed into a total function by mapping all unmapped addresses to $\mathtt{word}_1.(0)$. Then, we have $h[a..a + \mathrm{size}(\mathtt{Interface})] \in (\!| \mathtt{Interface} |\!)_{\mathcal{L},\upsilon}$, i.e. the interface is well typed for the labelling $\mathcal{L}$.

9.5 Conclusion

In this chapter, we have described a method to verify absence of privilege escalation and absence of runtime errors on OS kernels. This method is parameterized, in the sense that it is performed on the kernel independently of user code or data.

Note that, to complete the proof of ARTE and APE for a given user image, one must perform the base case checking on that user image. But as discussed in Section 9.3.4, any verification of an embedded kernel is conditional to a precondition on interface data, although this precondition is sometimes left implicit.

Next chapter details the application of our parameterized verification method on two embedded kernels.

Chapter 10

Kernel verification case study and experimental evaluation

Outline of the current chapter

We have applied the analysis described in this thesis to two embedded kernels, including Asterios, a kernel used in the industry. We use these case studies to evaluate our verification methodology.

We seek to test the following Research Questions (RQ):

RQ0: **Soundness check** Our machine code analysis is sound, and therefore should emit alarms whenever we try to verify APE and ARTE on a kernel that is vulnerable or buggy. To test that this is the case, we run our analysis on kernels with deliberately introduced bugs or security flaws.

RQ1: **Real-life Effectiveness** Can our method verify real (unmodified) embedded kernels? How practical is it?

RQ2: **Internal evaluation** Are the type-based shape domain and the differentiated handling of the boot code necessary elements of our method? And what is the impact of increasing or reducing the amount of manual annotations?

RQ3: **Genericity** Can our method apply to different kernels, hardware architectures and toolchains?

RQ4: **Automation** Is it possible to prove APE and ARTE in OS kernels fully automatically, i.e. without manual annotations?

RQ5: **Scalability with increasing number of tasks** Although parameterized verification (see Chapter 9) consists in analyzing the kernel alone, to verify an instance of the embedded system one must also perform the base case checking step on the interface (see Section 9.2). Since this step depends on the characteristics of user tasks, and notably the number of tasks, we evaluate how it scales with large task counts.

These results were presented, along with our kernel analysis methodology, in an article published in the RTAS 2021 conference [Nic+21].

Section 10.1 describes the two embedded kernels that we analyzed, as well as the implementation of the analysis and other experimental conditions common to all experiments. The following sections are dedicated to the research questions. Each section describes the experiments performed, their results, and discusses the conclusions. Sections 10.2 to 10.5 describe the experiments for RQ0, RQ1, RQ2 and RQ3, respectively, while Section 10.6 describes our joint experiment for RQ4 and RQ5.

10.1 Experimental setup

10.1.1 Asterios

We consider for RQ1 and RQ2 the Asterios kernel, an industrial kernel for implementing security- and safety-critical hard real-time applications, used in industrial automation, automotive, aerospace and defense. It is developed by the Krono-Safe company. Krono-Safe has no prior experience in the use of formal methods, can only afford limited expenses on formal verification, and uses standard build tools. We believe that many small companies developing OS kernels have the same constraints.

Studying the formal verification of security properties of Asterios was one of the motivations of the present work. This context explains the key requirements of the method we developed, which must be:

- **Non-invasive**: Krono-Safe engineers are system developers, not formal method experts. They have important time-to-market constraints. It is thus important that formal verification does not get in their way. In practice, they gave us a development and final version of their kernel executable file on which the verification was performed; we never saw the source code of the kernel, and never asked them to compile specific versions: we verified the executable "as is". This is an important fact because to our best knowledge, all existing formally verified operating systems kernels were developed with the goal of being formally verified; this is not the case of the Asterios kernel.

- **Cost-effective and automatic**: Krono-Safe is an small-sized company and cannot afford spending a huge amount of effort into formal verification, thus the method should be cost-effective. In addition, their operating system can be tailored using a large set of options (the hardware, the number of cores on the board, how protection tables are used, the scheduler, etc), resulting in many versions of the system, so the method should be able to handle this variety. The automation of the technique is thus a critical factor.
- With a **high level of trust**, as their system operates in safety-critical and security-critical environment. Especially, all the tools and scripts used to produce the executable needed to be left out of the trusted computing base (TCB), because on some architectures they sometimes have to use vendor-provided tools that are not formally verified. This, and the need to comply to some safety standards requiring working at machine-code level like the DO-333, contributed to the decision to work only on the executable.

We consider a port of the kernel to a *4-core* ARM Cortex-A9 processor with ARMv7 instruction set. It relies on a Memory Management Unit (MMU) for memory protection (*pagination*). The kernel features a hard real-time scheduler that dispatches the tasks between the cores (migrations are allowed), and monitors timing budgets and deadlines. The kernel adopts a "static microkernel" architecture, with unprivileged services used to monitor inter-process communication. The configuration for the user tasks (i.e. the interface) is generated by the Asterios toolchain, and all the memory is statically allocated. The system is parameterized: the kernel and the user images are compiled separately and both are loaded by the bootloader, as explained in Section 8.1.2.

We have analyzed two versions:

- **beta**, a preliminary version where we found a vulnerability;
- **v1**, a more polished version with the vulnerability fixed and debug code removed.

The code segment of the kernel executable contains 329 functions (`objdump` reports around 10,000 instructions), shared between the kernel and the core unprivileged services. Finally, Krono-Safe provided us with a sample user image containing a number of applications, as well as the associated interface.

10.1.2 EducRTOS

Since, for some of our experiments (RQ0, RQ3–5), we needed access to the kernel source code and control over its build process, which was impossible with Asterios for commercial reasons, a new embedded kernel called EducRTOS[1] was developed by Matthieu Lemerre and the author of this thesis. EducRTOS supports a variety of features such as different schedulers and dynamic thread creation, and its implementation targets a different architecture than Asterios, namely 32-bit x86. The memory protection mechanism used is segmentation, instead of pagination. EducRTOS is also being used to teach operating systems to master students. Depending on the included features, the kernel size ranges between 2,346 and 2,866 instructions.

10.1.3 Analysis implementation

We analyze kernels using the Binsec/Codex tool (described in Section 7.2) that we implemented, and following the kernel verification method with differentiated handling of boot and runtime code described in Section 9.4. The abstract domain used is similar to the combined shape–fixed memory domain $\mathbb{E}^\sharp$ described in Section 7.2.4, except that we did not use the retained and staged points-to predicates[2].

[1] Available at https://github.com/EducRTOS/EducRTOS.

[2] We had not yet developed the abstractions of retained and staged points-to predicates when we carried out these kernel verifications.

Since the analysis is performed on Binsec's intermediate representation [Dav+16], the analysis implementation is entirely independent from the hardware architecture, apart from the abstract attacker-controlled transition (Section 8.5.1). To implement this transition, we use a simple sound approximation consisting in setting to an arbitrary value any unprotected register or memory location. Its implementation takes 23 lines of OCaml for ARM and 62 lines for x86.

Because Asterios is a multicore kernel, we computed an approximation of the execution that is sound in all possible interleavings by proceeding as follows: we perform a first analysis of the boot code for each of the 4 CPUs (by changing the result returned by the `cpuid` assembly instruction), thus obtaining the set of memory addresses manipulated by at least two distinct CPUs. We then run the analysis again, considering that these regions contain a non-deterministic choice between the values from all CPUs (thereby approximating all interleavings), and verified that no new addresses were shared between CPUs (in which case we would have needed to run the analysis again, and so on until reaching a fixpoint). By the same method, we verify that two concurrent executions of the kernel runtime do not share any memory, and then analyze them independently.

Availability Binsec/Codex and EducRTOS are open-source; in addition, along with Nicole et al. [Nic+21] was published an artifact enabling to reproduce our results[3]. This artifact does not include Asterios, however, which is not freely distributable.

10.1.4 Experimental methodology

We performed our formal verification completely independently from Krono-Safe activities. In particular, we never saw the source code of Asterios, and our interactions with Krono-Safe engineers were limited to a general presentation of Asterios features. We ran all our analyses on a standard laptop with an Intel Xeon E3-1505M 3 GHz CPU with 32 GB RAM. All measurements of execution time or memory usage have been observed to vary by no more than 2 % across 10 runs. We took the mean value of the runs.

10.2 Soundness check

Our method is sound *in principle*, in that it proves by construction APE and ARTE only on kernels where these properties are true. Yet, this ultimately depends on the correct implementation of the static analysis; in this preliminary experiment, we want to empirically check that our tool indeed does not prove APE and ARTE on buggy kernels.

10.2.1 Protocol

We deliberately introduce 4 backdoors in EducRTOS by creating 4 new system calls that:

1. jump to an arbitrary code address with kernel privilege,
2. grant kernel privilege to user code segments,
3. write to an arbitrary address,
4. or modify the memory protection tables to cover parts of the kernel address space.

These backdoors can easily be exploited to gain control over the kernel. Additionally, we add 3 bugs possibly leading to crashes in existing system calls: a read at an arbitrary address, an illegal opcode error and a possible division by zero.

[3] The artifact is available at https://github.com/binsec/rtas2021_artifact and contains a copy of Binsec/Codex and EducRTOS. More information about the Binsec platform can be found on the website: https://binsec.github.io.

10.2.2 Results

For each of the added vulnerabilities, our analysis does not prove APE (it either reported a maximally imprecise result or an alarm, at the instruction where the error occurs); for each bug, it does not prove ARTE.

For instance, a write or read to a completely unknown address result in a maximally imprecise abstract state (precluding the verification of any property), whereas granting kernel privilege to a task, or a possible division by zero, triggers a specific alarm.

In most cases, the location of the precision loss or alarm allows to straightforwardly understand the nature and origin of the backdoor or bug. The only exceptions are modifying user code segments in an incorrect way (granting kernel privilege) and modifying the memory protection tables in a way that lets tasks access the kernel address space. In those cases, the alarm is located at the end of the kernel runtime, just after the code that gives control to a task. The user of the analysis must then track the cause of the incorrect configuration. To that end, the analysis outputs an over-approximated CFG of the kernel and a log giving the possible values of each register and memory location at each node in that CFG.

10.2.3 Conclusions

Binsec/Codex was able to detect all the privilege escalation vulnerabilities and runtime errors that we introduced.

Additional notes This corresponds to the experience that we had while developing EducRTOS: several times, we launched Binsec/Codex and discovered unintentional bugs, such as a wrong check on the number of syscalls, or writes to null pointers, that were not detected by testing. Our method even discovered a bitflip in a kernel executable that occurred when the file was copied.

10.3 Real-Life Effectiveness

10.3.1 Protocol

Our goal here is to evaluate the *effectiveness* of our parameterized verification method on an unmodified industrial kernel, measured by: (1) the fact that the *method does succeed* in computing a non-trivial invariant for the whole system, i.e., computes an invariant under precondition for the kernel runtime and checks that the user tasks establish the precondition; (2) the *precision* of the analysis, measured by the number of *alarms* (i.e. properties that the analyzer cannot prove); (3) the *effort* necessary to set up the analysis, measured by the number of lines of manual annotations; and (4) the *performance* of the analysis, measured in CPU time and memory utilization.

The bulk of the annotations (1,057 lines) consists in definitions of physical types, and was automatically generated from the sample user image, using debug information. Debug information is otherwise not used by the analysis (and the kernel executable does not contain any). Manual annotations consist in changing the definitions by adding refinement predicates, or information about the lengths of arrays. These annotations were reverse-engineered as the ones required for the kernel analysis to succeed with no remaining alarm (and also correspond to necessary requirements for the code to run without errors), but they could be obtained from the documentation or directly provided by the OS developers.

