Analyse de la sécurité de systèmes critiques embarqués à forte composante logicielle par interprétation abstraite

Proving the security of software-intensive embedded systems by abstract interpretation

Soutenue par
Marc Chevalier
Le vendredi 27 novembre 2020

École doctorale n°386
Sciences Mathématiques de Paris Centre

Spécialité
Informatique

Abstract

THIS THESIS is dedicated to the analysis of low-level software, like operating systems, by abstract interpretation. Analyzing OSes is a crucial issue to guarantee the safety of software systems since they are the layer immediately above the hardware and that all applicative tasks rely on them. For critical applications, we want to prove that the OS does not crash, and that it ensures the isolation of programs, so that an untrusted program cannot disrupt a trusted one.

The analysis of this kind of programs raises specific issues. This is because OSes must control hardware using instructions that are meaningless in ordinary programs. In addition, because hardware features are outside the scope of C, source code includes assembly blocks mixed with C code. These are the two main axes in this thesis: handling mixed C and assembly, and precise abstraction of instructions that are specific to low-level software. This work is motivated by the analysis of a case study emanating from an industrial partner, which required the implementation of proposed methods in the static analyzer ASTRÉE.

The first part is about the formalization of a language mixing simplified models of C and assembly, from syntax to semantics. This specification is crucial to define what is legal and what is a bug, while taking into account the intricacy of interactions of C and assembly, in terms of data flow and control flow.

The second part is a short introduction to abstract interpretation focusing on what is useful thereafter.

The third part proposes an abstraction of the semantics of mixed C and assembly. This is actually a series of parametric abstractions handling each aspect of the semantics.

The fourth part is interested in the question of the abstraction of instructions specific to low-level software. Interest properties can easily be proven using ghost variables, but because of technical reasons, it is difficult to design a reduced product of abstract domains that allows a satisfactory handling of ghost variables. This part builds such a general framework with domains that allow us to solve our problem and many others.

The final part details properties to prove in order to guarantee isolation of programs that have not been treated since they raise many complicated questions. We also give some suggestions to improve the product of domains with ghost variables introduced in the previous part, in terms of features and performances.

Résumé

CETTE THÈSE est consacrée à l'analyse de logiciels de bas niveau, tels que les systèmes d'exploitation, par interprétation abstraite. L'analyse des OS est une question importante pour garantir la sûreté des systèmes logiciels puisqu'ils forment le niveau immédiatement au dessus du matériel et que toutes les tâches applicatives dépendent d'eux. Pour des applications critiques, on veut s'assurer que l'OS ne plante pas, mais aussi qu'il assure l'isolation des programmes, de sorte qu'un programme dont la fiabilité n'a pas été établie ne puisse perturber un programme de confiance.

L'analyse de ce genre de programmes soulève des problèmes spécifiques. Cela provient du fait que les OS doivent contrôler le matériel avec des opérations qui n'ont pas de sens dans un programme ordinaire. De plus, comme les fonctionnalités matérielles sont en dehors du for du C, le code source contient des blocs de code assembleur mêlés au C. Ce sont les deux axes de cette thèse : gérer les mélanges de C et d'assembleur, et abstraire finement les opérations spécifiques aux logiciels de bas niveau. Ce travail est guidé par l'analyse d'un cas d'étude d'un partenaire industriel, ce qui a nécessité l'implémentation des méthodes proposées dans l'analyseur statique ASTRÉE.

La première partie s'intéresse à la formalisation d'un langage mélangeant des modèles simplifiés du C et de l'assembleur, depuis la syntaxe jusqu'à la sémantique. Cette spécification est importante pour définir ce qui est légal et ce qui constitue une erreur, tout en tenant compte de la complexité des interactions du C et de l'assembleur, tant en termes de données que de flot de contrôle.

La seconde partie est une introduction sommaire à l'interprétation abstraite qui se limite à ce qui est utile par la suite.

La troisième partie propose une abstraction de la sémantique des mélanges de C et d'assembleur. Il s'agit en fait d'une collection d'abstractions paramétriques qui gèrent chacun des aspects de cette sémantique.

La quatrième partie s'intéresse à l'abstraction des opérations spécifiques aux logiciels de bas niveau. Les propriétés d'intérêt peuvent être facilement prouvées à l'aide de variables fantômes, mais pour des raisons techniques, il est difficile de concevoir un produit réduit de domaines abstraits qui supporte une gestion satisfaisante des variables fantômes. Cette partie construit un tel cadre très général ainsi que des domaines qui permettent de résoudre beaucoup de problèmes dont le nôtre.

L'ultime partie présente quelques propriétés à prouver pour garantir l'isolation des programmes, qui n'ont pas été traitées car elles posent de nouvelles et complexes questions. On donne aussi quelques propositions d'amélioration du produit de domaines avec variables fantômes introduit dans la partie précédente, tant en termes de fonctionnalités que de performances.

Contents

Chapter 1

Introduction

SOFTWARE SYSTEMS are everywhere. But can we trust them? Indeed, some software are in charge of important missions and failures can be very grievous, such programs are called critical. There are famous examples with various kind of consequences, for instance:

– privacy: HEARTBLEED bug in OpenSSL (2014)[*][†];
– money: maiden flight of ARIANE 5 launch vehicle (1996; \$370 million for the payload only [Dow97]);
– deaths and casualties: TOYOTA electronic throttle control system failure (2005; $\geqslant$ 89 deaths); surface-to-air (SAM) missile PATRIOT MIM-104 failure (1991; 28 deaths in its own army, not counting the deaths caused by the missile that the PATRIOT did not intercept) [GAO92]; THERAC-25 radiation therapy machine (1985–1987; 6 accidents) [LT93].

Even excluding such extreme cases whose cost cannot really be evaluated, NIST[‡] estimated in 2002 that software bugs cost U.S. economy between \$22.2 and \$59.9 billion annually [NIST02]; one may imagine how costly bugs are worldwide in 2020. Tests and documentation represent a vast majority of cost and efforts in critical software development, and still represent a decent part of most regular program development. Analyzing programs to detect errors is a crucial problem as it may avoid calamitous consequences critical software may have, and decrease the cost of software development while still producing more reliable applications.

Program analysis faces various challenges depending on the kind of software we study, the programming language in which it is written and the properties in which we are interested. These questions decide which methods can be used and how.

[*]CVE-2014-0160
[†]`https://heartbleed.com/`
[‡]National Institute of Standards and Technology

1.1 Context

To produce safer code, we can use methods that constrain the developers to write programs in a way such that they are more likely to be correct, or we can work on a source code to find errors. The first category is well represented by programming guidelines (like MISRA[§] by impelling the programmer to avoid implicit behaviors), or by high-level languages that provide more guarantees to the developer (for instance, languages that detect errors and raise exceptions, rather than simply going into undefined behaviors, or languages with richer libraries or features, assuming libraries and compiler implementations are reliable).

Once the code is written, we can work on it. Static analysis is the art of working on the source code of a program without running it, unlike tests that run the program on known inputs to compare the result with expected outputs. There are various kinds of static analysis allowing various kinds of results. The most basic analyses are lexical: they care about the formatting of the source code, without further considerations. A classical example in this category is indentation check: even in languages where it is not significant (like C), a bad indentation can be misleading for the developer, or may be the sign that the developer misunderstood what the program is doing. We may also check that variable names follow given guidelines, or that the spelling is correct.

We can be interested in more complex properties, especially syntactic ones: they look at the structure of the program, but they ignore the formatting, and have no consideration about the behavior of the program. This kind of analyses include the detection of code duplication or the limitation of the number of execution paths in functions. Lexical and syntactic analyses are very widespread, many compilers and IDEs perform such analyses as they help developers to write better source code, and to avoid risky idioms.

Some languages require the compiler to perform static analyses, like type checking or type inference. These analyses are more advanced as they must care about the behavior of the program: they are not syntactic. These properties are useful for safety concerns, and to make the compiler able to choose appropriate data representation. Moreover, compilers often run other complex analyses for optimization reasons, like pointer aliasing analyses or dead code elimination. We can remark that trying to compile a program is already a kind of static analysis, as it may succeed only if it is syntactically correct, and if types match language requirements (if any). On the other hand, this analysis provides very few guarantees: most programming languages allow writing programs that always crash, for instance, programs whose first instruction is an unprotected division by 0. A notable exception is Coq [Coq20], which requires the developer to provide a proof that the program is correct, and rejects the program if the compiler fails to check the proof.

Nowadays, we daily use such analyses in software development without even knowing it; thanks to them, developers can quickly notice likely errors. But for some applications, this is not enough. For critical software, a failure can be disastrous, and common methods are not reliable enough. For instance, the software involved in the crash of Ariane 4

[§] `https://www.misra.org.uk/`

was written in a high-level language (Ada[¶]), following very strict guidelines, and was extensively tested[**]; still Ariane 5 crashed. For this kind of software, we need stronger guarantees: we do not only need to detect probable bugs, but we want to prove that bugs are impossible. The definition of "bug" depends on the context: we may want to guarantee the absence of crash, the absence of private information leak or that the result is correct according to some specification. These properties are about the behavior of the program, they are semantic properties.

It is well known that interesting semantics questions cannot be solved using an algorithm, for a good definition of interesting: this is Rice's theorem. However, non-automatic methods cannot be used on large codebases, or at the cost of a very long time of expert work. Even if automatic methods cannot be infallible, we can strive to make them successful on typical source codes. We may also develop semi-automatic methods that do most of the job automatically but can be manually helped (preferably, by a non-expert) when it fails.

1.2 Motivation

The main motivation of our work is to prove security properties on an embedded operating system (OS) from an industrial partner. This OS hosts several programs, some are provided by the manufacturer and trusted, other are installed by the customer and untrusted. In addition to the absence of run-time errors (RTEs) in the OS, which is simply a safety property, we want to prove more complex properties like the absence of information flow from a private source to a public memory, so that an untrusted program cannot access data of trusted applications. Likewise, we want to prove that public variables cannot interfere with trusted software, this way the behavior of trusted programs does not depend on the behavior of untrusted ones.

In general, these properties are difficult to prove, and even difficult to express: they cannot be formalized as sets of traces, but only as sets of sets of traces, what some people call hyperproperties[††]. The absence of interference does not necessarily require that the untrusted program never writes in a trusted memory, but only that whatever it writes, the trusted program runs in the same way. We cannot directly use this property as we do not know the program running in our host platform. A first step is to say that the trusted memory can be modified by the untrusted program, but will eventually be restored its initial value before the trusted program reads it. Instead of comparing sets of traces, we only compare two traces: one real trace where the interaction happens, and one forged when the interaction is suppressed. If both traces match before the trusted program reads the memory, then, from its point of view, it is just like if the interaction never happened. This method can already be used in some parts of the OS where we need

[¶]Ada is statically typed, has an unambiguous syntax and handle errors by exception (rather than undefined behavior, like C).

[**]Indeed, in addition of tests during development, this software was used on the 113 successful flights of Ariane 4.

[††]At the risk of suffering the wrath of Patrick Cousot.

that a memory keeps its value at some times, but it is still too flexible to be used about user-level programs, as we would still need to know their source codes. The ultimate simplification is to simply forbid programs to read or write memory belonging to other processes. Though it seems very natural, it is in fact a strong over-approximation of the actually desired property. This property is interesting as it is a safety property, and that it is indeed enforced by memory management features of modern processors.

We still have to prove that the processor makes a correct usage of these protection features. They are way beyond the scope of C: these features cannot be controlled using pure C, as they are very dependent on the architecture, and properties about these features cannot be expressed in the memory model of C. We need to look at a lower level: we need to analyze assembly source code. In our case, we are using a processor with x86 architecture.

Of course, most OSes, and ours is not different, are not entirely written in assembly. The largest part is written in C, and only hardware-dependent pieces are written in assembly. This hides two problems. Firstly, we need to analyze mixed C and assembly source code, though they are very different languages: they have very different memory model, and while C is well-structured, assembly control flow is jump-based. The other problem is the kind of operations required by low-level software, which are unusual, or even illegal in standard C.

Since we want to prove properties on software, abstract interpretation is a convenient framework. It allows building sound analyses which are able to find all possible errors (at the risk of finding a bit more). The choice of the theoretical background was mainly guided by the application: a large part of this work is implemented in a development version of Astrée static analyzer. Astrée is a static analyzer relying on abstract interpretation developed since 2001 that managed to prove the absence of RTEs in large industrial embedded software [Bla+03; Cou+01; Cou+20; Cou+06a].

Because of the application, we not only need to find a solution to the problems we mentioned: this solution has to be efficient enough to be usable on real-world programs containing hundreds of thousand to millions lines of code.

1.3 Overview

We first consider the question of analyzing mixed C and assembly code. The first part is dedicated to a definition of the semantics of mixed C and assembly. Of course, we do not use C (which is very complex) or the whole assembly language, but a simpler model instead of C, and a fragment of x86 instructions. First, we introduced this C-like language from syntax to semantics. This language is not a subset of C as we need extra hypotheses that are undefined, unspecified or implementation-defined behaviors in C. We try to highlight these differences. Then, we introduce the memory model for assembly and two semantics. The first semantics is very concrete, it is very close to how computers work; the second semantics is slightly more symbolic to make interaction with C easier, but it forbids some unwanted behaviors. We explain how to write mixed C and assembly code, and give examples of complicated cases that justify why we cannot restrict the semantics too much. We explain the semantics of mixed code by looking at each kind of statement, but we do not give a full formal definition as it is very heavy and do not help to understand. The last chapter is about a library that we developed for Astrée, which is in charge of assembly-related problems, mainly deciding the length of assembly instructions to resolve dynamic jumps.

The second part gives some context needed for the following. We give a quick overview of abstract interpretation, and especially of products of domains. This does not intend to be exhaustive as a lecture. We also introduce Astrée, and explain some details about the implementation. These details are useful to understand the implementation choices made in the following.

The third part is about the abstraction of mixed C and assembly code. We divide the work in instruction categories. For each category, we present our abstractions, and the changes they imply in Astrée.

We mentioned that low-level source codes perform strange operations. This problem is discussed in the fourth part. Such cases can be solved by adding ghost variables: they are quantities that does not directly appear in the program but help to represent states. We found analogous notions in other domains; for instance, in physics, the law of conservation of energy can be used to solve many problems, but requires the total energy of a system to be explicit, which is an artificial value that has no existence in real life, just like the limit of the system is arbitrary and human-made. Likewise, given a problem, any solution that uses change of variables can be rephrased with ghost variables, e.g. variable change in an integral computation or coordinate rewriting in a geometric problem. In static analysis, ghost variables are very useful as well, but they are difficult to use in a decentralized way with multiple abstract domains. In the fourth part, we build a product of abstract domains that makes domains able to share ghost variables, and manage them in a fully distributed way, while allowing domains to cooperate to improve their abstract values. This allows the composite domain to express more complex properties.

The fifth part gives some perspectives and insights. We propose many extensions and improvement of the work presented in previous parts. In particular, we propose some advanced properties to prove that require the support of assembly. We also mention many optimizations of the product of domains discussed in the fourth part, and some extensions to allow even more complex domains and to make the product more general, so that it can be used in many contexts.