We consider the two versions of the Asterios kernel, beta and v1, and two configurations (i.e., sets of type definitions):

- *Generic* contains types and parameter invariants which must hold for all legitimate user images;

		Generic		*Specific*	
# type annotations	generated	1057			
	manual	57 (5.11%)		58 (5.20%)	
Kernel version		BETA	v1	BETA	v1
analysis of runtime	status	✓	✓	✓	✓
	time (s)	647	417	599	406
# alarms in runtime		1 **true error** 2 false alarms	1 false alarm	1 **true error** 1 false alarm	**0** ✓
base case checking	status	✓	✓	✓	✓
	time (s)	32	29	31	30
Proves APE and ARTE?		N/A	✗	N/A	✓

Table 10.1: Main verification results on Asterios

- *Specific* further assumes that the stacks of all user tasks in the image have the same size. This is the default for Asterios applications, and it holds on our case study.

10.3.2 Results

The main results are given in Table 10.1. The *Generic* annotations consist in only 57 lines of manual annotations, in addition to 1,057 lines that were automatically generated (i.e. 5 % of manual annotations, and a manual annotations per instruction ratio of 0.58 %). The *Specific* set of annotations adds one more line.

We report the time and outcome of the parameterized analysis of the kernel runtime (row "analysis of runtime"). In all cases, the analysis was able successfully compute a state property for all executions starting from the post-boot state.

When analyzing the BETA version with these annotations, only *3 alarms* are raised in the runtime:

- One is a **true vulnerability**: in the supervisor call entry routine (written in manual assembly), the kernel extracts the system call number from the opcode that triggered the call, sanitizes it (ignoring numbers greater than, or equal to, 7), and uses it as an index in a table to jump to the target system call function; but this table has only 6 elements (therefore indexed 0 to 5), and is followed by a string in memory. This off-by-one error allows an `svc 6` system call to jump to an unplanned (constant) location, which can be attacker-controlled in some user images. The error is detected as the target of the jump goes to a memory address whose content is not known precisely, and thus that we cannot decode.
- One is a false alarm caused by *debugging code* temporarily violating the shape constraints: the code writes the constant `0xdeadbeef` in a memory location that should hold a pointer to a user stack (yielding an alarm as we cannot prove that this constant is a valid address for this type), and that memory location is always overwritten with a correct value further in the execution[4].
- The last one is a false alarm caused by an imprecision in our analyzer when user stacks can have different sizes. The analysis abstracts task data using the type-based shape domain: the data for each task is an instance of some type `Task`, akin to type `Task` in our example kernel (see Figure 9.1). In Asterios, instances of `Task` store the context of a suspended task, including the stack, which is treated as an opaque array of bytes. The size of that array is stored in another field. However, the

[4] This false alarm could have been avoided using the staged points-to predicates; however, we had not developed the points-to predicate extensions at the time this experiment was carried out.

current design of physical types does not allow to specify a field as holding the size of an array, as discussed in Section 4.4.2. As a consequence, there are two alternatives to perform the analysis:

- Let the stack size of each task be unknown. This is what is done in the *Generic* set of annotations. This causes a false alarm as the kernel runtime manipulates the saved stacks, and the analysis is unable to verify that the accessses are in the bounds of the array, since its size is unknown.
- Force all stack sizes to be equal. The size is still not precisely known at the time of the analysis, but it is the same for all saved contexts. This situation can be precisely encoded in physical types by using a global symbolic variable to represent the stack size. This is what is done in the *Specific* set of annotations.

Extending our type system with a form of dependent types, as discussed Section 4.4.2, would allow to lift this limitation.

When analyzing the v1 version, the first two alarms disappear, and no new alarm is added. Analyzing the kernel with the *Specific* annotations makes the last alarm disappear. In all cases, base case checking succeeds.

Analyzing the v1 kernel with the *Specific* annotations allows to reach 0 alarms, meaning that we have formally verified APE and ARTE.

The analysis time is almost unaffected by the small change in the kernel code, or in the type definitions given in parameter of the type-based domain. The analysis time is less than 11 minutes for the parameterized analysis, and 35 seconds for base case checking.

10.3.3 Conclusions

This experiment shows that *it is feasible to verify absence of privilege escalation of an industrial microkernel automatically, with a very small amount of manual annotations*, and *without any change to the original kernel.* In particular:

- The analysis is *effective*, in that it identifies real errors in the code, and verifies their absence once removed;
- The analysis is *precise*, as we were able to reach 0 false alarms on the correct code, and had no more than 2 false alarms on each configuration of the analysis;
- The annotation burden is very small (58 simple lines), as the kernel invariant is computed automatically and most of the type annotations are automatically converted from the interface types;
- Finally, the *analysis time*, for a kernel whose size is typical for embedded microkernels, is *small* (between 406 and 647 seconds).

10.4 Evaluation of the method

10.4.1 Protocol

The goal of this experiment is to evaluate the influence of each component in our three-step method (Section 9.2), in particular:

1. whether our shape domain is needed,
2. what is the nature and impact of the shape annotations, and
3. whether differentiated handling of boot code is mandatory.

Annotations	No annotation		Generated		Minimal		Generic		Specific		Dedicated	
Generated	0		1057		1057		1057		1057		1057	
Manual	0		0		10		57		58		62	
	boot	runt.	boot	runt.	boot	runt.	boot	runt.	boot	runt.	boot	runt.
Analysis time (s)	✗	N/A	✗	N/A	342	394	195	222	187	219	151	203
# alarms	N/A	N/A	N/A	N/A	85	13	60	1	59	0	43	0
Base case checking	N/A		N/A		✓		✓		✓		✓	

Table 10.2: Impact of the methodology.

We experiment on the v1 kernel version, and report results for both the boot code and the runtime, using different sets of annotations with an increasing amount of annotations:

- *No annotation* (equivalent to having no shape domain);
- *Generated* annotations (without any manual annotations);
- *Minimal* adds 10 lines of manual annotations, mainly limiting the range of array indices to prevent overflows in pointer arithmetic. This is the minimal set of annotations that allows the analysis to infer a non-trivial state property, but not without emitting alarms (see results below).
- *Generic* and *Specific* are the annotations defined above in Section 10.3.1; The *Generic* configuration adds 47 lines indicating which pointer types or structure fields may be null, which fields hold array indices, and relating array lengths with memory locations holding these lengths. The *Specific* configuration adds the constraint that stack sizes must be equal.
- *Dedicated* hardcodes some parameters, such as the number of tasks, and sets them to the values present in our sample user image.

10.4.2 Results

Table 10.2 shows the result of this evaluation. The analysis does not succeed in finding an invariant without the shape domain or without manual annotations: the analysis is too imprecise and aborts when performing the parameterized analysis of the boot code. In the table, we denote this result as ✗ in the “boot” column. In all cases, the fatal imprecision is caused by the inability to resolve the destination address of a write, making the entire memory unknown.

When the parameterized analysis of the boot code succeeds in yielding an abstract state $\mathbb{s}_{\text{booted}}$ (see Section 9.4), column “runt.” contains the time and number of alarms for the parameterized analysis of the kernel runtime.

Only 10 lines of manual annotations are necessary for the analysis to yield a non-trivial invariant (*Minimal*), albeit with many alarms in both boot code and runtime. The *Generic* configuration eliminates most alarms in the runtime, but 60 alarms remain in the boot code. The *Specific* annotations reach 0 alarms in runtime, but still 59 alarms in boot code. The *Dedicated* set of annotations is an attempt to remove the remaining alarms at the cost of parameterization, by specializing the analysis to our sample user tasks. This did reduce the number of alarms, but we could not eliminate them completely.

10.4.3 Conclusions

Parameterized verification of the kernel cannot be done without the shape domain. The ability to extract the shape configuration from types is extremely useful, as 95 % of the annotations are automatically extracted, requiring only 57 lines of manual annotations. Finally, *differentiated handling of boot code is necessary* as, firstly, the boot code is much harder to analyze than the runtime, and secondly, the shape invariants holds only after boot code terminates.

10.5 Genericity

10.5.1 Protocol

The fact that we successfully applied our method to verify Asterios and EducRTOS, which are two kernels running on different architectures (resp. ARM and x86) and memory protection mechanisms (resp. segmentation and pagination), shows that the approach is not tied to a specific hardware.

Further, to evaluate the dependence of the method on specific compilation toolchains or code patterns, we also performed a parameterized verification of 96 EducRTOS variants. Each variant was compiled with a different valuation of the following parameters:

- Compiler: either GCC 9.2.0 or Clang 7.1.0.
- Optimization flags: `-O1`, `-O2`, `-O3` or `-Os`.
- Scheduling algorithms: round-robin, fixed-priority, or earliest-deadline-first scheduling.
- Dynamic thread creation: enabled or disabled.
- Debug printing: enabled or disabled.

We then applied the parameterized verification method with differentiated handling of boot and runtime code. We also performed base case checking on a sample user image containing two tasks.

10.5.2 Results

The detailed results for each variant are provided in Appendix A. We could *parametrically* verify all 96 EducRTOS variants. The verification requires 98 lines of type definitions (which we extracted automatically, as we did for Asterios, from debug information) and 12 to 14 lines of manual annotations (depending on the scheduler), with 12 lines being common to all variants; corresponding to an annotation per instruction ratio of less than 0.59 %. The verification of each variant takes between 1.6 s and 73 s.

10.5.3 Conclusions

Our method is applicable to different kernel features and hardware architectures. It is robust: non-trivial changes in the kernel executable, like introduction of new features, or change in code patterns due to changes in the compilation process, do not not require to change the configuration. This robustness makes it an interesting tool for verifying kernels coming in many variants.

10.6 Automation and Scalability

10.6.1 Protocol

In this experiment, we perform a fully automated in-context verification (with *no annotation*) of a simple variant of the EducRTOS kernel with round-robin scheduling and no dynamic task creation. We then use this kernel to evaluate the scalability of this approach by modifying the number of tasks that run on the system, and compare it against our parameterized method.

Figure 10.1 gives the analysis time and memory usage of both approaches (parameterized and in-context verification) for various numbers of tasks on the system. For the parameterized analysis, the invariant computation takes less than a second and requires only 12 lines of annotations. In this case, the only part that depends on the number of tasks is the base case checking, and more precisely and the in-context static analysis of the almost-deterministic boot code.

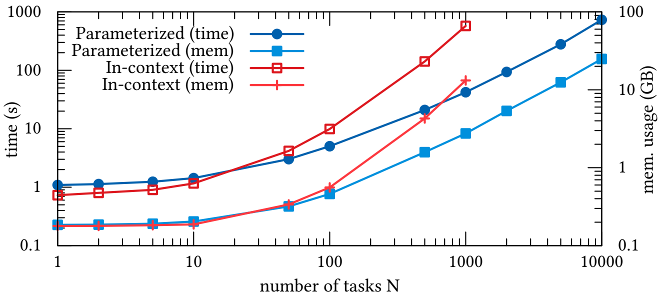


Figure 10.1: Performance when verifying a system with N tasks.

The parameterized analysis time seems to behave asymptotically as O($N^{1.2}$). This entails that multiplying the number of tasks by 10 multiplies the analysis time by a bit more than 15; and multiplying the number of tasks by 100 would multiply the analysis time by about 250. Therefore, for practical task counts (i.e. less than 100,000), the analysis time remains small.

By contrast, the in-context verification takes O(N^2) time which is much less practical. For example, the in-context verification with 10,000 tasks (which we could not run, not having enough RAM available) is expected to take about 1 hour 40 minutes, as opposed to 12 minutes with the parameterized verification. We interpret that by the fact that the number of steps to reach a fixpoint, the number of modified memory locations, and the number of target locations for the `Thread*` pointers grow with the number of tasks, resulting in quadratic complexity. This scalability issue is inherent to the use of in-context verification [Nor20].

Finally, the memory used by the in-context analysis of the boot code grows with the number of tasks. It should not be the case, since the analysis computes one abstract state per code location, and the number of code locations does not depend on the number of tasks. This is due to an implementation issue in our analyzer, caused by the fact that old abstract states are not garbage-collected because they never become unreachable. Although the growth is less than linear, it makes the parameterized analysis impractical beyond 10,000 tasks; it should therefore be fixed in the future.

10.6.2 Conclusions

In-context verification with no annotation—and thus fully automated—is achievable on very simple kernels, but is not robust enough for more complex kernels. Moreover, it does not scale to user images with very large numbers of tasks. On the contrary, parameterized verification (with few annotations) is robust and scales almost linearly to large numbers of tasks.

Chapter 11

Comparison with existing works on system and OS verification

Outline of the current chapter

The kernel verification method presented in this thesis is inscribed in a long history of using formal methods to verify operating systems. We give here a comprehensive overview of prior verification works. In Section 11.1, we give a number of evaluation criteria which we use to position our method among the existing works. Section 11.2 presents the relevant verification efforts in chronological order. Table 11.1 gives a synthetic overview of the discussion.

11.1 Classification and positioning

Besides large well-known monolithic kernels (e.g., Linux, Windows, *BSD) whose size and complexity are currently out of reach of formal verification, there is a rich ecosystem of small-size kernels found in many industrial applications, some of them security- or safety-critical. This includes security-oriented kernels like separation kernels [Rus81], microkernels [Kle+09], exokernels [EKJ95] and security-oriented hypervisors [Vas+16] or enclave software [Fer+17]; but also kernels used in embedded systems, for example in microcontrollers [Lev+17], real-time [RS94] or safety-critical [Ric10] operating systems.

11.1.1 Degree of automation

We can distinguish three classes of verification methods:

Table 11.1: Comparison of kernel verification efforts.

Target kernel	Verified property		Verification technique					Case study			
	Verified property	Implies APE?	Degree of automation	Verif. Level	Parame-terized	Multi-core	Infers invariants	Manual Annotations (LoC)	Unproved code (LoC)	Non-invasive	Analysis time (s)
UCLA Secure Unix [WKP80]	Compliance with specification	✓	Manual	Source	✓	✗	✗	N/A	80%	✗	N/A
Kit [Bev89]	Compliance with specification	✓	Manual	**Machine**	✗	✗	✗	1,020 definitions + 3561 lemmas	0 ✓	✗	N/A
seL4 [**sewellTranslationValidationVerified2013a**; Kle+09; Kle+14]	Compliance with specification	✓[a]	Manual	Source[g]	✓	✗	✗	200,000	1,200 (C, boot) + 500 (asm)	✗	N/A
Baby hypervisor [Alk+10; PSS12]	Compliance with specification	✓	Semi automated	Source+ assembly	✓	✗	✗	8,200 tokens	0 ✓	✗	4,571
Prosper [DGN13; Dam+13]	Compliance with specification	✓	Semi automated	**Machine**	✗[i]	✗	✗	6,400 [c]	0 ✓	✗	≤ 28,800
µC/OS-II [Xu+16]	Compliance with specification	✓[a]	Manual	Source	✓	✗[f]	✗	34,887 [d]	37%	✓	57,600[e]
CertiKOS [Gu+16]	Compliance with specification	✓	Manual	Source[g]+ assembly	✓	✓	✗	100,000	0 ✓	✗	N/A
CertiKOS^s, Komodo^s [Nel+19]	Task separation	✓	Semi automated	**Machine**	✓	✗	✗	859 (CertiKOS^s) 1,462 (Komodo^s)	0 ✓	✗	**166** (C.^{s}) **766** (K.^{s})
Komodo [Fer+17]	Task separation	✓	Semi automated	Assembly	✓	✗	✗	18,655	0 ✓	✗	14,400
Linux KVM [Li+21a; Li+21b]	Task separation	✓	Manual	Source + assembly	✓	✓	✗	32,700	0 ✓	✗	N/A
Verve Nucleus [YH11]	Type safety	✓	Semi automated	Assembly	✓	✗	✗	4,309	0 ✓	✗	**272**
XMHF [Vas+13]	Memory integrity	✗[b]	Semi automated	Source	✗[h]	✓	✗	N/A	422 (C) + 388 (asm)	✗	**76**
üXMHF [Vas+16]	Security properties	✓	Semi automated	Source + assembly	✗[h]	✗	✗	5,544	0 ✓	✗	3,739
Phidias [Nor20]	Absence of priv. escalation	✓	Semi automated	**Machine**	✗	✓	✗	unknown	0 ✓	✗	≤ 8,935
This work	Absence of priv. escalation	✓	**Fully automated**	**Machine**	✓	✓	✓	58✓	0 ✓	✓	**406**

[a] Assuming that the proof is completed to cover all the code. [b] Control flow integrity is assumed. [c] Generated from a 21,000 lines of HOL4 manual proof. [d] Plus 181,054 LoC of specification and support libraries. [e] The reported compilation time includes the support libraries. [f] The verification is concurrent because of in-kernel preemptions. [g] The translation to assembly is also verified. [h] The hypervisor supports a single guest [i] The hypervisor supports two guests

- *manual* [Bev89; Ric10; Gu+15; Xu+16; Kle+09; Li+21a]: These methods are based on assisted theorem proving. The tool user has to *provide for every program point a candidate invariant*, then use a proof assistant and guide the proof process to verify that every instruction preserves these candidate invariants;
- *semi-automated*[1] [Alk+10; YH11; Dam+13; Vas+16; Fer+17; Nel+19; Nor20]: the user has to provide the candidate invariants at some key program points (kernel entry and exit, loops, and optionally other program points like function entry and exit) and then use automated provers to verify that all paths between these points preserve the candidate invariants using deductive verification, symbolic execution or symbolic evaluation; Loop invariants, in particular, tend to require a lot of work; for this reason, semi-automated methods target kernels whose runtime does not contain complex loops [Dam+13; Nel+19; Nor20].
- *fully automated*: a sound static analyzer automatically infers correct invariants for every program point. The user only provides *invariant templates* by selecting or configuring the required abstract domains. This approach has been used to verify source code of critical embedded software [Bla+03], but never to verify a kernel.

In addition, all methods, regardless of their automation level, require a number of essential hypotheses for the proof to be valid. These hypotheses can include the hardware model, or properties of user tasks. For instance, in our work, the essential hypotheses are that our model of the hardware is correct and that the initialized state has a structure and contents such that our type annotations hold, which we verify during base case checking (see Section 9.2).

We demonstrate that abstract interpretation can automatically verify absence of privilege escalation and absence of runtime errors on embedded kernels from their executable with a very low annotation burden, sometimes with no manual intervention.

In the parameterized case, we verify kernels with a ratio of annotations per instruction which is at worst of 13 lines for 2346 instructions (0.59%). By comparison, the automated verification of CertiKOS^S [Nel+19] required 438 lines for 1845 instructions (23.7%).

11.1.2 Target property

Prior kernel verification methods generally target four kinds of program properties:

- *Functional correctness* [WKP80; Bev89; Kle+09; Alk+10; PSS12; Kle+14; Gu+15; Xu+16], i.e., compliance to a (manually written) formal specification of the kernel.
- *Task separation* [Rus81; Nel+19; Fer+17; Li+21a] i.e. verifying the absence of undesirable information flow between tasks.
- *Absence of privilege escalation* [YH11; Vas+16; Nor20] (also known as kernel integrity), i.e. proving that the kernel protects itself and that no attacker can gain control over it.
- *Absence of runtime errors* [Vas+16] like buffer overflows, null-pointer dereferences, or format string vulnerabilities.

To achieve maximal automation, we focus on *implicit* properties (Definition 8.6). Especially, we are the first to prove that absence of privilege escalation is implicit.

Note that the invariants we infer could significantly reduce the number of annotations required to verify functional correctness [Nel+19], and can generally help any other analysis; for instance worst-case execution time (WCET) estimation [Sch+19; CP01] requires knowledge about the CFG and memory accesses. Stronger invariants can be inferred by combining our analysis with other domains.

[1] Some authors call these techniques automated; we use the word semi-automated to emphasize the difference with fully-automated methods.

11.1.3 Trusted computing base (TCB) and verification comprehensiveness

We only trust that the bootloader correctly loads the ELF image in memory, that the hardware complies with its specification, and that our abstract interpreter (which has no dependencies) is sound. We do not trust the build toolchain (compiler, build scripts, assembler and linker), we analyze all of the code, and we do not assume any unverified hypothesis.

While push-button methods can verify all of the code [DGN13; Nel+19; Nor20], more manual methods often [Xu+16; WKP80; Kle+14; Gu+16] leave parts of the code unverified when the verification is hard or overly tedious. While source-level verifications methods sometimes carry to the assembly level [**sewellTranslationValidationVerified2013a**] or include support for assembly instructions [PSS12; Vas+16], they usually trust the compilation, assembly and linking phases. Unlike ours, the proof of the seL4 kernel trusts the boot code (about 1.2 kLOC) to bring the kernel in the expected state [Kle+14]. However, the property that they prove is stronger, and unlike ours it is not implied by any non-trivial state invariant (Theorem 8.3), and therefore the proof is much harder to extend to the entire kernel.

To further reduce our TCB, we would have to verify our static analyzer in a proof language with a small kernel (like Isabelle [NPW02] or Coq [CH88]), a huge effort that has been shown to be feasible [Jou+15].

11.1.4 Features of verified kernels

We focus on embedded systems kernels, where the kernel memory is mostly statically allocated (either in the kernel or in the user image) (e.g. [RS94; MF11; Ric10; DGN13; Nel+19; Nor20]). We do not consider some real-time kernels in which memory protection is absent or optional, as verifying APE on them would require unchecked assumptions on applicative code. Still, the kernels that we have analyzed feature complex code, including dynamic thread creation and dynamic memory allocation using object pools; different memory protection tables (using x86 segments or ARM page tables) modified at boot and runtime; different real-time schedulers working on arbitrary numbers of tasks; boot-time parameterization by a user image; and usage of multiple cores with shared memory.

Our method can handle kernels with features out of reach of prior fully automated techniques, such as real-time schedulers (which require unbounded loops). Rather than requiring kernels to be adapted to get around the limitations of the tool [Nel+19], *we are the first to verify an existing, unmodified real kernel.*

However, like any sound static analyzer, Binsec/Codex may still be too imprecise on some code patterns, emitting *false alarms* or failing to compute a non-trivial invariant. For instance, our tool would not handle well self-modification in the kernel, complex lock-free code, or using a general-purpose memory allocator outside of the boot code. Still, we can directly reuse progress in the automated analysis of these patterns. An important pattern for real-time systems is full preemptibility in the kernel. It should be possible to extend our analysis to these systems by leveraging our capacity to handle concurrent executions, notably by adding a stack abstraction that would be independent from the concrete address (as in [Rep+10]); this is an important topic for future work.