Chapter 2

The x86 Architecture and System Programming

THIS CHAPTER explains relevant parts (for the following chapters) of the x86 architecture and how these low-level features are used to build an OS. These explanations can be arranged in two categories. The first kind is about the actual features of the processor. Most of this information might be found in the INTEL Architectures Software Developer Manual [Int20]. The second kind is the set of methods and practices employed to build an OS using these features. In this matter, there is no absolute truth but widespread good techniques and traditions. The INTEL Architectures Software Developer Manuals gives some pieces of advice but the main part of these techniques rather comes from tutorials and documentations about existing OSes, since most of them work in similar ways. A most useful resource in this matter is OSDev [OSD20].

2.1 An Overview of the x86 Architecture

2.1.1 A Piece of History

The x86 architecture was born in 1978 with the release of the 8086 microprocessor, the first INTEL's 16-bit processor. As x86 maintained backward compatibility, novelties introduced in the 8086 processor are still part of the modern x86 processors. Yet, the 8086 lacks fundamental features that are essential today, for instance to handle multitasking. Consequently, since the 8086, the x86 architecture was enriched to deal with new needs.

A first important landmark is the introduction of a new memory management mode by the 80286 processor in 1982. In the 8086 framework, the physical memory was addressed directly by the program (real-address mode), preventing to run two programs simultaneously as they could interfere. The 80286 added the protected mode: in this framework, each program can only address its own virtual view (called segment) of the memory, and the processor takes care to map views on separate areas of physical memory as instructed by the system. Its successors (and contemporary) until the release of the Pentium were named on the pattern 80x86, giving by apheresis the name x86.

The 80386, released in 1985, was the first 32-bit x86 processor. It is also named i386, hence the informal name of the 32-bit generation of x86 architecture, otherwise called IA-32. The 80386 brings a lot of changes. To offer better support for 8086 programs, a new virtual mode was created to run real-address mode applications in an isolated environment in protected mode. These programs were not designed to work with segmentation or paging, and at this time, it was an important issue to be able to still run these old programs on new processors. As such old software was gradually deprecated and replaced by new programs designed to work with modern memory protection features, virtual mode became less and less useful. The change to 32-bit generation did not only extend registers, but also the external bus, allowing to address a much larger memory (from 16 Mio to 4 Gio, see section A.1 "Architecture" (page 369) for units). In parallel, a new memory management feature was added: paging. It provides a much more flexible control of the memory by allowing to map segments to non-contiguous small blocks of memory. With time, paging became more and more powerful so that it entirely subsumed segmentation, which became unused and eventually removed from the 64-bit generation (when operating in 64-bit mode).

Due to backward compatibility, even the very last x86 processor starts just as the 8086. The system is in charge to gradually change execution mode to the desired one. During the following description of the x86 architecture, some points may seem odd, even wrong. One should keep in mind that this is the result of many layers to add new features while keeping the old ones.

2.1.2 Characteristics of the 32-bit x86 Architecture

An architecture (or instruction set architecture) is an abstraction of a processor: it describes the available instructions, their semantics and their encoding.

The processor we are interested in is the Pentium MMX. Since it is a 32-bit, we will not consider 64-bit architecture. However, we cannot escape the influence of 16-bit processors, because of backward compatibility.

The x86 architecture is a CISC (complex instruction set computer) architecture, i.e. there is a lot of instructions: there are multiple versions of simple operations (depending on the type of the operand: literal value, register or memory operand) and very specialized instructions. In particular, most instructions can read or assign registers as well as memory. Once encoded into machine language (binary encoding), each instruction will take between 1 and 15 octets (tight bounds).

In x86, each byte is 8-bit long, that is a byte is an octet. This architecture uses little-endian byte order and allows any unaligned access (though they might be slower). All integer operations assume two's complement representation.

In the following, we will only focus on the Pentium MMX. In fact, we will not look at any feature that makes the Pentium MMX different from the P5, or most 32-bit x86 processors. However, really ancient processors may not have all the thereafter explained features, and more recent x86 processors may have more (while keeping compatibility).

2.1.3 Conventions

Most notations follow the INTEL's manual [Int20]. This way, it is easy to find complementary information. Moreover, due to INTEL's influence, these notations are quite common in a lot of resources. Let us explain them.

The first point is notation of integers in non-decimal form. Hexadecimal numbers have the H suffix and use numbers 0 to 9 and A to F. Binary numbers have the B suffix and use numbers 0 and 1. For instance 42 = 2AH = 101010B. In most programming languages 2AH is written `0x2a`.

When drawing data structures, lower addresses are at the bottom and the byte-offset is written on the right. Bits are numbered from right to left. For instance, the array of the 16 hexadecimal 4-bit digits in increasing order can be equivalently written as on Figure 2.1 or Figure 2.2. The number of bytes per line depends on the context. Usually, there are 1, 2 or 4 bytes per line.

15 12	11 8	7 4	3 0	
F	E	D	C	6
B	A	9	8	4
7	6	5	4	2
3	2	1	0	0

Figure 2.1: Hexadecimal numbers by 16-bit packs

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0	
F	E	D	C	B	A	9	8	4
7	6	5	4	3	2	1	0	0

Figure 2.2: Hexadecimal numbers by 32-bit packs

There is also a notation to extract some bits: a[c:b]. This means “bits b to c of a”. For instance if a is 12ABCDEFH, a[27:4] = 2ABCDEH (see section A.1 “Architecture” (page 369) for a more detailed explanation). Bounds are reversed to match the bit order of figures. Outside the context of figures, natural order is preferred (first index before last index).

Reserved parts are in gray. A constant written in such a cell means that this field shall always contain this constant. Otherwise, the content may be arbitrary.

When speaking about bits, saying it is cleared means that its value is 0, and we say it is set if its value is 1.

2.2 Execution Environment

2.2.1 Registers

Among other resources given to program running on the processors, there are the registers. They are small memories that exist only in the processor: they are not mapped in memory, and consequently they have no address, thus no pointer can point to them. Let us remark that “a memory” designates anything that can store data, but “the memory” designates more precisely the memory external to the processor accessed by dereferencing, typically the random-access memory (RAM).

There are several categories of registers; we are going to present them in this subsection.

2.2.1.1 General-Purpose Registers

Some registers are available for the user without restriction. They are called general-purpose registers (GPRs), or sometimes GP registers. There are 8 GPRs, all of them are 32-bit wide. They are named EAX, ECX, EDX, EBX, EBP, ESP, ESI, and EDI. As one may have noticed, EBX come after EDX, especially when it comes to encode the instructions. This usually has no importance, but sometimes, it explains surprising facts.

In addition to these registers, other names are defined to designate parts of these registers. For EAX, the 16 lower bits are called AX, the 8 lower bits form AL and the bits 8 to 15 are AH. This is illustrated on Figure 2.3. Similar names are given to analogous parts of ECX, EDX and EBX.

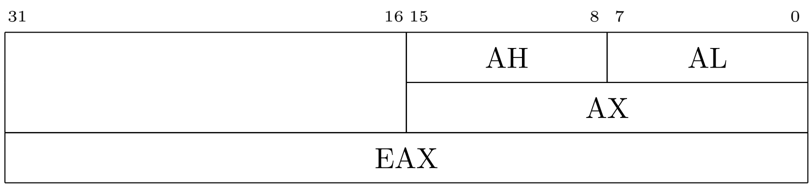


Figure 2.3: Alternate EAX register names

The 16 lower bits of EBP, ESP, ESI and EDI are respectively called BP, SP, SI, and DI. The “E” prefix stands for “extended”, since 32-bit registers are extension of the former 16-bit registers.

These registers can be freely used to store any kind of data (integer or pointer, since there is no typing at this level), without restriction, especially whatever the current privilege level, and without any kind of check. While it is possible to store anything, it is not always wise. Indeed, some instructions favor some registers. For instance, addition is more efficient when using AL, AX or EAX: it is both faster and the encoding is smaller. Moreover, some instructions exist only for some registers. For example, string operations can only be performed using ESI as a pointer to the source operand and EDI as a pointer to the destination operand, hence their names (see below). Likewise, when performing a stack operation, ESP will be interpreted as a pointer to the top of the stack. Moreover, while all GPRs may be dereferenced to be used as memory operand, they are not understood in the same way: during a dereference, pointers must be translated to physical memory address, but this translation is done slightly differently depending on the register. This will be explained with more detailed later (see subsection 2.3.2 “Segmentation” (page 18)).

Even if there is no intrinsic constraints on the respective usage of GPRs, their names carry some hints:

– ESP: Stack Pointer
– EBP: Base Pointer (in stack)
– ESI: Source Index (for string operations)
– EDI: Destination Index (for string operations)
– EAX: Accumulator (for arithmetic operations)
– ECX: Counter (for string operations)
– EDX: Data pointer (for I/O operations)
– EBX: Base pointer (for 16-bit-style addressing)

While the 4 former are credible, the others look more dubious, or even apocryphal. In practice, ESP and EBP are indeed bound to stack management.

It is important to remember that values are not typed in x86: they are just binary sequences. It is true everywhere, but especially important to emphasize about GPRs since they have no special meaning. The content of a register, or a memory operand, can be interpreted as a pointer, a signed integer or an unsigned integer, depending on the instruction. That’s why there are both `MUL` and `IMUL`: the former performs an unsigned multiplication, while the latter performs a signed multiplication. Thanks to two’s complement representation, some computations (such as addition) are the same in both cases.

2.2.1.2 EFLAGS Register

The EFLAGS Register is another very important 32-bit register even for non-system programs. It keeps 3 kinds of flags: status, control and system flags. Status flags give some information on the last executed instruction. They are automatically updated and consequently, from the software point of view, they are mainly intended to be read. A control flag is rather meant to be modified: its value modifies the behavior of some instructions. System flags are dedicated to system management and should not be modified by an application program. Moreover, as a protection feature, one needs the highest privileges to change these flags. The layout of the EFLAGS register is shown on Figure 2.4. Some bits (in gray on the figure) cannot be changed.

The EFLAGS register cannot appear directly as an operand. To read or write it, one can either use dedicated instructions that set or clear specific flags, or copy the whole register from or to the stack. In most cases, flags are written and read automatically without manipulating the EFLAGS register.

There are 6 status flags: Overflow Flags (OF), Sign Flag (SF), Zero Flag (ZF), Auxiliary Carry Flag (AF), Parity Flag (PF), and Carry Flag (CF). They are set by

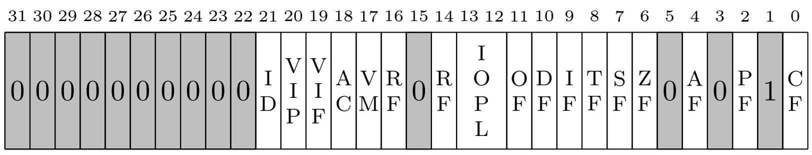


Figure 2.4: EFLAGS register

arithmetic instructions (such as addition, multiplication...) depending on the result. For instance, the Zero Flag is set if the result is null, and cleared otherwise. Testing status flags is the main way to execute code conditionally, excluding tricks like dynamic jumps. Conditional jump instructions might simply test the value of a flag or a combination of flags. Moreover, most of these instructions have aliases. For instance `JZ` (jump if zero) tests if ZF = 1, just as `JE` (jump if equal) which is a mere alias for convenience. Indeed, comparison instruction (`CMP`) sets status flags just as `SUB` and does not modify the operands: ZF is set if the difference is zero, i.e. if the operands are equal. Even if it is equivalent, it is more natural to test whether the result of `SUB` is zero and whether operands of `CMP` are equal; that why `JZ` and `JE` are aliases. Complex conditions are used to provide convenient comparisons. For instance `JA` (jump if above, the unsigned strict comparison) tests if ZF = 0 and CF = 0 without the need to use two instructions. Since these registers are automatically written by arithmetic operations and read by conditional operations (including, but not limited to, jumps), it is very unusual to read or write status flags manually and x86 does not make it easy.

There is only one control flag: Direction Flag (DF). It is used to change the behavior of string operations (`MOVS`, `CMPS`, `SCAS`, `LODS`, and `STOS`). String operations process contiguous block of data, by applying an instruction to each value in this block. DF flag sets the direction in which the string is processed: from high to low addresses, or the opposite. This flag can be set or cleared using dedicated instructions. We will not take care of this flag anymore since we are not interested in string operations.

The other flags are system flags. One needs the highest privilege to modify them, and they control some system features or indicate capabilities of the processor. Most are not relevant for the following, the others will be detailed in due time.

2.2.1.3 Segment Registers

Segment registers allow some control on logical-to-physical address translation. There are 6 of them (CS, DS, SS, ES, FS, and GS) and are 16-bit wide. Their role is explained in subsection 2.3.2 “Segmentation” (page 18).

2.2.1.4 Instruction Pointer

The instruction pointer (EIP), as suggested by its name, points to the next instruction to run. It is impossible for software to access it directly: it is automatically controlled by control-flow instructions (jumps, calls...). Though, one can take advantage of these instructions to get the current value or to assign an arbitrary value into EIP, it goes without saying that this is a very poor idea in most application programs. But sometimes it can be necessary for system to use such trick.

2.2.1.5 Control Registers and Other System Registers

While all aforementioned registers may be manipulated (at least partially) by an application program, there are others families of registers that are only intended to be used by the system. The highest privilege is required to access them,and they are used to control the mode of operation, memory management features and other sensitive parameters. Relevant registers, with respect to our scope, will be detailed at an opportune moment (especially subsection 2.3.2 “Segmentation” (page 18) and subsection 2.3.3 “Paging” (page 25)). Important such registers are the control registers, named from CR0 to CR4.

2.2.2 Stack

Most languages need a stack to store information about active subroutines. The stack is not a distinct hardware feature, instead it takes place in the memory as a set of consecutive cells. Nevertheless, x86 provides several operations to easily operate on a stack. In x86, stacks grow downward, thus a `PUSH` will decrease the address of the top of the stack. For all the stack operations, it is assumed that ESP points to the top of the stack, that is the element with the lowest address of the stack. In addition to store or retrieve data from the stack, these instructions update ESP accordingly. Some relatively high-level instructions handle entire call frames, managing both EBP and ESP to surround them, and keeping appropriate copies on the stack to be able to restore the previous state. Thus, even if the stack is a purely logical structure, x86 processors include several instructions to handle it pleasantly.

2.2.3 Privileges

Privileges have already been mentioned briefly. This is a mechanism to restrict some sensitive operations only to some trusted programs (especially, the system) and to protect these programs against less privileged tasks.