11.1.5 Verifying systems with unbounded memory

Prior works targeting machine code [Bev89; DGN13; Nel+19; Nor20] use a flat representation of the memory (where the abstract state enumerates all the memory cells in the kernel), causing scalability issues in the presence of a large number of tasks [Nor20] and preventing *parameterized* verification. To handle systems where the amount of memory is not statically known, we need more complex representations that *summarize* memory. Points-to and alias analyses are fast and easy to setup but are too imprecise for formal verification, and generally assume that the code behaves nicely, e.g., type-based alias analyses [DMM98] (see Section 2.3) assume that programs comply with the strict aliasing rule – while kernel codes often do not conform to the C standard [Kle+14]. On the other hand, shape analyses (see Section 2.2) can fully prove memory invariants, but require heavy configuration and generally cannot scale to a full embedded kernel.

We propose a lightweight type-based shape abstract domain that hits a middle ground: it is fast, precise, handles low-level behaviors (outside of the C standard) and requires little configuration. This is also the first time a shape analysis is performed on machine code.

Outside of fully-automated analyses, Walker et al. [WKP80] already observed in the 1980s that reasoning on type invariants is well suited to OS kernel verification. Several systems build around this idea [YH11; Fer+17], leveraging a dedicated typed language. Cohen et al. [Coh+09] describe a typed semantics for C with additional checks for memory typing preservation, similar to our own checks on memory accesses. While they use it in a deductive verification tool for C (to verify a hypervisor [Alk+10]), we build an abstract interpreter for machine code.

11.2 List of kernel verification efforts

We now present a comprehensive list of prior kernel verification works.

The first attempts to verify OS kernels were on the UCLA Secure Unix [WKP80] and the Provably Secure Operating System (PSOS) [FN79].

PSOS [FN79] was focused on kernel design and code proofs were not undertaken. Nevertheless, PSOS pioneered the concept of using abstraction layers in OS verification.

Bevier [Bev89] formally verified the functional correctness of a small operating system kernel, implemented in the machine code of an ideal hardware. Contrary to ours, their kernel was not parameterized (the number of tasks was fixed) and the proof was entirely manual. They proved a state invariant that implied security properties such as protection of the kernel from the tasks, and impossibility for a task to enter supervisor mode.

Heitmeyer et al. [Hei+08] manually verified a security policy (mostly consisting in separation of task data) a model of a security-oriented kernel for an embedded system using the PVS interactive prover, and provide a hand-written, non machine-checked proof of correspondence between this model and the source code. They identified that annotating the code to prove its security property is an important issue, and that methods to extract invariants from the code are needed for practical use.

Klein et al. [Kle+09] were the first to prove the correctness of an OS kernel, seL4, with respect to a high-level specification. Contrary to our work, strong functional properties are proved, and the kernel features dynamic task creation and memory reconfiguration; but the proof is entirely manual, the kernel was written for the purpose of verification, and the proof assumed correctness of the assembly code and of the compiler toolchain. The later work of **sewellTranslationValidationVerified2013a** extends the verification of seL4 by validating the compilation of their proven C code to machine code, thus removing the compiler toolchain from the trusted base. However, the manually-written assembly parts of the kernel are still trusted.

Yang and Hawblitzel [YH11] verified the Verve kernel, a single-processor kernel written in C#, using Hoare-style contracts on the kernel source code. The property verified is a form of type safety, which implies memory safety, but not absence of privilege escalation. Verve is built upon a “Nucleus” written in typed assembly language (TAL) whose type safety is verified automatically.

Paul, Schmaltz, and Shadrin [PSS12] completes the verification of the “baby hypervisor” of Alkassar et al. [Alk+10] (implemented on idealized hardware) by extending their deductive verification tool to support assembly code in addition to C code.

Dam et al. [Dam+13] and Dam, Guanciale, and Nemati [DGN13] have performed semi-automated machine-level verification of a security-oriented kernel, using deductive verification. Their focus is on proving correctness of a simplified kernel with regards to an executable specification. The verification is simplified by the fact that the kernel runtime does not contain any loops.

Gu et al. [Gu+15] verify the compliance to a functional specification of mCertiKOS, a simplified uniprocessor version of the CertiKOS kernel, using the Coq theorem prover. The approach is based on abstraction layers and a "contextual refinement" property, which allows the invariants to be simpler and the verification effort to be lighter and more easily adaptable than in the verification of seL4. The authors report a development time of less than 1 person-year, compared to 11 person-years for seL4. Gu et al. [Gu+16] extends the techniques developed in [Gu+15] to the concurrent context and verify mC2, a concurrent version of mCertiKOS. The authors report about 2 person-years for the verification of the concurrency features.

Vasudevan et al. [Vas+16] verify low-level security properties, such as memory separation and control-flow integrity, in some hypervisor modules, using a precise model of the hardware and assembly instructions. The majority of the hypervisor is verified using deductive verification (which requires annotations) and some assertions are left as runtime checks. In addition, the verification is performed on the source code, rather than machine code. A prior work [Vas+13] used a *fully automated* method, namely software model checking, to verify memory integrity (a property entailed by absence of privilege escalation). However, the process required manually decomposing this property into C assertions to be verified and assumed control-flow integrity. Some parts of the code (including the assembly code) were not formally verified.

Xu et al. [Xu+16] have verified functional correctness and priority-inversion-freedom on the C source code of key modules of an existing commercial OS kernel, using the Coq proof assistant. The main challenges of their work was discovering and verifying a loop invariant involving updates to Thread Control Blocks (like `Thread` in our example kernel, Figure 8.2); and the fact that, like ours, the OS pre-existing and written by an independent third party, and not written for the purpose of formal verification.

In a security context, enclaves are separated and encrypted regions for code and data providing a form of *physical security*, with hardware support. Ferraiuolo et al. [Fer+17] have verified noninterference in Komodo, a software implementation of SGX-like enclaves relying on minimal hardware support such as memory encryption. For that, they used deductive verification on an assembly-like language. They needed to trust an unverified, high-level specification of their software monitor.

Nelson et al. [Nel+19] ported CertiKOS [Gu+16] and Komodo [Fer+17] to the RISC-V architecture (the ports are named CertiKOS^s and Komodo^s), and applied their tool Serval to verify a number of invariants on the kernel. Those invariants imply task non-interference. Serval works by a mix of symbolic execution and bounded model checking. In order to perform the verification, they had to change the interface and implementations of these kernels. Their approach is limited to kernels with what they call a "finite interface", i.e. kernels that contain no unbounded loops. This restriction enables the verification process to be annotation-free, save for a specification, in the form of pre- and postconditions, of kernel runtime executions.

Nordholz [Nor20] proposes the Phidias hypervisor, whose core is devoid of any dynamic behaviour that is not strictly required at runtime. Most kernel data structures are statically instantiated. This lack of dynamicity enables Nordholz to verify APE using blunt symbolic execution. The author does not detail the amount of annotation necessary for the verification. As with Serval, the use of symbolic execution precludes hypervisors whose code contains unbounded loops.

Li et al. [Li+21a] retrofitted the Linux KVM hypervisor into a small core and a set of untrusted services, which enabled them to verify confidentiality and integrity (i.e. non-interference, up to declassification of data deliberately leaked by the VMs) of VM data manually using Coq. Li et al. [Li+21b] ported this proof to a more realistic hardware model, including a detailed model of the memory protection hardware and cache hierarchy.

Conclusion

This thesis aims to make the verification of memory properties, including memory safety, more effective on low-level and systems code. This issue is critical because violations of memory safety still account for a large portion of security vulnerabilities and software malfunctions. Yet, all modern software infrastructures still rely a lot on software written in memory-unsafe languages. In addition, developers still need to write in unsafe language or in assembly, either for performance or to access some hardware features directly (like in OS kernels).

The key to effective verification is to have a method which is efficient, both in terms of human effort and machine resources; which is sufficiently precise to succeed in verifying the property of interest on the program, and can handle the code patterns typical of systems code. For this we focused on automated verification methods (based on abstract interpretation, see Chapter 3), which are the most efficient in terms of human effort.

We introduced a memory abstraction based on types that express structural invariants on memory down to the byte level (Chapters 4 and 5). Such a type-based analysis offers a trade-off between precise, flow-sensitive shape analyses on the one end and scalable, flow-insensitive pointer analyses on the other. Simultaneously, it offers a good compromise between precision and automation: existing techniques without annotations are too imprecise; we trade some manual effort—namely, writing type definitions—for an important gain in precision.

However, this analysis alone is not sufficiently precise on some common programming patterns of low-level code; we add the abstractions of retained and staged points-to predicates (Chapter 6) to increase precision at a small cost. Our benchmarks show that these two abstractions are necessary to handle low-level C or machine code programs.

To demonstrate the effectiveness of our approach, we build two analyzers, for C and machine code (Chapter 7). In addition to the core technique, we added many ingredients necessary to handle low-level code, such as choosing the right numerical abstract domain, and, in the case of machine code, inferring control flow, delineating functions, and combining the type-based shape domain with a “flat model” abstraction. We demonstrate that our analyzer can prove the preservation of typing invariants (implying spatial memory safety) on low-level code, helping to eliminate an entire class of security vulnerabilities.

We then focused on the hard problem of verifying OS kernels, which represent the quintessence of low-level system programs. We applied our analysis of machine code to the executables of embedded kernels and showed that verifying type-based structural invariants enables to verify further properties, including absence of runtime errors (ARTE) and absence of privilege escalation (APE). We introduce the concept of implicit property (Chapter 8), which can be verified automatically, and prove that APE, like ARTE, is an implicit property.

We also explore the idea of automated parameterized verification (Chapter 9), i.e. verifying the system independently from the tasks, as kernels are usually compiled independently from the applicative code and data. Parameterized verification poses many challenges (such as the need to summarize memory, or the dependence on a complex precondition on the initial state), which are answered by our static analysis technique.

We applied our analysis on embedded kernels (Chapter 10), leading to the verification of ARTE and APE on a full, unmodified kernel with a very high level of automation.

However, our method is currently limited to those two properties, and cannot yet handle some complex kernel features. Although ARTE and APE are critical, it is desirable to progress towards the verification of stronger properties. Future research could focus on the following aspects:

- Using extensions to the type-based shape analysis to verify more advanced properties on kernels. For instance, dependent types as described in Section 4.4.2 would allow to express non-interference between tasks, by verifying that tasks have access to disjoint memory regions.
- Combining with other verification methods to verify stronger properties. Our method automatically infers a sound CFG (for machine code programs) and sound approximations of reachable states. This information can be valuable to guide less automated verification tools.
- Analyzing more complex kernels. Lifting some limitations, like the fact that the number and characteristics of the user tasks are static and fixed, would be beneficial.

Finally, the experiments that we conducted suggest that type-based shape analysis does is relevant not only on kernels, but also to verify useful properties, such as spatial memory safety, on any kind of program. Since effective verification of memory properties on low-level code is critical for the security and reliability of software, we believe that the analysis should be further improved to prove more properties on more programs. Specifically, the following directions could be explored:

- Support more language constructs. For example, tagged unions are currently treated very imprecisely by the analysis, because types cannot change dynamically depending on some value. Yet, tagged unions can be found in low-level programs. Support for them would be beneficial, maybe through the lens of sum types.
- Automated verification of more advanced properties. The type system of physical types can be extended to express richer properties (as discussed in Section 4.4.2): support could be added to express relations between structure fields; dependent types would permit to make types depend on values, e.g. to relate array with their lengths.
- Improving precision of the analysis. This point is related to previous one, as increasing precision often requires the ability to express stronger invariants. One direction would be to reduce the precision gap between our type-based analysis and shape analyses based on separation logic, by extending our abstraction with local separation predicates, e.g. expressing that the two children of a binary tree node are always separated. This would allow to reduce the number of spurious aliasing results and thus to improve precision on such structures.

Bibliography

[AHP18] Arthur Azevedo de Amorim, Cătălin Hriţcu, and Benjamin C. Pierce. “The Meaning of Memory Safety”. In: *Principles of Security and Trust - 7th International Conference (POST ’18).* Ed. by Lujo Bauer and Ralf Küsters. Lecture Notes in Computer Science. Thessaloniki, Greece: Springer International Publishing, 2018, pp. 79–105. URL: https://doi.org/10.1007/978-3-319-89722-6_4 (cit. on p. 1).

[Alk+10] Eyad Alkassar, Mark A. Hillebrand, Wolfgang Paul, and Elena Petrova. “Automated Verification of a Small Hypervisor”. In: *Verified Software: Theories, Tools, Experiments.* Ed. by Gary T. Leavens, Peter O’Hearn, and Sriram K. Rajamani. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 6217. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 40–54. URL: http://link.springer.com/10.1007/978-3-642-15057-9_3 (cit. on pp. 115, 140, 141, 143).

[And94] Lars Ole Andersen. “Program Analysis and Specialization for the C Programming Language”. University of Copenhagen, 1994 (cit. on pp. 3, 12).

[ASB17] Dennis Andriesse, Asia Slowinska, and Herbert Bos. “Compiler-Agnostic Function Detection in Binaries”. In: *IEEE European Symposium on Security and Privacy (EuroS&P ’17).* Paris, France: IEEE, 2017, pp. 177–189 (cit. on p. 98).

[Bal+05] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. “CodeSurfer/X86—A Platform for Analyzing X86 Executables”. In: *Compiler Construction, 14th International Conference (CC ’05), Held as Part of the Joint European Conferences on Theory and Practice of Software.* Lecture Notes in Computer Science. Edinburgh, United Kingdom: Springer, 2005, pp. 250–254. URL: https://doi.org/10.1007/978-3-540-31985-6_19 (cit. on p. 105).

[Bal07] Gogul Balakrishnan. “WYSINWYX: What You See Is Not What You Execute”. University of Wisconsin-Madison, 2007. URL: http://pages.cs.wisc.edu/~bgogul/Research/Thesis/bgogul.thesis.pdf (cit. on pp. 100, 105).

[Bev89] W.R. Bevier. “Kit: A Study in Operating System Verification”. In: *IEEE Transactions on Software Engineering* 15.11 (Nov. 1989), pp. 1382–1396. ISSN: 1939-3520 (cit. on pp. 115, 140–143).

[Bla+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “A Static Analyzer for Large Safety-Critical Software”. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI ’03).* Vol. 38. ACM, 2003, pp. 196–207. URL: https://doi.org/10.1145/781131.781153 (cit. on p. 141).

[Bou93] François Bourdoncle. "Efficient Chaotic Iteration Strategies with Widenings". In: *Proceedings of the International Conference on Formal Methods in Programming and Their Applications (FMPA '93)*. Ed. by Dines Bjørner, Manfred Broy, and Igor V. Pottosin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 128–141 (cit. on pp. 90, 105).

[BR06] Gogul Balakrishnan and Thomas Reps. "Recency-Abstraction for Heap-Allocated Storage". In: *Static Analysis, 13th International Symposium (SAS '06)*. Ed. by Kwangkeun Yi. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 4134. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 221–239. URL: http://link.springer.com/10.1007/11823230_15 (cit. on pp. 85, 105).

[Bro09] Neil Brown. "Linux Kernel Design Patterns - Part 2". In: *Linux Weekly News* (June 12, 2009). URL: https://lwn.net/Articles/336255/ (cit. on p. 35).

[BS16] George Balatsouras and Yannis Smaragdakis. "Structure-Sensitive Points-To Analysis for C and C++". In: *Static Analysis - 23rd International Symposium (SAS '16)*. Ed. by Xavier Rival. Vol. 9837. Lecture Notes in Computer Science. Edinburgh, United Kingdom: Springer Berlin Heidelberg, 2016, pp. 84–104. URL: http://link.springer.com/10.1007/978-3-662-53413-7_5 (cit. on p. 16).

[Cal+11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. "Compositional Shape Analysis by Means of Bi-Abduction". In: *Journal of the ACM* 58.6 (Dec. 2011), pp. 1–66. ISSN: 0004-5411, 1557-735X. URL: https://dl.acm.org/doi/10.1145/2049697.2049700 (cit. on pp. 3, 15).

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. The 4th ACM SIGACT-SIGPLAN Symposium. Los Angeles, California: ACM Press, 1977, pp. 238–252. URL: http://portal.acm.org/citation.cfm?doid=512950.512973 (cit. on pp. 2, 22, 24, 89, 90).

[CC79] Patrick Cousot and Radhia Cousot. "Systematic Design of Program Analysis Frameworks". In: *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages (POPL '79)*. Ed. by Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen. San Antonio, Texas, USA: ACM Press, 1979, pp. 269–282 (cit. on p. 26).

[CE82] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic". In: *Logics of Programs*. Ed. by Dexter Kozen. Vol. 131. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer-Verlag, 1982, pp. 52–71. URL: http://link.springer.com/10.1007/BFb0025774 (cit. on p. 2).

[CH00] Ben-Chung Cheng and Wen-Mei W. Hwu. "Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, and Evaluation". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. Vancouver, British Columbia, Canada: ACM Press, 2000, pp. 57–69. URL: http://portal.acm.org/citation.cfm?doid=349299.349311 (cit. on p. 13).

[CH78] Patrick Cousot and Nicolas Halbwachs. "Automatic Discovery of Linear Restraints among Variables of a Program". In: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '78)*. Tucson, Arizona: ACM Press, 1978, pp. 84–96. URL: http://portal.acm.org/citation.cfm?doid=512760.512770 (cit. on p. 24).

[CH88] Thierry Coquand and Gérard P. Huet. "The Calculus of Constructions". In: *Inf. Comput.* 76.2/3 (1988), pp. 95–120 (cit. on p. 142).

[Cha+20] Bor-Yuh Evan Chang, Cezara Drăgoi, Roman Manevich, Noam Rinetzky, and Xavier Rival. "Shape Analysis". In: *Foundations and Trends in Programming Languages* 6.1–2 (2020), pp. 1–158. ISSN: 2325-1107, 2325-1131 (cit. on p. 68).

[Coh+09] Ernie Cohen, Michał Moskal, Stephan Tobies, and Wolfram Schulte. "A Precise Yet Efficient Memory Model For C". In: *Electronic Notes in Theoretical Computer Science* 254 (Oct. 2009), pp. 85–103. ISSN: 15710661. URL: https://linkinghub.elsevier.com/retrieve/pii/S1571066109004150 (cit. on p. 143).

[Cou77] Patrick Cousot. *Asynchronous Iterative Methods for Solving a Fixed Point System of Monotone Equations in a Complete Lattice.* RR-88. Laboratoire IMAG, Université scientifique et médicale de Grenoble, Sept. 1977 (cit. on p. 90).

[CP01] Antoine Colin and Isabelle Puaut. "Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System". In: *13th Euromicro Conference on Real-Time Systems (ECRTS '01).* Delft, Netherlands: IEEE Computer Society, 2001, pp. 191–198 (cit. on p. 141).

[CR08] Bor-Yuh Evan Chang and Xavier Rival. "Relational Inductive Shape Analysis". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08).* Vol. 43. San Francisco, USA: ACM, 2008, pp. 247–260. URL: https://doi.org/10.1145/1328438.1328469 (cit. on pp. 3, 14).

[CR13] Bor-Yuh Evan Chang and Xavier Rival. "Modular Construction of Shape-Numeric Analyzers". In: Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of His Sixtieth Birthday. Vol. 129. EPTCS. Manhattan, Kansas, USA, Sept. 19, 2013. URL: https://doi.org/10.4204/EPTCS.129.11 (cit. on p. 25).

[CR99] Satish Chandra and Thomas Reps. "Physical Type Checking for C". In: *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '99).* Toulouse, France: ACM Press, 1999, pp. 66–75. URL: http://portal.acm.org/citation.cfm?doid=316158.316183 (cit. on pp. 4, 16, 68, 69).

[CRL99] Ramkrishna Chatterjee, Barbara G. Ryder, and William Landi. "Relevant Context Inference". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99).* Ed. by Andrew W. Appel and Alex Aiken. San Antonio, TX, USA: ACM, 1999, pp. 133–146 (cit. on p. 13).

[Cyt+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991), pp. 451–490 (cit. on p. 13).

[Dam+13] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. "Formal Verification of Information Flow Security for a Simple Arm-Based Separation Kernel". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13).* Berlin, Germany: ACM Press, 2013, pp. 223–234. URL: http://dl.acm.org/citation.cfm?doid=2508859.2516702 (cit. on pp. 6, 112, 120, 140, 141, 143).

[Das00] Manuvir Das. "Unification-Based Pointer Analysis with Directional Assignments". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00).* Vancouver, British Columbia, Canada: ACM Press, 2000, pp. 35–46. URL: http://portal.acm.org/citation.cfm?doid=349299.349309 (cit. on p. 12).

[Dav+16] Robin David, Sebastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. "BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis". In: *IEEE 23rd International Conference on Software Analysis, Evolution and Reengineering (SANER '16).* Suita: IEEE, Mar. 2016, pp. 653–656. URL: http://ieeexplore.ieee.org/document/7476691/ (cit. on pp. 91, 132).

[Deu92] Alain Deutsch. "A Storeless Model of Aliasing and Its Abstractions Using Finite Representations of Right-Regular Equivalence Relations". In: *Proceedings of the 1992 International Conference on Computer Languages (ICCL'92).* Ed. by James R. Cordy and Mario Barbacci. Oakland, California, USA: IEEE Computer Society, 1992, pp. 2–13 (cit. on pp. 15, 69).

[Deu94] Alain Deutsch. "Interprocedural May-Alias Analysis for Pointers: Beyond k-Limiting". In: *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI '94).* Ed. by Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa. Orlando, Florida, USA: ACM, 1994, pp. 230–241 (cit. on pp. 15, 69).

[DGN13] Mads Dam, Roberto Guanciale, and Hamed Nemati. "Machine Code Verification of a Tiny ARM Hypervisor". In: *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices - TrustED '13.* The 3rd International Workshop. Berlin, Germany: ACM Press, 2013, pp. 3–12. URL: http://dl.acm.org/citation.cfm?doid=2517300.2517302 (cit. on pp. 117, 140, 142, 143).

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. "Type-Based Alias Analysis". In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98).* Montreal, Quebec, Canada: ACM Press, 1998, pp. 106–117. URL: http://portal.acm.org/citation.cfm?doid=277650.277670 (cit. on pp. 16, 40, 68, 69, 143).

[Dor+08] N. Dor, J. Field, D. Gopan, T. Lev-Ami, A. Loginov, R. Manevich, G. Ramalingam, T. Reps, N. Rinetzky, M. Sagiv, R. Wilhelm, E. Yahav, and G. Yorsh. "Automatic Verification of Strongly Dynamic Software Systems". In: *Verified Software: Theories, Tools, Experiments.* Ed. by Bertrand Meyer and Jim Woodcock. Vol. 4171. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 82–92. URL: http://link.springer.com/10.1007/978-3-540-69149-5_11 (cit. on p. 14).