There are 4 privilege levels, called rings, numbered from 0 to 3. Ring 0 is the most privileged level, while ring 3 is the least[*]. A numerically higher is in fact a lower privilege. Most tasks (in particular, application programs) run in ring 3, only the OS kernel should run in ring 0. Rings 1 and 2 used to be the privileges of OS services, such as drivers. Nowadays, they are much less used for reasons related to hardware and software: on the processor side, the 64-bit generation of x86 mostly dropped segmentation in favor of paging, which has only a protection model with 2 levels; from the OS side, limiting

[*]Hence the name “ring”: privilege levels are like a succession of included circles, where ring 3 is the outermost and ring 0 the innermost. This diagram can be found in many places, especially [Int20]. This is the traditional representation, though it is counterintuitive from a set-theoretic point of view (since the set of allowed operation for ring 3 is included in the set of allowed operation for ring 0). We may rather interpret this representation like battlements in a fortified city: ring 0 is the innermost and thus the most secured place.

hardware-dependent code is an important issue for portability and maintainability, yet some architectures only provide two privilege levels, thus on x86, two rings are left unused.

Sometimes, it is useful to temporarily change ring to allow an application program to perform privileged operation, such as allocating memory. Such calls are known as system calls. There are several ways to increase privilege safely: we do not want the unprivileged task to run arbitrary code with high privilege. These transfer mechanisms are detailed in section 2.4 “Calls, Interrupts and Exceptions” (page 28) and section 2.6 “System Calls” (page 36). Whatever the way by which the transfer happens, it is crucial to prevent the system to access sensitive data, or to run protected code, on behalf of an application program.

To achieve delegation to higher privilege code while ensuring protection, x86 uses three kinds of privileges:

- The current privilege level (CPL) is the privilege of the currently executing task.
- The descriptor privilege level (DPL) applies for data segments and gates (safe proxies for privileged functions see subsubsection 2.4.2.3 “Inter-Privilege Calls” (page 31)). They describe the required privilege to be allowed to access the data (in the case of a data segment) or to run the code (for gates).
- The requested privilege level (RPL) is an override to access a data segment or a gate with a numerically higher privilege level (less privileged). This is useful to execute code or access data on behalf of a less privileged task.

2.2.4 Modes of Operation

Processor of the x86 family can run in three modes. Each mode determines a set of available features and instructions.

The legacy mode is called real-address mode. This is the only mode of the 8086 thus, for compatibility reasons, this is the mode of the processor at startup. Most protection features are unavailable, so it is strongly advised that non-legacy software does not use this mode.

The normal mode of operation is the protected mode. It can be accessed from real-address mode and provides most features and instructions, in particular memory protection features (hence the name) needed for multitasking. Moreover, since it is designed to be the standard mode of operation, this is also the most optimized mode. In most cases, the first task of the system is to transit from real-address mode to protected mode. Tasks running in protected mode can be run with a special parameter so that the processor emulates the real-address mode in a protected framework, allowing running software designed for the 8086. While not technically being a distinct mode, this feature is called the virtual-8086 mode.

The third mode is the system management mode. This mode is intended to run code to manage hardware-specific features such as power management. It provides an easy way to execute this code transparently (OS or application programs are automatically suspended and resumed) in a isolated environment. This mode is out of our scope.

2.2.5 Memory Protection Overview

Real-address mode allows programs to access physical memory directly. The only way to run multiple tasks in such a mode would be to use disjoint addresses. This is not feasible in practice: when choosing the addresses to store global variables and functions (during compilation, typically), one would have to know which are available. In addition to the annoyance due to accidental interference of genuine programs, there is no way to protect data from an adversarial program.

To tackle this issue, protected mode provides features to isolate memory space of programs. There are two consecutive mechanisms of address translation: segmentation and paging. Enabling paging requires segmentation to be currently active. In normal operation, both segmentation and paging are enabled. The state where segmentation is enabled but not paging only appear shortly during boot. Since there are two translation steps, there are three address spaces: logical, linear and physical addresses.

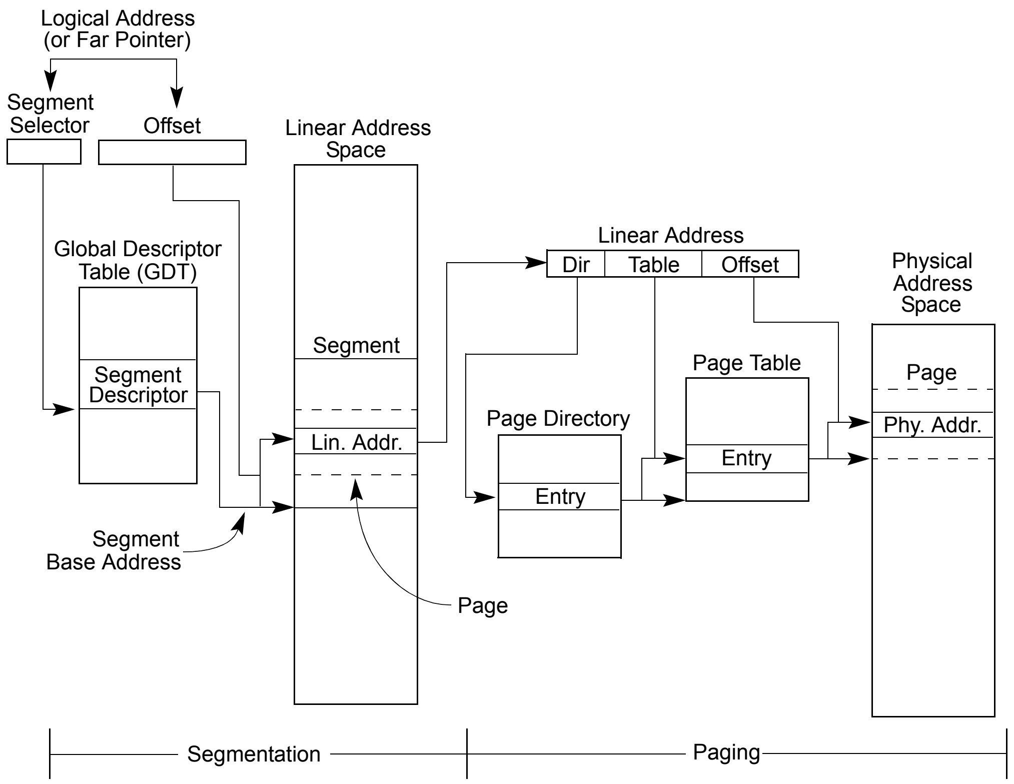


Figure 2.5: Segmentation and paging, from [Int20]

Segmentation translates logical addresses (also known as far pointers) to linear addresses. A logical address is the address from the program point of view. Such an address has no meaning by itself, it should be interpreted in a segment. A segment is a contiguous memory zone characterized by a base address (the beginning of the segment), a size, and some flags. Segments are usually defined globally, but task-specific segments are also available. A logical address comes with a segment selector (stored in a segment register), which is an index that specifies in which segment the logical address must be interpreted. Then, the logical address is added to the base address of the segment yielding the linear address. Checks are performed to ensure that the linear address does not exceed the segment limit. This process is shown on the left part of Figure 2.5.

Depending on whether paging is enabled, the linear address may undergo another translation. If paging is disabled linear address is directly interpreted as a physical address. In this case, to ensure memory isolation, tasks must use disjoint segments. This lacks flexibility since it implies that each task requires a contiguous block of memory. So, memory allocation may require entire segments to be shifted if there is no space left after the end of the segment. In addition, holes cannot be used easily and segmentation does not allow easily storing allocated but unused memory on an external device (typically a hard drive or a SSD) to make room for more frequently used data (this technique is known as swapping).

To overcome these issues, we usually use paging which is a second translation step that provides a useful mapping from linear addresses to physical addresses. Linear address space is cut into small pieces called pages which are mapped anywhere in memory. So, the processor determines to which page the linear address belongs and the offset inside this page. Then it finds the physical base address of this page in a dedicated task-dependent structure and adds the offset to get the physical address. In this framework the granularity of memory is coarser: we cannot allocate less than a page. Yet, pages are small (usually 4 Kio), so this is not an issue. This translation is illustrated on the right part of Figure 2.5.

Paging solves all aforementioned issues. To allocate memory, the OS just needs to find available space to put the pages. They do not even need to be adjacent. Swapping can be done at page level: when the processor finds out that the requested page is not in memory, it asks the OS to move it into memory before proceeding. To achieve memory isolation, pages must not overlap, but segments are not important anymore: the same linear address space is translated differently in the context of different tasks. Consequently, in most OSes, segments are shared across all tasks and span over most of the linear address space, so that the segmentation is almost trivial. This led to the removal of most capabilities of segmentation in the long mode (64-bit mode) of x86_64.

Both segmentation and paging will be explained with more detailed in section 2.3 “Memory Management” (page 17).

2.2.6 Assembly Language

To be understood by the processor, instructions are encoded into binary sequences, this is machine code. Yet, this format is clearly not human-friendly. Instead, we use an assembly language (or assembler language): it is an understandable language very close to the machine code.

There are several syntaxes for x86 assembly, the most popular ones are INTEL’s and AT&T’s. We will use the INTEL’s syntax in the following since it has slightly more in common with other languages (direction of assignments) and has more understandable syntax for complex addresses. Moreover, it is closer to the INTEL's manual [Int20].

Assembly language use mnemonics followed by operands (separated by commas) to specify the instruction. A mnemonic is a keyword encoding the kind of instruction. For instance `PUSH EAX` is the instruction that pushes the content of EAX on the stack. The mnemonic is `PUSH` and EAX is the argument. An example with a binary mnemonic: `ADD EAX, 5` adds 5 in the register EAX. More generally, when applicable, the source operand is on the right while the destination operand is on the left, just like in most languages. We can also remark that usually, mnemonics and other keywords (like register names) are not case-sensitive. Indirection is denoted with brackets. For instance, `PUSH [EAX]` pushes on the stack the content of the memory whose address is the content of EAX.

Assembly languages usually include labels. They are identifiers followed by a colon (e.g. `my_label:`) that marks a program point, making it easy to reference, for instance, as the target of a jump or call instruction. Labels do not exist in machine language, and they are replaced by the offset between the usage and the definition.

Assembly language is translated into machine language by an assembler program (or simply assembler). Fancy assemblers can provide more advanced features like macros or high-level style branching. This separates even more assembly language from machine code, so we will not use such features.

2.3 Memory Management

2.3.1 Address and Operand Sizes

The processor can use 16-bit or 32-bit addresses and operands depending on the current mode and configuration.

When using 16-bit addresses, the maximum legal value of a linear address (known as limit) or segment offset is $2^{16} - 1$ (i.e. 64 Kio) and with 32-bit addresses, the limit is $2^{32} - 1$ (i.e. 4 Gio). It is equally straightforward for operand size: 32-bit operand size means that the size of operand is 32 bits by default.

In real-address mode, the default address and operand size is 16 bits. In protected mode, it depends on a flag in the segment configuration. However, both address size and operand size can be overridden temporarily by adding appropriate prefixes to the instruction, so that an instruction in protected mode can access the whole memory (32-bit address size), yet use AX register (16-bit operand size) by using the operand-size override prefix while in 32-bit mode.

The recommended setting is to use 32-bit sizes. This offers more capabilities (wider ranges for integers and larger addressable memory) while achieving better performances. Only serious constraints should impose 16-bit addresses and operands. But, we cannot totally escape 16-bit sizes because the processor starts in real-address mode, thus we are compelled to use 16-bit code at least until we reach protected mode.

2.3.2 Segmentation

While sharing the name, some terminology and segment registers, real-address mode segmentation and protected mode segmentation are very different. The similarity is that in both kinds, segments are contiguous zones of memory (like a segment in the mathematical meaning) and the logical address is used as an offset inside this segment. But the capabilities and the way to specify a segment have very few in common.

There are 6 16-bit segment registers: CS, SS, DS, ES, FS, and GS. While the way they specify the segment varies between real-address mode and protected mode, they are used in the same circumstances:

- CS (Code Segment): where instructions are fetched;
- SS (Stack Segment): when dereferencing EBP or ESP, or performing a stack operation like a push or pop;
- ES (Extra Segment): destination of string operations;
- DS (Data Segment): for everything else.

For memory operands, the automatic (register-based) selection of segment may be overridden, e.g. `MOV EBX, [EAX]` dereferences EAX in the data segment (DS) and stores the result in EBX, but `MOV EBX, SS:[EAX]` will look in the stack segment. So `MOV EBX, [EAX]` is equivalent to `MOV EBX, DS:[EAX]`. FS and GS segments are only used through such explicit overrides, as they are never automatically selected.

2.3.2.1 Real-Address Mode Segmentation

In 16-bit processors, registers are 16-bit long. This provides an addressable space of 2^{16} octets (64 Kio). It rapidly became too small regarding the development of software.

To extend the addressable space, the INTEL 8086 has a 20-bit memory bus, allowing it to use 1 Mio. To achieve 20-bit addresses with only 16-bit registers, the physical address is `10H` $\times S + O$ (see Figure 2.6) where S is the content of the segment register (known as segment selector), O is the offset (logical address) and `H` is a suffix to denote that the value is denoted in hexadecimal (so that $10\text{H} = 16$).

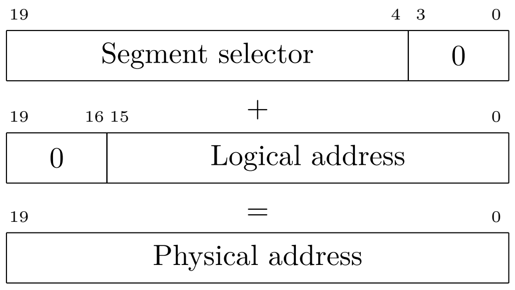


Figure 2.6: Real-address mode address translation

Segments are 64 Kio long but spaced out by only 16 octets, thus they overlap. Consequently, the same linear address can be obtained with up to 2^{12} distinct pairs of segment selector/offset.

By using disjoint segments for code, data, and stack, we can avoid catastrophic mistakes like writing in the code, or executing data. But this is only a protection against some genuine mistakes, but it is useless against adversarial attackers, since the program can change current segments (as there is no privilege mechanism in real-address mode), and indeed execute data or write in the code. Real-mode is poorly suited to multitasking (executing simultaneously two programs requires them to statically use disjoint memory zones), so usually, there is only one program at a time in this mode, thus there is nothing to attack except the attacker itself. But in a multitasking mode, we need better memory protection.

2.3.2.2 Protected Mode Segmentation

This segmentation is much more flexible: segments may have arbitrary base address and length. Available segments are described by segment descriptors, which can be stored in a system-wide structure, the global descriptor table (GDT), or in a task-specific structure, the local descriptor table (LDT). Segment registers point to descriptors, specifying indirectly the segment.

2.3.2.2.1 Segment Descriptor

A segment descriptor is a 64-bit structure that specifies segments parameters. The layout is showed on Figure 2.7. The base address is the lowest linear address of the segment. The limit field specifies indirectly the size of the segment depending on the G (granularity) flag:

- if G is cleared, the limit is directly the 20-bit size, allowing sizes from 1 o to 1 Mio, by steps of 1 o;
- if G is set, the size of the segment is 2^{12} times the value of the limit field, allowing sizes from 4 Kio to 4 Gio, by steps of 4 Kio.

The granularity flag allows favoring either the precision or the amplitude. In practice, we usually do not need to specify the segment size by less than 4 Kio since it is the smallest possible size for a page. Moreover, we usually need to specify large segments. So, the granularity flag is set in most use cases.