[DOY06] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. "A Local Shape Analysis Based on Separation Logic". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06).* Ed. by Holger Hermanns and Jens Palsberg. Vol. 3920. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 287–302. URL: http://link.springer.com/10.1007/11691372_19 (cit. on pp. 3, 14, 15).

[DPV11] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. "Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic". In: *Computer Aided Verification - 23rd International Conference (CAV '11).* Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 372–378. URL: http://link.springer.com/10.1007/978-3-642-22110-1_29 (cit. on pp. 3, 14).

[DPV13] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. "Byte-Precise Verification of Low-Level List Manipulation". In: *Static Analysis - 20th International Symposium (SAS '13).* Ed. by Francesco Logozzo and Manuel Fähndrich. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 7935. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 215–237. URL: http://link.springer.com/10.1007/978-3-642-38856-9_13 (cit. on pp. 14, 15, 68).

[ECS20] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. “Is Rust Used Safely by Software Developers?” In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE ’20).* Seoul, South Korea: ACM, June 27, 2020, pp. 246–257. URL: https://dl.acm.org/doi/10.1145/3377811.3380413 (cit. on p. 2).

[EKJ95] Dawson R. Engler, M. Frans Kaashoek, and James W. O’Toole Jr. “Exokernel: An Operating System Architecture for Application-Level Resource Management”. In: *Proceedings of the Fifteenth ACM Symposium on Operating System Principles (SOSP ’95).* Ed. by Michael B. Jones. Copper Mountain Resort, Colorado, USA: ACM, 1995, pp. 251–266 (cit. on p. 139).

[Ell+18] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. “Checked C: Making C Safe by Extension”. In: *2018 IEEE Cybersecurity Development (SecDev ’18).* Cambridge, MA: IEEE, Sept. 2018, pp. 53–60. URL: https://doi.org/10.1109/SecDev.2018.00015 (cit. on pp. 17, 69).

[FB] Bryan Ford and Erich Stefan Boleyn. *Multiboot Specification.* URL: https://www.gnu.org/software/grub/manual/multiboot/multiboot.html (visited on 08/03/2021) (cit. on p. 112).

[Fer+17] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. “Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software”. In: *Proceedings of the 26th Symposium on Operating Systems Principles.* SOSP ’17: ACM SIGOPS 26th Symposium on Operating Systems Principles. Shanghai China: ACM, Oct. 14, 2017, pp. 287–305. URL: https://dl.acm.org/doi/10.1145/3132747.3132782 (cit. on pp. 115, 139–141, 143, 144).

[FN79] Richard J. Feiertag and Peter G. Neumann. “The Foundations of a Provably Secure Operating System (PSOS)”. In: *1979 International Workshop on Managing Requirements Knowledge (MARK ’19)* (1979). URL: https://ieeexplore.ieee.org/document/8817256 (cit. on p. 143).

[FP91] Tim Freeman and Frank Pfenning. “Refinement Types for ML”. In: Proceedings of the ACM SIGPLAN’91 Conference on Programming Language Design and Implementation (PLDI ’91). Ed. by David S. Wise. ACM, 1991, p. 10. URL: https://doi.org/10.1145/113445.113468 (cit. on p. 16).

[GH96] Rakesh Ghiya and Laurie J. Hendren. “Is It a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C”. In: *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’96).* Ed. by Hans-Juergen Boehm and Guy L. Steele Jr. St. Petersburg Beach, Florida, USA: ACM Press, 1996, pp. 1–15 (cit. on pp. 15, 16, 68).

[Gu+15] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. “Deep Specifications and Certified Abstraction Layers”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’15).* Vol. 50. ACM, 2015, pp. 595–608 (cit. on pp. 115, 141, 144).

[Gu+16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. “CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’16).* Vol. 16. Savannah, GA, USA, 2016, pp. 653–669. URL: https://dl.acm.org/doi/10.5555/3026877.3026928 (cit. on pp. 4, 140, 142, 144).

[Hab+12] Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. “Forest Automata for Verification of Heap Manipulation”. In: *Formal Methods in System Design* 41.1 (Aug. 2012), pp. 83–106. ISSN: 0925-9856, 1572-8102. URL: http://link.springer.com/10.1007/s10703-012-0150-8 (cit. on pp. 3, 15).

[Hei+08] Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John McLean. “Applying Formal Methods to a Certifiably Secure Software System”. In: *IEEE Trans. Software Eng.* 34.1 (2008), pp. 82–98 (cit. on p. 143).

[HL07] Ben Hardekopf and Calvin Lin. “The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code”. In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’07).* San Diego, California, USA: ACM Press, 2007, p. 290. URL: http://portal.acm.org/citation.cfm?doid=1250734.1250767 (cit. on p. 13).

[HL11] Ben Hardekopf and Calvin Lin. “Flow-Sensitive Pointer Analysis for Millions of Lines of Code”. In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO ’11).* Washington, DC, USA: IEEE Computer Society, 2011, pp. 289–298. URL: http://dl.acm.org/citation.cfm?id=2190025.2190075 (cit. on p. 13).

[Hol+13] Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. “Fully Automated Shape Analysis Based on Forest Automata”. In: *Computer Aided Verification - 25th International Conference (CAV ’13).* Ed. by Natasha Sharygina and Helmut Veith. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 8044. Lecture Notes in Computer Science. Saint Petersburg, Russia: Springer Berlin Heidelberg, 2013, pp. 740–755. URL: http://link.springer.com/10.1007/978-3-642-39799-8_52 (cit. on pp. 15, 16).

[ILR20] Hugo Illous, Matthieu Lemerre, and Xavier Rival. “Interprocedural Shape Analysis Using Separation Logic-Based Transformer Summaries”. In: *Static Analysis - 27th International Symposium (SAS ’20).* Ed. by David Pichardie and Mihaela Sighireanu. Vol. 12389. Lecture Notes in Computer Science. Virtual Event: Springer, 2020, pp. 248–273. URL: https://doi.org/10.1007/978-3-030-65474-0%5C_12 (visited on 01/22/2022) (cit. on p. 15).

[IsoC18] International Organization for Standardization (ISO). *The ISO C Standard (C17).* 2018. URL: https://www.iso.org/standard/74528.html (cit. on p. 90).

[Jon81] Hans B. M. Jonkers. “Abstract Storage Structures”. In: *Algorithmic Languages.* Ed. by de Bakker and van Vliet. IFIP, North Holland Publishing Company, 1981, pp. 321–343 (cit. on pp. 15, 69).

[Jou+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. “A Formally-Verified C Static Analyzer”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 247–259 (cit. on p. 142).

[Ken07] Andrew Kennedy. “Compiling with Continuations, Continued”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP ’07).* Ed. by Ralf Hinze and Norman Ramsey. ACM, 2007, pp. 177–190. URL: https://doi.org/10.1145/1291151.1291179 (cit. on p. 35).

[Kle+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. “seL4: Formal Verification of an OS Kernel”. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP ’09).* ACM, 2009, pp. 207–220 (cit. on pp. 2, 4, 115, 139–141, 143).

[Kle+14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. "Comprehensive Formal Verification of an OS Microkernel". In: *ACM Trans. Comput. Syst.* 32.1 (2014), 2:1–2:70 (cit. on pp. 140–143).

[KN19] Steve Klabnik and Carol Nichols. *The Rust Programming Language.* San Francisco: No Starch Press, 2019. 526 pp. (cit. on p. 2).

[KSV10] Jörg Kreiker, Helmut Seidl, and Vesal Vojdani. "Shape Analysis of Low-Level C with Overlapping Structures". In: *Verification, Model Checking, and Abstract Interpretation, 11th International Conference (VMCAI '10).* Ed. by Gilles Barthe and Manuel Hermenegildo. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 5944. Madrid, Spain: Springer Berlin Heidelberg, 2010, pp. 214–230. URL: http://link.springer.com/10.1007/978-3-642-11319-2_17 (cit. on p. 105).

[KZV09] Johannes Kinder, Florian Zuleger, and Helmut Veith. "An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries". In: *Verification, Model Checking, and Abstract Interpretation, 10th International Conference (VMCAI '09).* Ed. by Neil D. Jones and Markus Müller-Olm. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 214–228. URL: https://doi.org/10.1007/978-3-540-93900-9_19 (cit. on p. 104).

[Lat05] Chris Lattner. "Macroscopic Data Structure Analysis and Optimization". University of Illinois, 2005 (cit. on p. 16).

[LCR10] Vincent Laviron, Bor-Yuh Evan Chang, and Xavier Rival. "Separating Shape Graphs". In: *19th European Symposium on Programming (ESOP '10).* Ed. by Andrew D. Gordon. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 6012. Lecture Notes in Computer Science. Paphos, Cyprus: Springer Berlin Heidelberg, 2010, pp. 387–406. URL: http://link.springer.com/10.1007/978-3-642-11957-6_21 (cit. on pp. 25, 105).

[Lev+17] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Alexander Levis. "Multiprogramming a 64kB Computer Safely and Efficiently". In: *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17).* Shanghai, China: ACM, 2017, pp. 234–251 (cit. on p. 139).

[Li+17] Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival. "Semantic-Directed Clumping of Disjunctive Abstract States". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17).* Paris, France: ACM, 2017, p. 14. URL: https://doi.org/10.1145/3009837.3009881 (cit. on pp. 3, 14, 15, 101, 104).

[Li+21a] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. "A Secure and Formally Verified Linux KVM Hypervisor". In: *42nd IEEE Symposium on Security and Privacy (SP '21).* San Francisco, CA, USA: IEEE, 2021, pp. 1782–1799 (cit. on pp. 140, 141, 144).

[Li+21b] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. "Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor". In: *30th USENIX Security Symposium (USENIX Security '21).* Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 3953–3970. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/li-shih-wei (cit. on pp. 140, 144).

[LRC15] Huisong Li, Xavier Rival, and Bor-Yuh Evan Chang. “Shape Analysis for Unstructured Sharing”. In: *Static Analysis - 22nd International Symposium (SAS ’15).* Saint-Malo, France: Springer, 2015, pp. 90–108. URL: https://doi.org/10.1007/978-3-662-48288-9_6 (cit. on pp. 15, 68, 101).

[LW94] Barbara H. Liskov and Jeannette M. Wing. “A Behavioral Notion of Subtyping”. In: *ACM Transactions on Programming Languages and Systems* 16.6 (Nov. 1994), pp. 1811–1841. ISSN: 0164-0925, 1558-4593. URL: https://dl.acm.org/doi/10.1145/197320.197383 (cit. on p. 41).

[Mar+07] Mark Marron, Deepak Kapur, Darko Stefanovic, and Manuel Hermenegildo. “A Static Heap Analysis for Shape and Connectivity: Unified Memory Analysis: The Base Framework”. In: *Languages and Compilers for Parallel Computing, 19th International Workshop (LCPC ’06).* Ed. by George Almási, Călin Caşcaval, and Peng Wu. Vol. 4382. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 345–363. URL: http://link.springer.com/10.1007/978-3-540-72521-3_25 (cit. on p. 16).

[McC10] William Terrence McCloskey. “Practical Shape Analysis”. PhD thesis. University of California, Berkeley, 2010 (cit. on p. 14).

[MF11] Jan Tobias Mühlberg and Leo Freitas. “Verifying FreeRTOS: From Requirements to Binary Code”. In: *Proceedings of the 11th International Workshop on Automated Verification of Critical Systems (AVoCS ’11).* 2011. URL: https://lirias.kuleuven.be/retrieve/172784/ (cit. on p. 142).