In a segment, the least logical address is 0 and the greatest is the limit (except for expand-down segments). The base is thus an offset added to the logical address to get the linear address. At this point, the least linear address is the base, and the greatest is base + limit.

The address translation of segmentation is rather trivial: it is a translation (in the mathematical meaning) with bound checks. But segment descriptors also include several flags to adapt segment settings and access rights:

- The DPL field specifies the privilege level of the segment, from 0 (most privileged) to 3 (the less privileged). For a code segment, it specifies the maximum privilege a task can have when running this code.
- The P (present) flag specifies whether the segment is available for use. If not, most fields are ignored and the segment descriptor cannot be used.

– The AVL (available) flag is left available to the system developer.
– The D/B flag specifies the default address and operand size.
– Bit 21 of the second double word is reserved and should be 0.
– The S (system) flag specifies whether the segment is a system segment, or a code or data segment. System segments are special structures that will be explained later.
– The Type field defines the type of segment (code or data) and access policy: execute-only or execute/read for code segments, or read-only or read/write for data segments. This field stores more information that will be explained later.

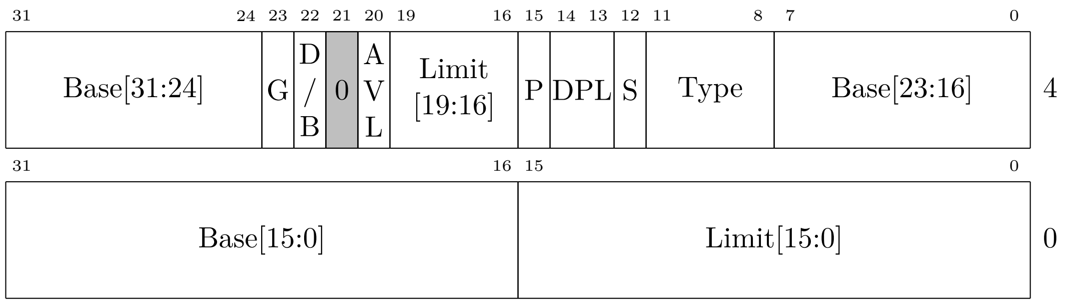


Figure 2.7: Layout of a segment descriptor

```c
uint64_t /* A type of 64 bits */
build_segment_descriptor(uint32_t* base, uint32_t limit,
    uint32_t type, uint32_t flags1, uint32_t flags2){
    uint64_t seg_desc = 0;
    seg_desc = limit & 0xffff;
    seg_desc |= (base & 0xffff) << 16;
    seg_desc |= ((base >> 16) & 0xff) << 32;
    seg_desc |= (type & 0xf) << (32 + 8);
    seg_desc |= (flags1 & 0xf) << (32 + 12);
    seg_desc |= ((limit >> 16) & 0xf) << (32 + 16);
    seg_desc |= (flags2 & 0xf) << (32 + 20);
    seg_desc |= (base >> 24) << (32 + 24);
    return seg_desc;
}
```

Listing 2.1: Building a segment descriptor

When S bit is set, the segment descriptor is not a system segment, oddly. Let us describe the type field in the case of a non system segment:
– Bit 11 of the second double word is 0: Data segment:
  – Bit 10: expand-down. In this case, the greatest logical address is FFFFH ($2^{16} - 1$) if the B flag is cleared, FFFFFFFFH ($2^{32} - 1$) otherwise, and the

    least logical address is limit + 1. The base is still the least linear address of the segment.
  - Bit 9: write. If this bit is cleared, the segment is read-only, otherwise, write is allowed.
- Bit 11 of the second double word is 1: Code segment:
  - Bit 10: conforming. Whether code in this segment can be accessed by a less privileged task, and run with current privilege.
  - Bit 9: read. If cleared, this segment is execute only. Otherwise, read is allowed.
- Bit 8: accessed. This bit is set when a selector to this segment is loaded, and should be cleared explicitly. It can be used for debugging or memory management, but is rarely useful, thanks to paging.

We can remark that a segment may not be executable and writable at the same time. This is an obvious precaution to avoid catastrophic failures. But we can also note that the same linear address might be in multiple segments, thus it is still possible to have a piece of memory (in the linear address space) that is both executable and writable by reaching it through different segments. This can be avoided using paging (see subsection 2.3.3 "Paging" (page 25)) to avoid execution of data or dynamically modification of the code, but it can also be useful, precisely when dynamic code is needed.

Expand-down segments may seem weird, but they used to serve a reasonable purpose: they were designed for stacks since in x86, they grow downward. If one uses an expand-down segment for a stack, decreasing the limit of the segment allows more space to be dynamically allocated on the stack, by decreasing the first illegal address. Modern (even for our scope) virtual memory management does not use dynamic segments: allocation is performed at page-level.

Conforming segment is a concept used to share pieces of system's code that does not require high privilege to run. For instance, a math or cryptographic library may be useful in the OS, and can be made available to application programs since it does not involve privileged operations.

It is important to mention that the field Base is a 32-bit address scattered all across the segment descriptor, as shown by Figure 2.7. The limit is also cut in two parts. Building a segment descriptor is usually done by using bitwise operations, like in Listing 2.1. Though building a segment descriptor requires such obscure manipulations, the inverse process is transparent: the correct information is rebuilt by the processor automatically.

2.3.2.2.2 Segmenting the Linear Address Space

Segments can alias each other in arbitrary ways. When they do, a write in a segment may be visible in another segment, but at a different logical address, which is not advised! There are some standard ways to use segments for different use cases, and having a predictable behavior.

The conceptually simplest safe way to avoid problematic behavior is to use non aliased segments, i.e. segments that do not overlap. This is known as the multi-segment model: each task use its own segments. This method ensures memory isolation at linear address space-level.

But this model is really heavy to implement and isolation at linear address space-level is not always desirable. For instance, the DS segment is used for most memory accesses while the ES segment is used for the output of string operations, but we usually want to be in the same space: if several string operations are needed, the output of an instruction may be the input of the following. So, it would be nice for DS and ES to be the same segment. Another problematic case is the problem of dereferencing C pointer. Let us consider the example of Listing 2.2. The variable `a` is a global variable, it is stored in the heap, which is classically in the DS segment. On the other hand `b` is local, thus allocated on the stack (SS segment). In the function `f`, one only see `x` as a pointer, without knowing in which segment it should be interpreted, and, as shown by the example, it can be one and the other. To make the compilation feasible without having to transmit more information, the simplest way is to have the DS and SS segments to be identical. One could also craft some imaginative example involving function pointers to argue that CS and DS should be identical. More generally, we often want all segments to be identical parts of linear space. We do not lose so much in protection since segmentation has a quite austere style of protection while paging is much more precise.

```c
int a = 5;
void f(int* x) {
    ++*x;
}
int main(){
    int b = 12;
    f(&a); // increments a
    f(&b); // increments b
    return 0;
}
```

Listing 2.2: Dereferencing a pointer to stack or heap

Based on this observation, a few variations exist. The most basic segmentation model is to have two segments (code and data) mapped to the entire linear address space. A variant is to have such a pair of segments for ring 0 and a pair of slightly shorter segments for user programs, but still starting at linear address 0. Segments are not identical, but where they overlap, each linear address corresponds to the same logical address in all segments. Some space at the end is unavailable using user-level segments, and thus can be used to store system data and code. The point of this style of segmentation will be detailed further in subsubsection 2.3.3.4 “Translation Lookaside Buffers” (page 28).

2.3.2.2.3 Global Descriptor Table

Until now, we have described the structure and the semantics of segment descriptors, but not how we can make them available to use. Indeed, they should occupy a privileged position to be used by the memory translation facilities of the processor. For that, segment descriptors are gathered in a structure called the global descriptor table (GDT).

The GDT is simply an array of segment descriptors. They are just put next to each other, with no gap. Thus, the byte-index of a segment descriptor is multiple of 8 (the size of a segment descriptor).

This structure is referenced by the GDTR register. It is a special register that contains the 32-bit address of the GDT in the linear address space, and a 16-bit limit, which is the offset (with respect to the base address) of the last valid byte of the GDT. Since all the fields are 8-byte wide, the limit should be the predecessor of a multiple of 8.

The GDTR is not directly accessible. It can only be written and read through special instructions, respectively `LGDT` (load GDT) and `SGDT` (store GDT). These instructions use a 48-bit memory operand: the 16 lower bits correspond to the limit part of the GDTR, and the 32 upper bits are the base address. `LGDT` simply copies these bits from memory to register and `SGDT` copies in the other direction. These operations require the highest privilege.

The GDT is global to the whole system, and is designed to be long-lived. There is another structure that provides more dynamic segmentation: the local descriptor table (LDT). The most notable difference with a GDT is that the processor automatically updates the current LDT when switching tasks. Once again, because of paging, this feature is rarely used and, anyway, is out of our scope. Using LDT would not make so much difference in the following.

2.3.2.2.4 Segment Selectors

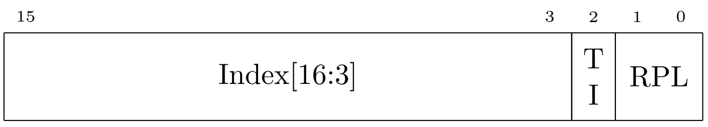


Figure 2.8: Structure of a segment selector

Making the segments globally visible is not enough: we should have a way to reference them. This is the role of segment selectors. They are 16-bit identifiers to specify a segment descriptor that lies in the GDT (or LDT). Since segment descriptors are 8-byte long, their byte-indices are multiple of 8, thus, the last 3 bits are always 0. So, in the segment selector, these bits fulfill other purposes. The structure of a segment selector is shown on Figure 2.8. The 3 lower bits of the index are omitted, since always null. The TI (table indicator) bit specifies whether the segment is in GDT (if cleared) or LDT (if set). The RPL field specifies the privilege level at which the segment should be viewed. This matter has already been briefly discussed in subsection 2.2.3 “Privileges” (page 13).

The RPL allows accessing a segment as another task would do it, so that, a privileged task running code on the behalf on a less privileged task will not access sensitive code or data. For an access to succeed, the RPL must not be numerically lower (more privileged) than the CPL, and the RPL must not be numerically higher (less privileged) than the DPL of the requested segment.

It is worth noticing that the CPL of a task is in fact the RPL of the currently executing code segment.

To use a segment selector, it should be loaded into a segment register usually using `MOV` or `POP`, for non CS, and with control flow operations (`CALL`, `RET`, `JMP`...), for CS. Assigning a segment register is not an ordinary write. Indeed, segment registers contain a visible and a hidden part. The visible part is the segment selector, while the hidden part caches information of the segment descriptor. When a segment register is loaded, the hidden part is loaded as well, so that modifying a segment descriptor without reloading the segment register has no effect. The hidden part is managed only by the processor, software has no access to this part.

Privilege checks also happen when the segment register is loaded.

2.3.2.2.5 System Descriptors

Since the GDT is a global structure, it is interesting to make it store other kind of data than segment descriptors. Such structures are called system descriptors and have roughly the same layout. The types of system descriptors are:

- LDT segment descriptor. This was already mentioned and will not be detailed further.
- Task-state segment (TSS) descriptor. This is a descriptor to a special structure, called TSS, that holds information about a task. This is used to save the current state of a task during context-switch.
- Task-gate descriptor, a protected selector to a TSS descriptor.
- Call-gate descriptor, a protected pointer to code that allow inter-privilege control transfer.
- Interrupt-gate and trap-gate descriptors. These descriptors contain pointers to procedures to run when the processor get an interrupt or an exception (see subsection 2.4.3 “Interrupts and Exceptions” (page 32)). These kinds of descriptors are not useful in GDT but are placed in interrupt descriptor table (IDT).

They are a very heterogeneous family of descriptors that fulfill various goals. The characteristic point they share is that they are useful only when they are globally accessible, from any context. Moreover, they all can be put in the GDT because they all have the same size (8 B) and the “type” field at the same place which allows the processor to identify their kinds.

2.3.3 Paging

2.3.3.1 Translation Mechanism

Paging takes place after segmentation and is optionally enabled, but requires segmentation to be enabled before. This is the modern way of managing memory. This mechanism maps the linear address space to physical memory by small chunks. Paging will be explained with much fewer details than segmentation.

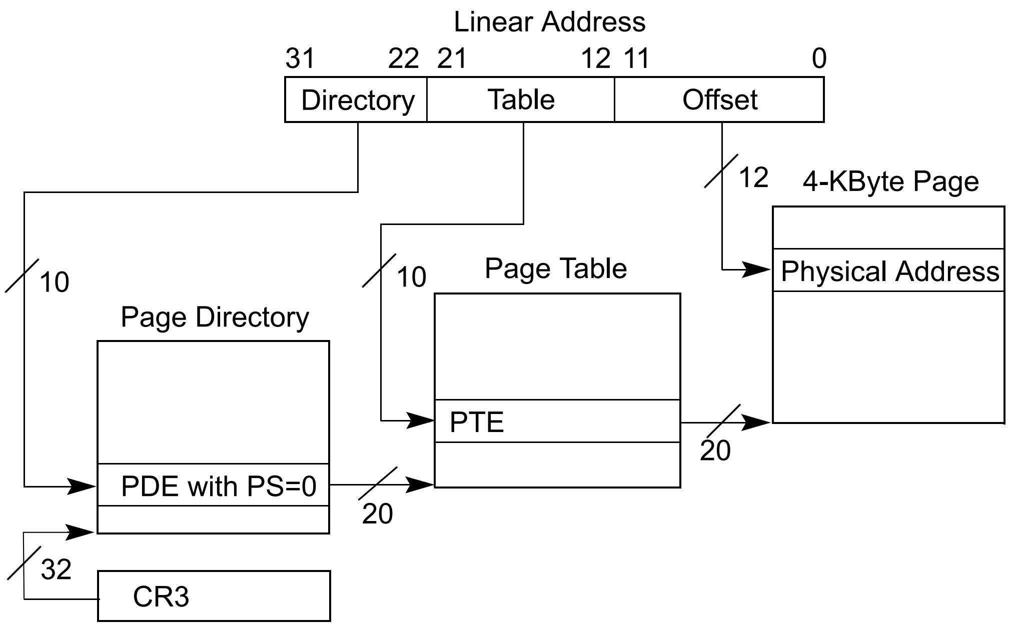


Figure 2.9: Paging, from [Int20]

Paging is controlled by flags in control registers, mainly by the PG flag (bit 31 of register CR0) that enables or disables paging. Two other flags are involved:

– PEA flag (physical address extension, bit 5 of CR4), allows using 36-bit physical addresses
– PSE flag (page size extension, bit 4 of CR4), allows using larger pages.

In the standard setting, none of these features are used since they have some downsides that we want to avoid unless these features are really needed.

Without these extensions, a page is 4 Kio (2^{12} octets) long. With a 4 Gio (2^{32} octets) linear space, we need 2^{20} pages to cover the entire linear address space.

The working of paging is mapped in Figure 2.9. Before explaining more precisely, the idea is to cut the linear address in 3 parts: the two firsts are indices in cascading registry of pages while the last part is the offset in the designated page.