[Mil19] Matt Miller. “Trends, Challenges and Strategic Shifts in the Software Vulnerability Mitigation Landscape” (BlueHat IL). Feb. 7, 2019. URL: https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf (cit. on p. 1).

[Min04] Antoine Miné. “Weakly Relational Numerical Abstract Domains”. PhD thesis. École Polytechnique, Dec. 2004. URL: https://pastel.archives-ouvertes.fr/tel-00136630/document (cit. on p. 25).

[Min06] Antoine Miné. “Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics”. In: *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’06).* New York, NY, USA: ACM, 2006, pp. 54–63. URL: http://doi.acm.org/10.1145/1134650.1134659 (cit. on pp. 99, 105).

[Min17] Antoine Miné. “Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation”. In: *Foundations and Trends in Programming Languages* 4.3-4 (2017), pp. 120–372. ISSN: 2325-1107, 2325-1131. URL: http://www.nowpublishers.com/article/Details/PGL-034 (cit. on pp. 24, 90).

[Mit14] Mitre Corporation. *CVE-2014-0160.* 2014. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160 (visited on 08/02/2021) (cit. on p. 1).

[Mit20] Mitre Corporation. *2020 CWE Top 25 Most Dangerous Software Weaknesses.* 2020. URL: http://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html (visited on 07/16/2021) (cit. on p. 1).

[Mit21a] Mitre Corporation. *CVE-2021-22555.* 2021. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=2021-22555 (visited on 08/02/2021) (cit. on p. 1).

[Mit21b] Mitre Corporation. *CVE-2021-3156.* 2021. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156 (visited on 08/02/2021) (cit. on p. 1).

[Mor+02] Greg Morrisett, James Cheney, Dan Grossman, Michael Hicks, and Yanling Wang. "Cyclone: A Safe Dialect of C". In: *Proceedings of the General Track: 2002 USENIX Annual Technical Conference (USENIX '02).* Monterey, California, USA: USENIX, 2002, p. 15. URL: http://www.usenix.org/publications/library/proceedings/usenix02/jim.html (cit. on pp. 17, 69).

[Nec+05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. "CCured: Type-Safe Retrofitting of Legacy Software". In: *ACM Transactions on Programming Languages and Systems* 27.3 (May 2005), pp. 477–526. ISSN: 0164-0925, 1558-4593 (cit. on pp. 17, 69).

[Nel+19] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. "Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19).* Huntsville, Ontario, Canada: ACM Press, 2019, pp. 225–242. URL: http://dl.acm.org/citation.cfm?doid=3341301.3359641 (cit. on pp. 6, 112, 117, 120, 140–142, 144).

[Nic+21] Olivier Nicole, Matthieu Lemerre, Sébastien Bardin, and Xavier Rival. "No Crash, No Exploit: Automated Verification of Embedded Kernels". In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '21). 2021 (cit. on pp. 7, 130, 132).

[NLR22] Olivier Nicole, Matthieu Lemerre, and Xavier Rival. "Lightweight Shape Analysis Based on Physical Types". In: *23rd International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '22).* 2022 (cit. on pp. 7, 100).

[Nor20] Jan Nordholz. "Design of a Symbolically Executable Embedded Hypervisor". In: *Fifteenth EuroSys Conference 2020 (EuroSys '20).* Heraklion, Greece, 2020, p. 16. URL: https://doi.org/10.1145/3342195.3387516 (cit. on pp. 6, 112, 117, 120, 138, 140–142, 144).

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* Vol. 2283. Lecture Notes in Computer Science. Springer, 2002 (cit. on p. 142).

[PSS12] Wolfgang J. Paul, Sabine Schmaltz, and Andrey Shadrin. "Completing the Automated Verification of a Small Hypervisor - Assembler Code Verification". In: *Software Engineering and Formal Methods - 10th International Conference (SEFM '12).* Ed. by George Eleftherakis, Mike Hinchey, and Mike Holcombe. Vol. 7504. Lecture Notes in Computer Science. Thessaloniki, Greece: Springer, 2012, pp. 188–202. URL: https://doi.org/10.1007/978-3-642-33826-7%5C_13 (cit. on pp. 140–143).

[QS82] J. P. Queille and J. Sifakis. "Specification and Verification of Concurrent Systems in CESAR". In: *International Symposium on Programming.* Ed. by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Red. by G. Goos, J. Hartmanis, W. Brauer, P. Brinch Hansen, D. Gries, C. Moler, G. Seegmüller, J. Stoer, and N. Wirth. Vol. 137. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 337–351. URL: http://link.springer.com/10.1007/3-540-11494-7_22 (cit. on p. 2).

[Rep+10] Thomas W. Reps, Junghee Lim, Aditya V. Thakur, Gogul Balakrishnan, and Akash Lal. "There's Plenty of Room at the Bottom: Analyzing and Verifying Machine Code". In: *Computer Aided Verification, 22nd International Conference (CAV '10).* Ed. by Tayssir Touili, Byron Cook, and Paul B. Jackson. Vol. 6174. Lecture Notes in Computer Science. Edinburgh, UK: Springer, 2010, pp. 41–56. URL: https://doi.org/10.1007/978-3-642-14295-6%5C_6 (cit. on p. 142).

[Rey02] J.C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science (LICS ’02).* Copenhagen, Denmark: IEEE Comput. Soc, 2002, pp. 55–74. URL: http://ieeexplore.ieee.org/document/1029817/ (cit. on p. 14).

[Ric10] Raymond J. Richards. “Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel”. In: *Design and Verification of Microprocessor Systems for High-Assurance Applications.* Ed. by David S. Hardin. Boston, MA: Springer US, 2010, pp. 301–322. URL: https://doi.org/10.1007/978-1-4419-1539-9_10 (cit. on pp. 115, 139, 141, 142).

[Ric53] H. G. Rice. “Classes of Recursively Enumerable Sets and Their Decision Problems”. In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366. ISSN: 0002-9947, 1088-6850. URL: https://www.ams.org/tran/1953-074-02/S0002-9947-1953-0053041-6/ (cit. on p. 2).

[RKJ10] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. “Low-Level Liquid Types”. In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’10).* Madrid, Spain: Association for Computing Machinery, 2010, pp. 131–144 (cit. on pp. 16, 68, 69).

[RS94] K. Ramamritham and J.A. Stankovic. “Scheduling Algorithms and Operating Systems Support for Real-Time Systems”. In: *Proceedings of the IEEE* 82.1 (Jan. 1994), pp. 55–67. ISSN: 1558-2256 (cit. on pp. 139, 142).

[Rus81] John M. Rushby. “Design and Verification of Secure Systems”. In: *Proceedings of the Eighth Symposium on Operating System Principles (SOSP ’81).* Ed. by John Howard and David P. Reed. Pacific Grove, California, USA: ACM, 1981, pp. 12–21. URL: https://doi.org/10.1145/800216.806586 (cit. on pp. 139, 141).

[SB15] Yannis Smaragdakis and George Balatsouras. “Pointer Analysis”. In: *Foundations and Trends in Programming Languages* 2.1 (2015), pp. 1–69. ISSN: 2325-1107, 2325-1131. URL: http://www.nowpublishers.com/article/Details/PGL-014 (cit. on p. 12).

[SBL11] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. “Pick Your Contexts Well: Understanding Object-Sensitivity”. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’11).* Austin, Texas, USA: ACM Press, 2011, p. 17. URL: http://portal.acm.org/citation.cfm?doid=1926385.1926390 (cit. on p. 12).

[Sch+19] Simon Schuster, Peter Wägemann, Peter Ulbrich, and Wolfgang Schröder-Preikschat. “Proving Real-Time Capability of Generic Operating Systems by System-Aware Timing Analysis”. In: *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS ’19).* Ed. by Björn B. Brandenburg. Montreal, Canada: IEEE, 2019, pp. 318–330 (cit. on p. 141).

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. “Parametric Shape Analysis via 3-Valued Logic”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’99).* San Antonio, Texas, United States: ACM Press, 1999, pp. 105–118 (cit. on pp. 3, 14, 15).

[Ste96] Bjarne Steensgaard. “Points-to Analysis in Almost Linear Time”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’96).* New York, NY, USA: ACM, 1996, pp. 32–41. URL: http://doi.acm.org/10.1145/237721.237727 (cit. on pp. 3, 12).

[SXX12] Lei Shang, Xinwei Xie, and Jingling Xue. "On-Demand Dynamic Summary-Based Points-to Analysis". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CHO '12).* San Jose, California: ACM Press, 2012, p. 264. URL: http://dl.acm.org/citation.cfm?doid=2259016.2259050 (cit. on p. 13).

[TvL84] Robert E. Tarjan and Jan van Leeuwen. "Worst-Case Analysis of Set Union Algorithms". In: *Journal of the Association for Computing Machinery* 31 (1984), pp. 245–281 (cit. on p. 35).

[Vas+13] A. Vasudevan, S. Chaki, Limin Jia, J. McCune, J. Newsome, and A. Datta. "Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework". In: *2013 IEEE Symposium on Security and Privacy (SP '13).* Berkeley, CA: IEEE, May 2013, pp. 430–444. URL: http://ieeexplore.ieee.org/document/6547125/ (cit. on pp. 140, 144).

[Vas+16] Amit Vasudevan, Sagar Chaki, Petros Maniatis, Limin Jia, and Anupam Datta. "überSpark: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor". In: 25th USENIX Security Symposium (USENIX Security 16). 2016, pp. 87–104. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/vasudevan (cit. on pp. 115, 139–142, 144).

[WKP80] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. "Specification and Verification of the UCLA Unix Security Kernel". In: *Communications of the ACM* 23.2 (Feb. 1980), pp. 118–131. ISSN: 0001-0782, 1557-7317. URL: https://dl.acm.org/doi/10.1145/358818.358825 (cit. on pp. 115, 140–143).

[WL95] Robert P. Wilson and Monica S. Lam. "Efficient Context-Sensitive Pointer Analysis for C Programs". In: *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI '95).* Ed. by David W. Wall. La Jolla, California, USA: ACM, 1995, p. 1 (cit. on p. 13).

[Xu+16] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. "A Practical Verification Framework for Preemptive OS Kernels". In: *Computer Aided Verification.* Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 59–79. URL: http://link.springer.com/10.1007/978-3-319-41540-6_4 (visited on 04/28/2021) (cit. on pp. 115, 140–142, 144).

[YH11] Jean Yang and Chris Hawblitzel. "Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System". In: *Communications of the ACM* 54.12 (Dec. 1, 2011), p. 123. ISSN: 00010782. URL: http://dl.acm.org/citation.cfm?doid=2043174.2043197 (cit. on pp. 115, 140, 141, 143).

Appendix A

Detailed verification results of all EducRTOS variants

We give here the detailed results of analyzing all 96 variants of EducRTOS as described in Section 10.5. In all cases, the parameterized verification was successful, as well as the base case checking on a sample user image containing two tasks. For each possible choice of compiler (Comp.), optimization flag (Opt.), scheduling algorithm (Sched.), enabling or disabling dynamic thread creation (DT), and enabling or disabling debug printing, we report the total analysis time, the number of (unique) instructions in the CFG computed by the analysis (#instr.), and the peak memory usage (Mem.).