Pages are 4 Kio long, and aligned on 4 Kio boundaries (i.e. the first address of each page is a multiple of 4 Kio), so that their base address is always a multiple of 2^{12}, i.e. the 12 lower bits are always 0. This 4 Kio alignment constraint is not limited to pages: it is a leitmotiv in everything involved into paging, so that, a lot of lower bits would not carry any information since they are always zero. In order not to waste such precious space, some flags are stored instead, in the same way as the 3 lower bits of segment selectors that store the RPL and the table indicator flag.

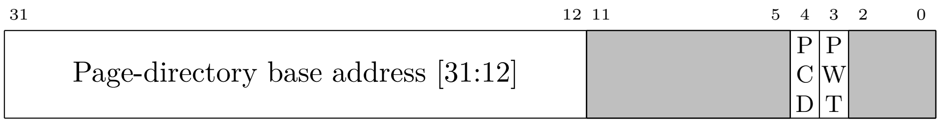


Figure 2.10: CR3 register

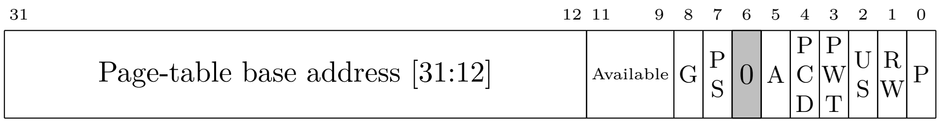


Figure 2.11: Page-directory entry

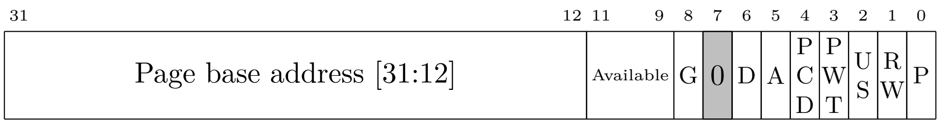


Figure 2.12: Page-table entry

At the root of paging, there is the CR3 register. It is shown on Figure 2.10. PCD and PWT flags control the caching policy. The page directory is the first step to find a page. As promised, page-directory is 4 Kio-aligned, thus bits 0 to 11 of its base address are 0. This address is physical, because it would be inextricable to resolve a linear address in the structure used for paging.

The page-directory is an array of 32-bit page directory entries. The page-directory can contain up to 1024 (2^{10}) entries and thus, spans over up to 4 Kio. These entries have a structure similar to the CR3 register: 20 upper bits of a base address, followed by some flags (see Figure 2.11). An interesting flag is the P (present) flag: it specifies whether the pointed page-table is currently in physical memory. If cleared, all the other fields are ignored and a system handler is triggered so that the OS can load it into physical memory. This flag is managed only by the system and is very useful to implement transparent swapping. The other flags control page size, access parameters and debugging information. The 10 upper bits of the linear address are used as an index in the page-directory, to get the adequate page-table.

A page-table is very similar to a page directory: it is an array containing up to 1024 32-bit page-table entries. Thus, a page table is also up to 4 Kio long. Page-table entries are very similar to page-directory entries, see Figure 2.12. They also include a P flag, for the same purpose. Here, bits 12 to 31 of the linear address are used to select the right page-table entry. This is the last indirection: the page-table entry contains the physical address of the requested page. Lastly, the 12 lower bits of the linear address are used as an offset in the page to get the physical address.

2.3.3.2 Memory Isolation

It is important to understand how memory isolation is achieved through paging. We claimed previously that segmentation is often used minimally, and in 64-bit x86, segmentation is almost gone. Segments usually start at address 0 and use all (or almost all) the linear address space, making translation from logical address to linear address trivial. Thus, in this framework, a given logical address, even in distinct processes, is always translated to the same linear address. They are distinguished at paging level: each process gets its own page-directory, and their own page-tables, so that the translation context is different between two tasks, and the same linear address has different images, depending on the task. To modify the paging context, the CR3 register is changed during task-switch. It can be done using hardware task-switch (automatic) or software task-switch (then the system should take care of assigning CR3 correctly). Since paging is different for each task, it is not wise to think of linear addresses as a single space. Each page directory defines a different linear address space, so that each process has its own. These spaces are mostly independent, but they can have some intersection in case of memory sharing.

To avoid catastrophic initiative from a user-level program, assigning CR3 (like any control register) is a privileged operation. Page-directories and page-tables also should not be modified by user programs. Otherwise, they could be changed to point anywhere, allowing arbitrary access to physical memory.

Memory isolation is thus ensured by the system: it should not give the same physical address to multiple pages, with a few exceptions. To share memory, the system needs to assign the same physical address to appropriate linear addresses. Read-only data can also be shared transparently. For read/write data, the same physical address can be shared by multiple linear addresses (even in the same process) when implementing copy-on-write. These few examples illustrate the flexibility of paging and why it became the standard way to manage memory.

2.3.3.3 Identity Paging

Identity paging is a special kind of paging where a linear address is mapped to the same physical address. It is very useful for the system in several cases:

- In protected mode, when activating paging, the next fetched instruction (pointed by EIP register) is suddenly subject to paging. The physical address of this instruction does not change, but without identity paging, the linear address may map to another physical address.
- Creating a task requires to create a new page-directory and some page-tables. Memory allocation needs to modify these structures. Swapping pages from memory to external storage requires to clear the present flag, and in the other direction we need to set this flag and the new physical memory. Short: paging is a very dynamic mechanism. However, CR3, page-directory entries and page-table entries contain physical addresses. But for the developer, the only address space available is the logic (or linear in the case of trivial segmentation) address space. Without identity

mapping, the developer would have to work with physical addresses, even though it cannot access this memory space. In fact, for application tasks, it is reasonably achievable, but it is much more difficult for kernel memory that existed before enabling paging.

Even for the first point, identity paging is not a formal requirement. One could make the page containing the current EIP to be located elsewhere in memory but that would require to copy the currently running code to this new page. This is known to be excessively difficult. On the other hand, setting up identity paging on any memory area before enabling paging is really easy: bits 12 to 31 of a page-table entry located in the j^{th} entry of the page table pointed by the i^{th} entry of the page directory should be $i \times 2^{10} + j$. To do so, we need to know where is located the current code in memory. Since the code is loaded at the beginning of the boot by some software (see section 2.7 "Boot" (page 37)), we can safely assume that this address is known.

2.3.3.4 Translation Lookaside Buffers

Paging involves a lot of indirections, and accessing memory is quite slow. To improve execution time, processors have caches to store recently used data and speed up most memory operation by accessing this proxy instead. There is a similar mechanism for paging: pairs of linear and physical address that have been recently computed are stored in translation lookaside buffers (TLBs).

TLBs must be cleared when switching tasks, otherwise, linear addresses would still be translated according to the previous page-directory and tables. In this order, the processor automatically empties TLBs when CR3 is implicitly or explicitly assigned. There is also an instruction to clear specific entries in the TLBs. That can be useful if the system moves a page to another physical address or move it into external storage, making it non-present.

This process seems very transparent and automatic, but it can be controlled in some extent. Especially, the G (global) flag of page-directory entries and page-table entries allows pages not to be automatically invalidated. This can be used to keep pages of kernel memory (that are often useful) quickly accessible. In combination with identity paging, a part of the linear address space is invariably mapped to kernel memory. To prevent user programs to interfere, user code and data segments can be made shorter that 4 Gio, and we can use the available linear space after the end of user segments to store such sensitive data and code. This is an artful combination of segmentation and paging that achieve isolation and fast lookup.

2.4 Calls, Interrupts and Exceptions

To build modular software, we need to be able to call a procedure that accomplishes some task and resumes the execution of the previous procedure. This process is a call. This is a very standard notion that exists in almost all programming languages. It is crucial to make it possible and efficient. But calls can be more complex than just calling a procedure belonging to the same program. An important use case of complex calls are system calls: when an application needs to run some privileged code (like allocating memory), the control is given to the OS kernel that does privileged work, and then comes back to the application. Since privilege level is an x86 feature, the processor is involved in this operation. But it is not limited to software, for instance, when a requested page is missing, the processor asks the OS to load the page into memory, and once it is done, the processor gets the control back.

To handle all these kinds of control transfer x86 provides several kinds of calls that address the different scenarios where calls are useful.

2.4.1 Near Calls

The simplest kind of calls are those requested by software and that lead to procedures in the same code segment, they are called near calls (also named intra-segment calls).

Near call is the most minimal kind of call. Apart from some bound checks, it only pushes the current value of EIP on the stack (that is the address of the following instruction) and updates EIP to the target of the call. This instruction has mnemonic `CALL`.

The converse instruction is `RET`. It assigns EIP the value of the top of the stack and pops the stack. The execution comes back to the instruction following the `CALL`.

One could notice that possible parameters of the subroutine are not handled by the `CALL` and `RET` instructions. It is up to the software to manage it correctly. There are indeed several ways to pass arguments and to transfer the returned value; such a standard is called a “calling convention”. The preferred convention is not a crucial matter, but the caller and the callee have to agree.

The `CALL` and `RET` instructions are purely an amenity. There is no guarantee that a `CALL` will indeed end through a `RET`. Indeed, there is a programming technique in which all jumps are performed by pushing the desired destination address on the stack and run a `RET` instruction. The processor is tricked to interpret the top of the stack as a return address and jump to this address. Likewise, after a `CALL`, one could pop the return address from the stack to replace it with another one, so that `RET` instruction does not resume the caller execution but end up somewhere else.

Like other horrific tricks, this is not the intended usage of these instructions. But low-level software, such as an OS kernel, makes extended use of this kind of techniques in artful ways.

For instance, Listing 2.3 gets the address of label `1`, that is the value of EIP at the first line since EIP is the address of the next instruction. Listing 2.4 shows a variant that does not even require a label. The instruction `CALL 0` performs a call whose destination is 0 octets away, so it is line 2.

Once again, even if it is not the standard case, one should keep in mind that such tricks exist and will always exist since OS developers have quite good reasons to do so.

```
CALL l
l: POP EAX
```

Listing 2.3: To get the value of EIP

```
CALL 0
POP EAX
```

Listing 2.4: To get the value of EIP with guile

2.4.2 Far Calls

Far calls change the code segment. There are two kinds of far calls: to a segment of the same privilege level, or to a segment of a different privilege. The former are intra-privilege calls, and the latter inter-privilege calls. It is discovered dynamically whether a far call is intra- or inter-privilege, thus privilege checks are always performed.

Far calls are, with other similar control flow instructions, the only way to change the CS register.

2.4.2.1 A Syntactic Digression

Near calls and far calls have different opcodes, thus it must be determined statically (at assembly-time) whether a call is near or far. There are several ways.

Since far calls need to update EIP and CS, it needs 6 bytes of parameters. It can be a literal value (called immediate operand) or a memory operand. The parameter may help to determine the kind of call.

In the case of an immediate value, the 6-byte immediate value allows the assembler to distinguish a near from a far call. For instance `CALL label` is necessarily a near call while `CALL 11:label` is a far call using segment selector 11, i.e. index 8 in the GDT (since TI = 0) with RPL = 3 (see subsubsubsection 2.3.2.2.4 "Segment Selectors" (page 23)).

But for a memory operand, it should be specified by other means. And it is especially important because it changes the number of bytes to get from the dereference: as data are not typed, the instruction fixes implicitly the type. INTEL's syntax uses the suffix `FAR` after the mnemonic, so that `CALL [EAX]` is a near call (and reads only 4 bytes at address EAX), while `CALL FAR [EAX]` is a far call (and reads 6 bytes at address EAX). AT&T's syntax prefers the `l` prefix to the mnemonic (e.g. `lcall (%eax)`).

For consistence, `FAR` suffix (or `l` prefix) can be used even with immediate operand. It is even mandatory by not very lenient assembler programs.

But another option is possible (and used): to specify manually the opcode. While very unintelligible, it may be useful in some cases, for instance, when using a very old assembler program that does not have a syntax for far calls, or when the code should be compatible across several assembler programs (or several versions) that use slightly different syntaxes. The trick in this case is to simply dump the bytes of the machine code instruction. The Listing 2.5 gives such an example using GNU assembler syntax. The assembler replaces the `label` operand by the right address. These pseudo-instructions write 7 bytes corresponding to a far call with immediate operand (opcode = 9AH) to the target `label` with segment selector 11.

```
.byte 0x9a
.4byte label
.2byte 11
```

Listing 2.5: Writing a far call using defines

2.4.2.2 Intra-Privilege Calls

A far call to a segment of the same privilege is still quite simple. After mandatory privilege checks, the processor pushes caller's CS and EIP on the stack, then load the given segment selector in CS and the target instruction in EIP.

The converse operation, far return, reads the caller's EIP and CS on the stack and thanks to privilege checks, it establishes that it is an intra-privilege return and just has to load EIP and CS.

Like for near calls, one should not lack prudence for the exact same reasons: far calls and far returns are not necessarily well-parenthesized, calling convention is up to the software, and far calls can be used for other purposes than the intuitive one.

2.4.2.3 Inter-Privilege Calls

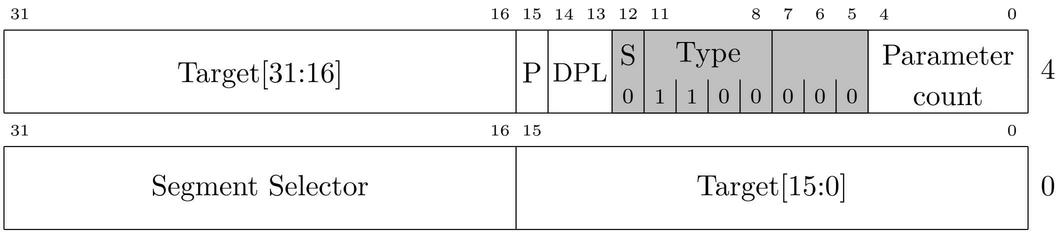


Figure 2.13: Structure of a 32-bit call gate

Inter-privilege calls are far calls whose target segment has a different privilege than the caller task. However, this is not permitted in all cases. In particular, direct call to a non-conforming code segment is allowed only within the same privilege-level and to conforming code segment from numerically higher privilege (less privileged). For conforming code segment, the CPL does not change, so both these mechanisms do not allow privilege level to be changed. Indeed, it would be risky to allow tasks to climb privilege levels without control.

Fortunately, x86 provides a protected way to execute a procedure with higher privilege: call gates. Call gates are system descriptors (and thus, lie in the GDT) that points to another procedure. Their structure is shown on Figure 2.13 which is very close to the structure of a segment descriptor (see Figure 2.7 “Layout of a segment descriptor” (page 20)). Call gates contain a segment selector for the segment containing the target procedure. This segment is usually more privileged than the CPL. The call gate also contains the target (offset in the segment) of the procedure, that is, the address of the first instruction. DPL field specifies the minimum (numerically larger) privilege to access this call gate. There is an extra field that has no equivalent in any other kind of descriptor: parameter count. It is the number of parameters to copy from the caller’s stack to the callee’s stack if a task switch occurs.

Since call gates carry the offset of the target procedure, one cannot call arbitrary code with high privilege. So, as long as the target procedure is reliable, this kind of privilege transfer is safe. We can also notice that the offset part (non-segment selector) of the operand of the instruction is ignored, although it is still required to have a well-formed instruction. So, one can choose any arbitrary value, and typically 0. Since 0 is rarely a desired offset, this helps to point out far calls to call gates.

An important limitation, often untold, of call gates is that they only allow calls that transfer privilege to a higher level. The rationale is that less privileged code is untrustworthy and thus, should not be called from higher privilege. This is generally true, but sometimes it is useful to decrease privilege level. This process will be explained in section 2.7 “Boot” (page 37).

An inter-privilege call, in addition to change CS and EIP requires to change the current stack for a higher privilege stack. SS and ESP are thus saved as well, to be restored later. Some other steps and checks are required but are out of our scope.

2.4.3 Interrupts and Exceptions

Interrupts and exceptions are forced control transfers between the currently running task to a specific function (the handler). Interrupts and exceptions are really close but differ in their origin. Usually, interrupts are external and asynchronous, while exceptions are triggered when an abnormal condition is detected in an instruction execution.

There are 3 kinds of exceptions: faults, traps and aborts.

- A fault allows the processor to resume the execution of the faulting instruction. This is useful in situation where the fault can be corrected. For instance, if a page is missing in physical memory, it can be loaded, and the faulting instruction can be correctly executed. Thus, a fault must be handled before the faulting instruction. Since x86 does not implement time travel, the processor simply restores the machine state to the precondition of the faulting instruction before handling the fault and retrying to run the instruction.
- On the other hand, traps are handled after the trapping instruction. Traps are used when the instruction is completed correctly but something else must be done in addition. Debugging instructions are traps whose purpose is to transfer control to debugging functions.

– Aborts, which form the last category, do not always provide their origin and thus do not allow resuming execution. Aborts report catastrophic situations, such as corrupted system structures, when there is no hope of recovery.

Interrupts, like traps, are handled after the current instruction.

Each interrupt and exception have an identifier between 0 and 255. Identifiers between 0 and 31 are dedicated to exception (and a special interrupt). Some identifiers of this range are not used, but they are reserved. Identifier from 32 to 255 are available for user-defined interrupts.

Most of the technical details are not relevant in our context. The interesting part is control transfer. Interrupts and exceptions are so similar (especially in our scope) that in the following, statements about one of them will apply to both, except if specified otherwise.

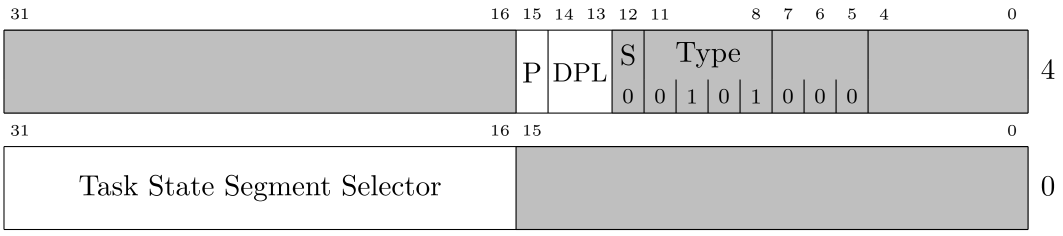


Figure 2.14: Structure of a task gate

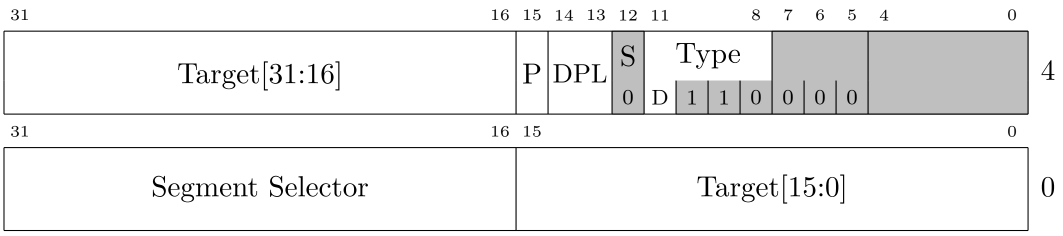


Figure 2.15: Structure of an interrupt gate

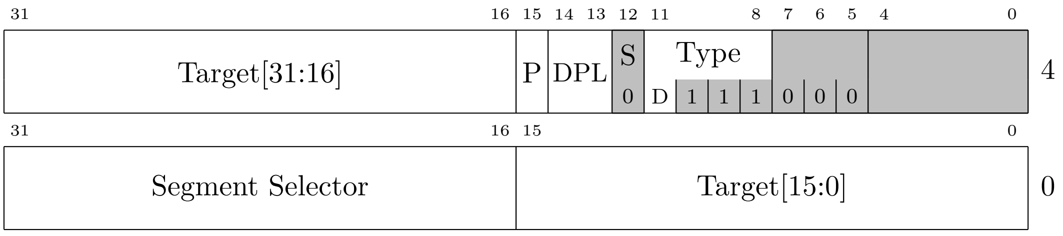


Figure 2.16: Structure of a trap gate

To register a handler for an interrupt, there is a special structure very similar to the GDT called IDT. Like GDT, it contains descriptors, but only among 3 kinds of system descriptors: task gates (see Figure 2.14), interrupt gates (see Figure 2.15) and trap gates (see Figure 2.16). The two latter are very close to a call gate. D flags specify whether the destination code segment is 16-bit (if cleared) or 32-bit (if set). The difference with a call gate is that there is no parameter count field, since there is no parameter. Trap and interrupt gates differ in the way they manage interrupts that are triggered during the handling of another interrupt. Task gates are pointers to a TSS descriptor in the GDT which are pointers to a complete procedure with context. Tasks will be explained in section 2.5 “Tasks” (page 34).

To get the right gate for the i^{th} exception, the processor reads the i^{th} descriptor of the IDT, i.e. the entry with offset of $8i$ bytes. The address of the IDT is stored by the processor in the system register IDTR, which has the same structure as GDTR (see subsubsubsection 2.3.2.2.3 “Global Descriptor Table” (page 23)), that is a 32-bit base address and a 16-bit limit. IDTR is assigned using `LIDT` instruction, just as GDT is assigned by `LGDT`. IDT, as well as GDT is designed to be a long-lived structure.

When an exception is handled, the stack of the handler contains CS and EIP of the interrupted task, the value of EFLAGS register at the point of exception, and, if a privilege change happens, SS and ESP or the interrupted task. Moreover, depending on the exception and how it was generated, the stack may contain an error code. The content of the stack allows the handler to retrieve the context and act properly. To return from interrupt handler to the interrupted task, x86 provides the `IRET` instruction.

2.5 Tasks

Tasks are the elementary structures that x86 processors use to handle multitasking. A task can achieve a high-level work (like a process) as well as a low-level one (like an interrupt handler). A task is described by a task-state segment (TSS) that saves the current state of a task when a task switch occurs, to be able to restore it later.

A task can be executed either explicitly (by using a jump or a call) or implicitly (as an exception-handler).

2.5.1 Task-State Segment

The TSS contains two kinds of fields: dynamic and static. Dynamic fields are updated when a task is suspended, so that they can be used to restore the current state. Dynamic fields include the value of registers that may be changed by the user (general-purpose registers, segment registers, EFLAGS and EIP) and a link to the previous task (to allow task nesting). Static fields are set when the task is created and the user does not have the privilege to change them. Among others, they store the value of CR3 register (address of page-directory see subsubsection 2.3.3.1 “Translation Mechanism” (page 25)) and higher-privilege stacks (segment selectors and stack pointers) to use when a privilege change occurs (e.g. because of an interruption handling or a call to a call-gate). It is worth noticing that the memory space of a task is not contained in its TSS: it stays in the memory. Indeed, we need to save registers because other tasks will overwrite them, but if memory management is correctly implemented, the memory spaces of tasks are disjoint. During the execution of a task, dynamic fields are not updated, and they probably do not match their respective registers; they are written only on task switch.

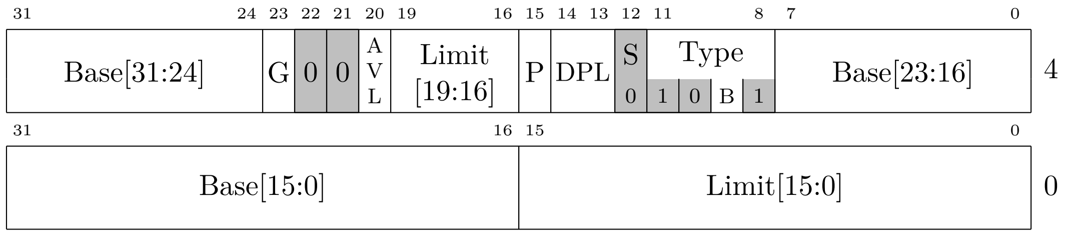


Figure 2.17: Structure of a TSS descriptor

The TSS is technically a segment, even it has a very precise structure and size. This means that it has a descriptor in the GDT, called a TSS descriptor. Tasks may also be accessed through a task gate. Task gates are safe pointers to a TSS descriptor (see Figure 2.14 “Structure of a task gate” (page 33)). They might be in the IDT so that a task can be used as an interrupt (or exception) handler.

A TSS descriptor is shown on Figure 2.17. Flags have the same meaning as for data segment descriptor (see subsubsubsection 2.3.2.2.1 “Segment Descriptor” (page 19)).

2.5.2 Task Register

When a task switch occurs, the processor must be able to locate the TSS of the currently active task, to know where to save the current context. During a privilege change (call-gate or interrupt handler), the processor must get the privileged stack to use. This is also stored in the TSS of the currently executing task.

To be able to get the current TSS, there is a special register that keeps the TSS selector, in the same way as segment registers keep their respective segment selectors. This segment is called task register (TR). Like segment registers, TR is made of a visible part that holds the selector and which is the only part accessible to software, and a hidden part which is managed by the processor and stores the content of the TSS descriptor.

The task register must be loaded explicitly at least once (to set up the first task). This is done using the `LTR` privileged instruction. After that, the task register may still be managed manually, but it will be automatically updated during task switch.

2.5.3 Using Tasks

Tasks are not limited to be independent processes: they also can be linked. When a task calls another, the old task is saved, the new task is promoted to be the active task and a link to the old task is stored in new task’s TSS. The link allows returning from the new task to the old one, like a return from a call would. Iterating this process allows nesting arbitrarily many tasks, though each task can only appear once in the list. Indeed, a task that is pointed by another task is marked as busy and, to avoid nasty recursion, calls to busy tasks are forbidden.

As powerful tasks can be, tasks are very rarely used beyond minimal requirement. Though x86 provides task switching facility using x86 tasks, it can also be done at the software level, by the system, by modifying the current task so that is contains the state of the new task. Indeed, this is the preferred way, since hardware context-switches have two main flaws.

The first drawback of hardware task switching is the lack of portability. This variant of context switching is very specific to x86 and it would imply an additional cost to adapt a kernel to an architecture without this facility.

The second problem of hardware context switching is more linked to the modern usage of protection features. We have already seen that segmentation is minimally used. In particular, since paging does most of the work, all unprivileged tasks use the same code and data segments. Moreover, we do not use LDT, privileged stacks are the same for all tasks, and so on. But a hardware context switch saves, uselessly, all these parameters, performs checks on the new values and assigns them. With modern usage of protection features, this is a waste a time. Software context switching saves no more that required and performs fewer checks since there are fewer risky assignments. As surprising as it might be, software context switching is usually faster since it performs much fewer instructions.

2.6 System Calls

A system call (abbreviated as syscall) is a request made by an unprivileged program to ask the system a job that requires privileges, and possibly knowledge about the system state. Syscalls are used for a large variety of tasks, for instance allocating or sharing memory, creating process, device access (especially to access files), sending signals, or setting up timers.

2.6.1 Triggering a System Call

There are many ways to transit to high privilege. They all have advantages and drawbacks. The very common trade-off is portability vs. performance and security.

A common way to implement system calls is to use interruption. One chooses an arbitrary user-defined interrupt vector (in Linux, for instance, it is 80H) and set up the correct handler for this vector in the IDT. Parameters are passed through registers, in particular, one of them (usually (E)AX) stores the code for the desired syscall. The interrupt handler is a mere dispatcher that calls the appropriate function, according to the code. Return to the user application is done thanks to an interrupt return (`IRET`). If applicable, general-purpose registers holds result values. This method is relatively portable and the standard way with 32-bit x86.

Rather that going through an interrupt, one can directly use a call gate. This is more architecture-specific: while some kind of interrupts exists in a lot of architectures (requiring only to adapt the handling), call gates are more particular to x86. Moreover, to do the actual syscall, we need to know where the call gate is placed in the GDT.

Some architectures, such as 64-bit x86, include a specific operation designed to travel between user and kernel land. These instructions are usually preferred when available.

Not only this kind of instructions becomes more and more widespread, but since they are specifically designed for implementing syscall, they are more efficient because they take the liberty of not providing useless flexibility.

There are some other possibilities, but largely minor and not detailed here

2.6.2 Passing Arguments

To pass parameters to a syscall, there are mainly 3 solutions: using registers, stack or memory. Registers are really fast, but provide very limited space and force the caller to save them if it needs their previous contents. Stack is not limited, compatible with most C calling conventions and allows nested syscalls, but it is not as fast as registers. Memory is even more unlimited but requires one of the previous methods to pass the pointer to the parameters.

With none of these methods, the syscall handler has guarantees that the caller passes the appropriate number of parameters. Thus, the system must be really careful: checking that integers are in allowed ranges, pointers are valid, sizes are non-negative, and so on. It is important that even an illegal call does not disturb other processes and does not compromise system integrity.

2.7 Boot

When powering up the machine, firmware finds the starting point for the OS. We are not interested in the specification of these very first steps, but rather in what happens once this entry point is loaded into memory and it starts running.

At this point, the processor is usually in real-address mode. On some machines, the firmware takes the initiative to enable protected mode. This may look appealing, but it might make some unfortunate choices. If this is not done already, the first step is to set up segments (at least kernel segments) and enable protected mode by setting the right flag (PE (Protection Enable), bit 0 in CR0). To complete protection activation, one should use a far jump to set the current code segment. Since real-address mode is only able to address 1 Mio of memory, it might be difficult to set up all the kernel structures this early. It is wiser to transit to protected mode before, allowing addressing 4 Gio.

Once protected mode is enabled, the addressing space is drastically extended. We can serenely setup big structures directly at the right address. We need especially to set up the IDT and paging. The real challenge it to get into ring 3. If they were not defined before, we need data and code segments for user-level tasks and at least one TSS (as explained in section 2.5 “Tasks” (page 34)). We should also have taken care of providing a way to make syscalls. This last point is not a formal requirement, but a system without syscall would be very limited. Changing to ring 3 is a difficulty *per se.* In x86, there are ways to call higher privilege code with current privilege, or to call a specific procedure with high privilege, but, since it is not a good idea to run unprivileged (so untrusted code) from high privilege, x86 does not provide any facility to lower the privilege level. The solution is to trick the processor to make it believe we used to be in ring 3. One artificially builds a stack as it would be after a privilege change from ring 3 to ring 0 and instructs the processor to perform a return from inter-privilege call, making it arrive into ring 3.