Comp.	Opt.	Sched.	DT	Print.	Time (s)	#instr.	Mem. (MB)
gcc	-O1	RR	yes	no	1.6	2649	283
gcc	-O1	RR	yes	yes	40.5	2734	5704
gcc	-O1	RR	no	no	1.3	2640	253
gcc	-O1	RR	no	yes	40.5	2725	5671
gcc	-O1	FP	yes	no	7.1	2695	672
gcc	-O1	FP	yes	yes	72.6	2780	9287
gcc	-O1	FP	no	no	5.5	2686	559
gcc	-O1	FP	no	yes	70.2	2771	9072
gcc	-O1	EDF	yes	no	8.3	2711	753
gcc	-O1	EDF	yes	yes	72.3	2796	9185
gcc	-O1	EDF	no	no	6.3	2702	602
gcc	-O1	EDF	no	yes	70.9	2787	9259
clang	-O1	RR	yes	no	1.8	2400	325
clang	-O1	RR	yes	yes	26.9	2501	3608
clang	-O1	RR	no	no	1.5	2387	280
clang	-O1	RR	no	yes	26.3	2488	3598
clang	-O1	FP	yes	no	4.1	2476	518
clang	-O1	FP	yes	yes	28.9	2577	3792
clang	-O1	FP	no	no	4.7	2463	538
clang	-O1	FP	no	yes	45.5	2564	5696
clang	-O1	EDF	yes	no	4.6	2484	517

clang	-O1	EDF	yes	yes	29.5	2585	3802
clang	-O1	EDF	no	no	5.4	2471	575
clang	-O1	EDF	no	yes	46	2572	5689
gcc	-O2	RR	yes	no	1.7	2705	278
gcc	-O2	RR	yes	yes	41	2787	5705
gcc	-O2	RR	no	no	1.4	2642	258
gcc	-O2	RR	no	yes	40.1	2724	5695
gcc	-O2	FP	yes	no	7	2750	669
gcc	-O2	FP	yes	yes	72.8	2832	9091
gcc	-O2	FP	no	no	5.4	2687	555
gcc	-O2	FP	no	yes	70.5	2769	9120
gcc	-O2	EDF	yes	no	8.2	2766	726
gcc	-O2	EDF	yes	yes	73.3	2848	9108
gcc	-O2	EDF	no	no	6.3	2703	586
gcc	-O2	EDF	no	yes	70.9	2785	9168
clang	-O2	RR	yes	no	1.7	2361	299
clang	-O2	RR	yes	yes	26.4	2429	3566
clang	-O2	RR	no	no	1.3	2423	264
clang	-O2	RR	no	yes	25.8	2491	3557
clang	-O2	FP	yes	no	4	2429	472
clang	-O2	FP	yes	yes	28.6	2497	3737
clang	-O2	FP	no	no	4.5	2490	515
clang	-O2	FP	no	yes	44.8	2558	5704
clang	-O2	EDF	yes	no	4.5	2436	488
clang	-O2	EDF	yes	yes	29	2504	3747
clang	-O2	EDF	no	no	5.1	2498	537
clang	-O2	EDF	no	yes	45.4	2566	5715
gcc	-O3	RR	yes	no	1.7	2705	288
gcc	-O3	RR	yes	yes	29.9	2796	4232
gcc	-O3	RR	no	no	1.4	2642	258
gcc	-O3	RR	no	yes	29.2	2733	4281
gcc	-O3	FP	yes	no	7	2739	647
gcc	-O3	FP	yes	yes	52.5	2830	6805
gcc	-O3	FP	no	no	5.4	2676	550
gcc	-O3	FP	no	yes	51.2	2767	6795
gcc	-O3	EDF	yes	no	8.5	2761	742
gcc	-O3	EDF	yes	yes	53.1	2852	6850
gcc	-O3	EDF	no	no	6.2	2698	592
gcc	-O3	EDF	no	yes	51.9	2789	6678
clang	-O3	RR	yes	no	1.7	2486	300
clang	-O3	RR	yes	yes	25.4	2788	3516
clang	-O3	RR	no	no	1.4	2423	264
clang	-O3	RR	no	yes	25.2	2725	3401
clang	-O3	FP	yes	no	4	2556	470
clang	-O3	FP	yes	yes	45	2858	5691
clang	-O3	FP	no	no	4.6	2492	523
clang	-O3	FP	no	yes	43.4	2794	5419
clang	-O3	EDF	yes	no	4.6	2564	508
clang	-O3	EDF	yes	yes	45.5	2866	5658

clang	-O3	EDF	no	no	5.2	2501	545
clang	-O3	EDF	no	yes	44.5	2803	5438
gcc	-Os	RR	yes	no	1.7	2664	275
gcc	-Os	RR	yes	yes	32.4	2750	4612
gcc	-Os	RR	no	no	1.3	2652	244
gcc	-Os	RR	no	yes	31.8	2738	4514
gcc	-Os	FP	yes	no	6.6	2717	628
gcc	-Os	FP	yes	yes	55.1	2803	7063
gcc	-Os	FP	no	no	5.1	2705	529
gcc	-Os	FP	no	yes	54	2791	6930
gcc	-Os	EDF	yes	no	7.8	2730	699
gcc	-Os	EDF	yes	yes	57.1	2816	7302
gcc	-Os	EDF	no	no	5.8	2718	567
gcc	-Os	EDF	no	yes	54.2	2804	7062
clang	-Os	RR	yes	no	1.7	2368	312
clang	-Os	RR	yes	yes	20.7	2439	2864
clang	-Os	RR	no	no	1.4	2346	268
clang	-Os	RR	no	yes	20.3	2417	2841
clang	-Os	FP	yes	no	4.1	2437	484
clang	-Os	FP	yes	yes	23.1	2508	3046
clang	-Os	FP	no	no	4.5	2413	489
clang	-Os	FP	no	yes	34.2	2484	4320
clang	-Os	EDF	yes	no	4.7	2445	522
clang	-Os	EDF	yes	yes	23.7	2516	3022
clang	-Os	EDF	no	no	5.2	2423	552
clang	-Os	EDF	no	yes	34.9	2494	4408

Glossary

APE absence of privilege escalation. 4, 6, 7, 113, 114, 115, 116, 118, 119, 123, 125, 126, 127, 130, 132, 133, 135, 141, 145, 146

ARTE absence of runtime errors. 4, 6, 7, 112, 115, 117, 118, 119, 126, 127, 130, 132, 133, 135, 141, 145, 146

AST abstract syntax tree. 54, 88

CFG control flow graph. 89, 93, 94, 95, 96, 97, 98, 104, 105, 117, 133, 146

DAG direct acyclic graph. 14

MMU Memory Management Unit. 131

MPU Memory Protection Unit. 5

SLOC source lines of code: number of source code lines, excluding blank lines and comments. 88

TCB trusted computing base: set of components that are trusted, in the context of a security-critical system or a verification method. 131

UMA Unified Memory Analysis. 16

RÉSUMÉ

Les logiciels étant des composants essentiels de nombreux systèmes embarqués et de nombreux systèmes d'information, un dysfonctionnement logiciel peut entraîner d'importants dommages ou des failles de sécurité. De nouveaux bugs et de nouvelles vulnérabilités sont trouvés régulièrement dans les programmes existants; une grande partie d'entre eux est causeé par des violations de la sûreté mémoire. En particulier, le code bas niveau, écrit dans des langages de programmation qui offrent peu de garanties de sûreté, est le plus susceptible de contenir ce type de bug. Malgré cela, écrire dans un langage bas niveau reste parfois nécessaire pour des raisons de performance, ou pour accéder directement aux fonctionnalités du matériel. Les méthodes formelles peuvent permettre de vérifier la sûreté des programmes bas niveau, mais les techniques automatisées de vérification de propriétés mémoire, telles que les analyses de forme, nécessitent encore un effort manuel important, ce qui est un obstacle à une adoption large. Dans cette thèse, nous proposons une analyse automatisée facilement applicable, basée sur un système de types exprimant des invariants structurels sur la mémoire, précis jusqu'au niveau de l'octet. Cette analyse, que nous formalisons dans le cadre de l'interprétation abstraite, offre un compromis entre les analyses de forme, précises et sensibles au flot de contrôle, et les analyses de pointeurs, qui sont insensibles au flot de contrôle mais passent très bien à l'échelle. Elle peut être appliquée à du code bas niveau avec peu d'annotations manuelles. Nous montrons comment cette analyse basée sur les types peut être complémentée par des prédicats de pointeurs conservés et reportés, afin de supporter précisément des motifs fréquents en code bas niveau tels que l'initialisation de structures de données. Nous démontrons l'efficacité et l'applicabilité de l'analyse en vérifiant la conservation d'invariants structurels (qui impliquent la sûreté mémoire spatiale) sur des programmes C et du code machine, montrant qu'elle peut être utile pour éliminer toute une classe de failles de sécurité. Nous appliquons ensuite notre analyse à des exécutables de noyaux embarqués, et nous montrons que nos invariants à base de types permettent de vérifier l'absence d'erreurs à l'exécution et l'absence d'escalade de privilèges. Pour cela, nous introduisons le concept de propriété implicite, c'est-à-dire de propriété qui peut être définie sans référence à un programme en particulier, qui se prêtent bien à la vérification automatique; et nous montrons que l'absence d'escalade de privilèges est une propriété implicite. La vérification paramétrée, c'est-à-dire la vérification de noyaux indépendamment du code et des données des applications, comporte plusieurs défis, comme le besoin de résumer la mémoire ou bien la dépendance à une précondition complexe sur l'état initial. Nous proposons une méthodologie pour les résoudre à l'aide de notre technique d'analyse. À l'aide de cette méthodologie, nous vérifions l'absence d'erreurs à l'exécution et l'absence d'escalade de privilèges sur un noyau entier sans modification, avec un haut niveau d'automatisation.

MOTS CLÉS

analyse statique, analyse mémoire, vérification d’OS

ABSTRACT

As software is an essential component of many embedded systems or online information systems, its malfunction can cause harm or security vulnerabilities. New bugs and vulnerabilities keep being discovered in existing software; many of those bugs and vulnerabilities are caused by violations of memory safety. In particular, low-level code, written in languages that offer few safety guarantees, is the most prone to this kind of bug. However, writing low-level code is sometimes necessary for performance or direct access to hardware features. Formal methods can be used to verify the safety of low-level programs, but automated analysis techniques to verify memory-related properties, such as shape analyses, still require important human effort, preventing wide adoption. In this thesis, we propose a practical automated analysis based on types that express structural invariants on memory down to the byte level. This analysis, which we formalize in the framework of abstract interpretation, offers a trade-off between precise, flow-sensitive shape analyses and scalable, flow-insensitive pointer analyses. It can be applied to low-level code with only a small amount of manual annotations. We show how the type-based abstraction can be complemented with retained and staged points-to predicates to handle precisely common low-level code patterns, such as data structure initialization. We demonstrate the effectiveness and practicality of the analysis by verifying the preservation of structural invariants (implying spatial memory safety) on C and machine code programs, showing that it can be helpful in eliminating an entire class of security vulnerabilities. We then apply our analysis to executables of embedded kernels and show that our type-based invariants allow to verify absence of runtime errors and absence of privilege escalation. To do this, we introduce the concept of implicit properties, i.e. properties which can be defined without reference to a specific program, and therefore lend themselves well to automated verification; and we prove that absence of privilege escalation is an implicit property. Parameterized verification, i.e. verification of the kernel independently from applicative code and data, poses many challenges, such as the need to summarize memory, or the dependence on a complex precondition on the initial state. We propose a methodology to solve them using our analysis technique. We apply this methodology to verify absence of runtime errors and absence of privilege escalation on a full, unmodified embedded kernel with a high level of automation.

KEYWORDS

static analysis, memory analysis, OS verification