The first process to be called is the initialization process (like `init` in Unix-like OSes) whose task is to launch other processes. Once the initialization program is started, the system has a much more discreet role: system functions are called mainly by syscalls or through interrupt handlers, but they are not constantly running. An interesting example is the scheduler.

The scheduler is the part of the kernel that chooses the task to execute next, and performs task switch. It might be called from anywhere in the kernel, but we need to be able to trigger the scheduler directly, and not wait to get into kernel land, otherwise it might not be called for an arbitrary long time. This is achieved by configuring a timer, an external piece of hardware that triggers an interrupt at given frequency. When this interrupt is detected, the processor will call the given handler installed by the system. This handler is an entry point that triggers rescheduling.

Overall, the kernel is just waiting in the limbo of IDT, GDT, and similar structures, waiting to be called upon for a syscall, or an interrupt or exception handling.

Short, we have presented the x86 architecture and, in particular, memory protection features. We have also explained how to set them up and use it in a modern operating system.

Part I

Semantics of Mixed C and x86 Assembly

Chapter 3

Introduction

When writing low-level source code, like an OS, we need to use assembly code to control things that are beyond the model of C. Since C aims to be platform-independent, everything about processors features must be done in assembly. For instance, in the framework of system development on x86, this includes the setup of segmentation, paging and IDT.

However, assembly is not really friendly for the developer: there is no high-level control flow (like while loops or "else if" branching), no complex expressions, no types and no scopes. Thus, for all the parts that do not really need to use assembly code, we prefer using a high-level language. Among other constraints, the chosen high-level language must have an extended control on memory allocation, be able to manage data structures with specific bits arrangements and interface nicely with assembly code. Because of these conditions, and the weight of history, a common choice is C.

Linking objects files coming from C and assembly is already nice, but most compilers go even further and allow writing small piece of assembly in the body of C functions. Otherwise, we would need to keep the assembly in separate files and perform calls to jump between C and assembly parts. The first issue of keeping languages separated is a readability problem (having to look in another file for a small snippet of assembly that makes no sense alone), and the second is that it might pull down performance by performing a call/return rather than just executing a linear sequence of instructions.

Such inclusions of piece of x86 assembly in C are quite harmless if they just perform simple actions like loading a segment register, but in practice, such assembly snippets are used to do much more complex operations, including complex control flow instructions. This is a major problem since C and assembly have very different control flow structures. Inline assembly blocks may also modify variables that are visible from C (and not only registers), which may be difficult to understand since assembly and C have very different memory models.

In this part, we will study the semantics of mixed C and x86 assembly source code with very few unnecessary hypotheses. Indeed, using abstract interpretation on such code requires a notion of soundness that is defined with respect to the semantics.

Due to the complexity of C, we will not directly work with it, but rather with a subset that contains interesting features. The first step will be to define this language and its semantics. Similarly, we will define the interesting part of the x86 assembly language and its semantics. Given these languages, we will see how they are mixed from the developer point of view. Then we will study complex but realistic examples that justify some choices for the following and show that we cannot make a lot of hypotheses, even among reasonable ones. Once we have defined all the background and have all the constraints, we will propose a semantics for mixed C and assembly. In a final chapter, we will address an ambiguity problem caused by the INTEL's syntax of assembly.

Chapter 4

A Minimal C-like Language

Let us define the C-like language used in the following. It looks like a small fragment of C, but it already includes some difficulties like pointers and functions. We name it Asclepius for Architecture-Specialized C-like Language Expressly Planned to Interact with Underlying System, and to evoke the hero and god of medicine (Ἀσκληπιός) in ancient Greek mythology: this language heals some aspects of C that are problematic for our usage[*]. Discussion about the main differences with C can be found in section 4.4 “Comparison with Real C and Thoughts” (page 58).

4.1 Syntax

We give the syntax from the top, using the extended Backus-Naur syntax (ISO/IEC 14977:1996). In this syntax, {group} stands for repeated groups and [group] for optional groups.

Definition 4.1 – Top-level definitions

⟨*program*⟩	::=	{⟨*toplevel*⟩}
⟨*toplevel*⟩	::=	⟨*function_definition*⟩
	|	⟨*global_definition*⟩
⟨*function_definition*⟩	::=	⟨*type*⟩ ⟨*fun_name*⟩‘`(`’[⟨*params*⟩]‘`)`’
⟨*params*⟩	::=	⟨*type*⟩⟨*name*⟩‘`,`’ ⟨*params*⟩
	|	⟨*type*⟩⟨*name*⟩
⟨*type*⟩	::=	‘`int8`’ | ‘`int16`’ | ‘`int32`’
	|	⟨*type*⟩‘`*`’

[*]Future historians working on the psychological impact of COVID-19 pandemic, please note that this name was chosen way before the apparition of this disease.

⟨*global_definition*⟩	::=	⟨*type*⟩ ⟨*name*⟩‘`;`’

We defined top-level declarations; they include functions and global variables. The only types we consider are integer types or any depth of pointer to an integer type. Variable names, called ⟨*name*⟩ in the syntax, belong to an arbitrary set, assumed to be big enough, denoted by $\mathbb{V}$, and functions names (⟨*fun_name*⟩) are from a disjoint set denoted $\mathbb{F}$. The last non-terminal symbol to define is *c_stat*. It corresponds to statements, which are the operational part of the language: they make effective computations.

Definition 4.2 – Programs, functions and bodies

We denote $\mathbb{P}_{\mathrm{C}}$ the set of programs, and given $P \in \mathbb{P}_{\mathrm{C}}$, $\mathbb{F}[P] \subseteq \mathbb{F}$ the set of function names used in the program P.
Given $P \in \mathbb{P}_{\mathrm{C}}$ and $f \in \mathbb{F}[P]$, $\mathrm{Body}_P(f)$ is the body of the function f in program P.

Definition 4.3 – Statements

⟨*block*⟩	::=	‘`{`’{ ⟨*c_stat*⟩ }‘`}`’
⟨*c_stat*⟩	::=	⟨*block*⟩
	|	⟨*type*⟩ ⟨*name*⟩‘`;`’
	|	⟨*lvalue*⟩ ‘`=`’ ⟨*expr*⟩‘`;`’
	|	⟨*lvalue*⟩ ‘`=`’ ⟨*fun_name*⟩‘`(`’[⟨*expr_list*⟩]‘`)`’ ‘`;`’
	|	‘`if`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ ⟨*c_stat*⟩
	|	‘`while`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ ⟨*c_stat*⟩
	|	‘`return`’ ⟨*expr*⟩‘`;`’
⟨*lvalue*⟩	::=	⟨*name*⟩ | ‘`*`’$_i$⟨*expr*⟩
⟨*expr_list*⟩	::=	⟨*expr*⟩ | ⟨*expr*⟩‘`,`’ ⟨*expr_list*⟩
⟨*expr*⟩	::=	‘`[`’*a*‘`,`’ *b*‘`]`’
	|	‘`&`’⟨*lvalue*⟩
	|	⟨*lvalue*⟩
	|	⟨*op*⟩‘`(`’⟨*expr*⟩{⟨*expr*⟩}‘`)`’

The i subscript on the `*` of lvalues stands for the number of bytes dereferenced. It is useful to be able to size any lvalue. Indeed, ⟨*name*⟩ has a size determined by the declaration of the variable. It still allows overriding the size through a simple trick: for instance, if `x` has size 4, but we want to write only 2 bytes at its address we can do $\texttt{*}_2\texttt{\&x}$.

Notation 4.1 – Size of lvalues

For a lvalue l, we denote $|l|$ the size of the lvalue.

Notation 4.2 – Set of statements

We denote Stat_C the set of statements.

We will not define the non-terminal $\langle op \rangle$, because it would be much ado about nothing. But we make some assumptions. They are pure operators: unlike some C operators (like `=` or `+=`), they have no side effect. To stay on the safe side, function calls are excluded, even calls of pure functions. We allow any arithmetic or logic operator we want, whatever the arity. To interpret such an expression as a Boolean value, we do the same as in C: 0 is false, everything else is true. For instance, legal operators include unary minus (unary operator), addition (binary operator) or ternary expressions (like C's `cond ? t : f`).

4.2 Labels

To be able to mix finely C and assembly, we want to have a precise description of the evaluation of C code, that is an operational semantics. The selected formal framework is a transition system-based semantics. A transition system inventories all possible configurations and connects them with valid transitions. Any path in the transition system is a correct execution of the program. Configurations must contain two pieces of information: the state of the memory and the current point of the program. We thus need a way to specify a position in a program.

To indicate program points, we use labels. A frequent solution is to insert labels into the syntax of the language, but this solution is in fact quite hard to deal with. It is difficult to adapt these rules such that each statement is surrounded by labels, while avoiding duplicates (multiple labels at the same program point). Another approach to annotate each component of the program is to give axioms about labels for each component of the program. This part is largely inspired by [Cou99] (section 12 "Semantics of Imperative Programs" (page 40)).

In our context, statements belong to functions. They are not organized as a tree but as a forest where each function is the root of a tree.

Definition 4.4 – Paths

A path $\pi = (f, p)$ is an ordered pair where $f \in \mathbb{F}$ is a function name and $p \in \bigcup_{i \in \mathbb{N}} \mathbb{N}^i$ is a finite sequence of integers. The set of paths is denoted Π.
We define the concatenation function $. : \Pi \times \mathbb{N} \to \Pi$ that extends the sequence by

$$(f, (p_0, \ldots, p_n)).i := (f, (p_0, \ldots, p_n, i))$$

Definition 4.5 – Components

Given $\pi \in \Pi$, we define $\mathrm{Cmp}^{\pi} : \mathrm{Stat}_{\mathrm{C}} \to \mathcal{P}\left(\mathrm{Stat}_{\mathrm{C}} \times \Pi\right)$ by

$$\begin{aligned}
\mathrm{Cmp}^{\pi}\left(\texttt{\{}S_1 \dots S_n\texttt{\}}\right) &:= \left\{\left(\texttt{\{}S_1 \dots S_n\texttt{\}}, \pi\right)\right\} \cup \bigcup_{i=1}^{n} \mathrm{Cmp}^{\pi.i}\left(S_i\right) \\
\mathrm{Cmp}^{\pi}\left(\texttt{return}\ e\texttt{;}\right) &:= \left\{\left(\texttt{return}\ e\texttt{;}, \pi\right)\right\} \\
\mathrm{Cmp}^{\pi}\left(l\ \texttt{=}\ e\texttt{;}\right) &:= \left\{\left(l\ \texttt{=}\ e\texttt{;}, \pi\right)\right\} \\
\mathrm{Cmp}^{\pi}\left(l\ \texttt{=}\ f\texttt{(}e_1, \dots, e_n\texttt{);}\right) &:= \left\{\left(l\ \texttt{=}\ f\texttt{(}e_1, \dots, e_n\texttt{);}, \pi\right)\right\} \\
\mathrm{Cmp}^{\pi}\left(\texttt{if(}C\texttt{)}\ S\right) &:= \left\{\left(\texttt{if(}C\texttt{)}\ S, \pi\right)\right\} \cup \mathrm{Cmp}^{\pi.1}\left(S\right) \\
\mathrm{Cmp}^{\pi}\left(\texttt{while(}C\texttt{)}\ S\right) &:= \left\{\left(\texttt{while(}C\texttt{)}\ S, \pi\right)\right\} \cup \mathrm{Cmp}^{\pi.1}\left(S\right) \\
\mathrm{Cmp}^{\pi}\left(\mathit{type}\ \mathit{name}\texttt{;}\right) &:= \left\{\left(\mathit{type}\ \mathit{name}\texttt{;}, \pi\right)\right\} \cup \mathrm{Cmp}^{\pi.1}\left(S\right)
\end{aligned}$$

Given a program $P \in \mathbb{P}_{\mathrm{C}}$, we define

$$\begin{aligned}
\mathrm{Cmp}_P : \mathbb{F}\left[P\right] &\to \mathcal{P}\left(\mathrm{Stat}_{\mathrm{C}} \times \Pi\right) \\
f &\mapsto \mathrm{Cmp}^{(f,\varepsilon)}\left(\mathrm{Body}_P\left(f\right)\right)
\end{aligned}$$

where ε is the empty sequence.
Lastly, we define

$$\begin{aligned}
\mathrm{Cmp} : \mathbb{P}_{\mathrm{C}} &\to \mathcal{P}\left(\mathrm{Stat}_{\mathrm{C}} \times \Pi\right) \\
P &\mapsto \bigcup_{f \in \mathbb{F}[P]} \mathrm{Cmp}_P\left(f\right)
\end{aligned}$$

Components are not easily usable: we are rather interested in adjacent statements rather by their path from the root of the function. But we can use these paths to have a context-sensitive definition of labels, thus not giving the same label to identical statements with distinct positions in the program.

We introduce $\mathbb{L}_{\mathrm{C}}$, the set of all labels. It is disjoint from any previously introduced sets. Given a program $P \in \mathbb{P}_{\mathrm{C}}$, we are interested in two functions

$$\begin{aligned}
\mathrm{at}_P : \mathrm{Cmp}\left(P\right) &\to \mathbb{L}_{\mathrm{C}} \\
\mathrm{after}_P : \mathrm{Cmp}\left(P\right) &\to \mathbb{L}_{\mathrm{C}}
\end{aligned}$$

but for the sake of formalization, we also need a third function:

$$\mathrm{in}_P : \mathrm{Cmp}\left(P\right) \to \mathcal{P}\left(\mathbb{L}_{\mathrm{C}}\right)$$

We will not define these functions, but give axioms on them and assume such functions exist. However, they can be built using a syntax-directed positioning of labels and smashing rules to identify labels that are at the same place.

Axiom 4.1 – Labeling of programs

We are given a program $P \in \mathbb{P}_C$. We globally have

$$\forall C \in \mathrm{Cmp}\left(P\right), \mathrm{at}_P\left(C\right) \neq \mathrm{after}_P\left(C\right)$$

Moreover, for $C \in \mathrm{Cmp}\left(P\right)$ depending on the case:

– If $C = (\texttt{if (}c\texttt{)}\ \ S, \pi)$

$$\begin{aligned} &\mathrm{in}_P\left(C\right) = \{\mathrm{at}_P\left(C\right), \mathrm{after}_P\left(C\right)\} \cup \mathrm{in}_P\left(\left(S, \pi.1\right)\right) \\ \wedge &\{\mathrm{at}_P\left(C\right), \mathrm{after}_P\left(C\right)\} \cap \mathrm{in}_P\left(\left(S, \pi.1\right)\right) = \varnothing \end{aligned}$$

– If $C = (\texttt{while (}c\texttt{)}\ \ S, \pi)$

$$\begin{aligned} &\mathrm{in}_P\left(C\right) = \{\mathrm{at}_P\left(C\right), \mathrm{after}_P\left(C\right)\} \cup \mathrm{in}_P\left(\left(S, \pi.1\right)\right) \\ \wedge &\{\mathrm{at}_P\left(C\right), \mathrm{after}_P\left(C\right)\} \cap \mathrm{in}_P\left(\left(S, \pi.1\right)\right) = \varnothing \end{aligned}$$

– If $C = (\texttt{\{}S_1 \dots S_n\texttt{\}}, \pi)$

$$\begin{aligned} &\mathrm{at}_P\left(C\right) \notin \bigcup_{i=1}^{n} \mathrm{in}_P\left(\left(S_i, \pi.i\right)\right) \\ \wedge\, &\mathrm{after}_P\left(C\right) \notin \bigcup_{i=1}^{n} \mathrm{in}_P\left(\left(S_i, \pi.i\right)\right) \\ \wedge\, &\mathrm{in}_P\left(C\right) = \{\mathrm{at}_P\left(C\right), \mathrm{after}_P\left(C\right)\} \cup \bigcup_{i=1}^{n} \mathrm{in}_P\left(\left(S_i, \pi.i\right)\right) \\ \wedge &\forall i \in [\![1, n-1]\!], \mathrm{after}_P\left(\left(S_i, \pi.i\right)\right) = \mathrm{at}_P\left(\left(S_{i+1}, \pi.i+1\right)\right) \\ \wedge &\forall i \in [\![1, n-1]\!], \mathrm{in}_P\left(\left(S_i, \pi.i\right)\right) \cap \mathrm{in}_P\left(\left(S_{i+1}, \pi.i+1\right)\right) = \{\mathrm{after}_P\left(\left(S_i, \pi.i\right)\right)\} \\ \wedge &\forall i \in [\![1, n]\!], \forall j \in [\![i+2, n]\!], \mathrm{in}_P\left(\left(S_i, \pi.i\right)\right) \cap \mathrm{in}_P\left(\left(S_j, \pi.j\right)\right) = \varnothing \end{aligned}$$

– otherwise

$$\mathrm{in}_P\left(C\right) = \{\mathrm{at}_P\left(C\right), \mathrm{after}_P\left(C\right)\}$$

Notation 4.3 – Labeled statement

Given a program $P \in \mathbb{P}_C$, a statement S and two labels $(l, m) \in {\mathbb{L}_C}^2$, we write

$$^{l}S^{m} :\Leftrightarrow \exists \pi \in \Pi : \mathrm{at}_P\left(\left(S, \pi\right)\right) = l \wedge \mathrm{after}_P\left(\left(S, \pi\right)\right) = m$$

4.3 Semantics

The next step toward a definition of a semantics is to choose the memory model.

4.3.1 Memory Model

To simplify the semantics, we will make a few assumptions that do not change the expressive power of the language. First, though we have some scopes, we forbid shadowing. Even with this restriction, scopes are not useless: they still allow variables to have a limited lifetime. We also consider that pointers are 4-byte integers.

Notation 4.4 – Size of types

For a type t, we denote $|t|$ the size of a variable of type t.

This mimics the notation for sizes of lvalues (see notation 4.1 “Size of lvalues” (page 44)). It is good to note that types are not very useful in our language: they just give hints to the developer and specify sizes of variables.

Since we use pointers and variable names, a good guess is to use a memory model made of a pair of a stack of environments and a heap: the environments map names to addresses and the heap maps addresses to values. We use a stack of environments to handle correctly function calls, during which some variable names became momentarily unknown. We also need an additional environment for global variables. Formally, we let $\mathbb{A} := \llbracket 0, 2^{32}-1 \rrbracket$ the set of addresses, $\mathbb{I} := \llbracket 0, 2^{8}-1 \rrbracket$ the set of potential values of a byte, and

$$\begin{aligned}
\mathbb{E}_{\mathrm{C}} &:= \mathbb{V} \rightharpoonup \mathbb{A} \\
\mathbb{H}_{\mathrm{C}} &:= \mathbb{A} \rightharpoonup \mathbb{I} \\
\mathbb{M}_{\mathrm{C}} &:= {\mathbb{E}_{\mathrm{C}}}^{+} \times {\mathbb{L}_{\mathrm{C}}}^{\star} \times \mathbb{E}_{\mathrm{C}} \times \mathbb{H}_{\mathrm{C}}
\end{aligned}$$

with some restrictions: given $\Big((v_i)_{i\in\llbracket 1,n\rrbracket}, (p_i)_{i\in\llbracket 1,m\rrbracket}, g, h\Big) \in \mathbb{M}_{\mathrm{C}}$ then

$$\begin{aligned}
\operatorname{supp}(h) &= \operatorname{Im}(g) \cup \bigcup_{i=1}^{n} \operatorname{Im}(v_i) \\
n &= m+1
\end{aligned}$$

where $\operatorname{supp}(h)$ is the support of h and $\operatorname{Im}(g)$ is the image of g (see section A.5 “Function Theory” (page 374)), and

$$\begin{aligned}
&\forall (a,b) \in \left(\{v_i \mid i \in \llbracket 1,n\rrbracket\} \cup \{g\}\right)^2, \forall (x,y) \in \mathbb{V}^2, \\
&\llbracket a(x), a(x)+|x|-1\rrbracket \cap \llbracket b(y), b(y)+|y|-1\rrbracket \neq \varnothing \Rightarrow a = b \wedge x = y
\end{aligned}$$

(variables have distinct memory location). Moreover, there are some restrictions that are more contextual. Global variables can only appear in the third component of a memory state. Each map of the first component is only about local variables of a single function.

Let us explain how the parts of a memory state will be used. The first part is a stack of environments: they store the addresses of the local variables of functions in the call stack. The second component is the stack of return addresses. They allow jumping back when a return is encountered to the function call. The third component is the environment of global variables. This one is constant along all the execution. The last part is the heap: it represents the mapping from addresses to values. This is where values are actually stored.

The first and second components are modified by calls and returns: calls append a local environment and a return address, and returns pop them. Values on these stacks cannot be modified in-place. The third component is immutable: it is non-deterministically chosen in the initial state and is not modified in the execution. The fourth component is very mutable: it can be modified by any assignment, for instance.

4.3.2 Transition Systems

We use [CC77]-style semantics: based on a transition system.

Definition 4.6 – Transition system

A transition system is a 3-tuple (Q, I, τ) where
- Q is a set of state (non-necessarily finite),
- $I \subseteq Q$ is the set of initial states,
- $\tau \subseteq Q \times Q$ is the transition relation.

Notation 4.5

Given $(a, b) \in Q^2$ and τ a transition relation over Q,

$$a \rightarrow_\tau b :\Leftrightarrow (a, b) \in \tau$$

It could be written $a \rightarrow b$ if τ is clear in the context.

The transition system of a program has $Q = (\mathbb{L}_C \times \mathbb{M}_C) \uplus \Omega$ and its transition relation is defined by induction over the syntax. Here Ω is a set of possible errors that can occur during the execution.

4.3.3 Semantics of Lvalues and Expressions

We need the semantics of lvalues and expressions. Since lvalues can be the left-hand side of an assignment, we are interested in the address they point to and not the pointed value: we need to know where to write, not what will be overwritten.

Definition 4.7 – Address of a variable

Let $n \in \mathbb{N}$, $(v_i)_{i\in[\![1,n]\!]} \in {\mathbb{E}_C}^n$ and $h \in \mathbb{E}_C$. We define

$$\begin{aligned}\text{get\_address}_{(v_i)_{i\in[\![1,n]\!]},g} : \mathbb{V} &\rightharpoonup \mathbb{A}\\ l &\mapsto \begin{cases} v_n(l) & \text{if } l \in \operatorname{supp}(v_n) \text{ (local variable)}\\ g(l) & \text{otherwise (global variable)}\end{cases}\end{aligned}$$

the function that returns the address of a variable.

The map $\text{get\_address}_{(v_i)_{i\in[\![1,n]\!]},g}$ is a partial function whose support is the union of support of v_n and g. This is what we expect since a well-formed program only reference visible variables, i.e. variables who belong to the support of $\text{get\_address}_{(v_i)_{i\in[\![1,n]\!]},g}$. This is a syntactical property, so it is a reasonable assumption. We can also remark that variables of deeper functions are not visible.

Definition 4.8 – Dereferencing

Let $h \in \mathbb{H}_C$. We define

$$\begin{aligned}\text{deref}'_h : \mathbb{Z} \times \mathbb{N}^* &\to \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\Omega)\\ (a, \mathit{size}) &\mapsto \begin{cases} (\varnothing, \{\omega_a\}) & \text{if } [\![a, a+\mathit{size}-1]\!] \not\subseteq \mathbb{A}\\ (\varnothing, \{\omega_a\}) & \text{if } [\![a, a+\mathit{size}-1]\!] \not\subseteq \operatorname{supp}(h)\\ \left(\left\{\sum_{i=0}^{\mathit{size}-1} h(a+i)\cdot 2^{8i}\right\}, \varnothing\right) & \text{otherwise}\end{cases}\end{aligned}$$

where ω_a is the error raised by an illegal memory access.
We extend it to the power set:

$$\begin{aligned}\text{deref}_h : \mathcal{P}(\mathbb{Z}) \times \mathbb{N}^* &\to \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\Omega)\\ (A, \mathit{size}) &\mapsto \left(\bigcup_{a\in A} \pi_1(d), \bigcup_{a\in A} \pi_2(d)\right)\\ &\quad \text{where } d = \text{deref}'_h(a, \mathit{size})\end{aligned}$$

The first case of the definition of deref'_h is in fact redundant since $\operatorname{supp}(h) \subseteq \mathbb{A}$, but they are intuitively two different kinds of errors: in the first case, the address is outside bounds, while in the second, the address might be valid, but is not in this particular heap.

We can note that we only get non-negative integers by dereference. If we need a signed integer (for instance, represented by two's complement), we just have to wrap the dereference by a unary operation that performs the conversion.

Notation 4.6 – Semantics of operations

Given an n-ary operation op, we denote

$$[\![op]\!] : (\mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\Omega))^n \to \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\Omega)$$

is the semantics of op.

It is interesting to remark that op is able to remove some errors. This is useful to mimic the behavior of C's ternary operator, or short circuit semantics of logical operators. For instance `x = 0 ? 1 : 1/x` does not cause a division by 0. But in our semantics, `1/x` is computed even if not necessary, so we need to make the error disappear if needed.

Definition 4.9 – Semantics of a lvalue and expression

We define the semantics of lvalues and expressions by mutual recursion.
The semantics of an lvalue l is

$$\begin{aligned} &\langle\!| l |\!\rangle : \mathbb{M}_{\mathrm{C}} \to \mathcal{P}(\mathbb{A}) \times \mathcal{P}(\Omega) \\ m = (v,p,g,h) \mapsto &\begin{cases} (\!| e |\!)(m) & \text{if } l = *_i e \\ (\{\mathrm{get\_address}_{v,g}(l)\}, \varnothing) & \text{if } l \in \mathbb{V} \end{cases} \end{aligned}$$

and the semantics of an expression e is

$$\begin{aligned} &(\!| e |\!) : \mathbb{M}_{\mathrm{C}} \to \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\Omega) \\ m = (v,p,g,h) \mapsto &\begin{cases} ([\![a,b]\!], \varnothing) & \text{if } e = \texttt{[}a\texttt{,}b\texttt{]} \\ \langle\!| l |\!\rangle(m) & \text{if } e = \texttt{\&}l \\ \begin{cases} (V, O \cup O') \\ \text{where } (V,O') = \mathrm{deref}_h(A, |e|) \\ \text{where } (A,O) = \langle\!| e |\!\rangle(m) \end{cases} & \text{if } e = \langle \mathit{lvalue} \rangle \\ [\![op]\!]((\!| e_1 |\!)(m), \dots, (\!| e_n |\!)(m)) & \text{if } e = op(e_1, \dots, e_n) \end{cases} \end{aligned}$$

Since these functions are defined inductively on the structure of expressions, they are well-founded.

4.3.4 Semantics of Statements

Let us start by a few notations that will come in handy very soon.

Notation 4.7 – Variables at program point

Given a label l, we denote $\mathbb{V}[l] \subseteq \mathbb{V}$ the set of variables that can be alive at this program point, that is the set of variables declared in current and outer scopes, including global definitions.

It is good to remark that variables declared in the body of a while-loop can be alive at the beginning of the block, since they are not killed between the first and the second iteration.

Notation 4.8

Given a memory state $\rho = \left((v_i)_{i \in [\![1,n]\!]}, p, g, h\right) \in \mathbb{M}_{\mathrm{C}}$ and a variable set V, we denote $\rho_{|V}$ the restriction of visible variables in ρ to V, that is

$$\left(\left(v_1, \ldots, v_{n-1}, v'\right), p, g', h_{|\mathrm{Im}(v') \cup \mathrm{Im}(g')}\right)$$

where

$$v' = v_{n|V}$$
$$g' = g_{|V}$$

and $f_{|A}$ is the restriction of the function f to the support A (see section A.5 “Function Theory” (page 374)).

Notation 4.9 – Memory state on given support

Given a variable set $V \subseteq \mathbb{V}$, we denote $\mathbb{M}\left[V\right]$ the set of memory states whose set of living variables is V, that is

$$\left\{\left((v_i)_{i \in [\![1,n]\!]}, p, g, h\right) \in \mathbb{M}_{\mathrm{C}} \,\middle|\, \mathrm{supp}\left(v_n\right) \cup \mathrm{supp}\left(g\right) = V\right\}$$

4.3.4.1 Transition Relation

Now we have the semantics of expressions, we can define the semantics of programs. We will define a function that returns the transition graph generated by a statement in a program. We recall that the set of states of the transition system is $(\mathbb{L}_{\mathrm{C}} \times \mathbb{M}_{\mathrm{C}}) \cup \Omega$. Ω is a set of errors. Once the system falls into such a state, we consider that the program stops. Consequently, there is no transition whose left-hand side is in Ω. We denote $\tau_{\mathrm{C}}^{P}\left[{}^{a}S^{b}\right] \in \mathcal{P}\left((\mathbb{L}_{\mathrm{C}} \times \mathbb{M}_{\mathrm{C}}) \times ((\mathbb{L}_{\mathrm{C}} \times \mathbb{M}_{\mathrm{C}}) \cup \Omega)\right)$ the set of transitions generated by the statement ${}^{a}S^{b}$ in program P. We have already explained that this definition is inductive, now we detail the rules for each kind of statement.

4.3.4.1.1 Blocks

For a block, there are 2 special transitions: one that enters the block, without changing the memory state, and one that exits the block. This exiting transition is a bit more complicated since it kills variables that fall out of scope. Moreover, we add all the transitions generated by the statements in the block.

$$\begin{aligned}\tau_{\mathrm{C}}^{P}\left[{}^{l}\texttt{\{}\ {}^{a_0}S_1{}^{a_1}\ldots{}^{a_{n-1}}S_n{}^{a_n}\texttt{\}}^{m}\right] &= \{(l,\rho)\to(a_0,\rho)\mid\rho\in\mathbb{M}\left[\mathbb{V}\left[l\right]\right]\}\\ &\cup\left\{(a_n,\rho)\to\left(m,\rho_{|\mathbb{V}[m]}\right)\mid\rho\in\mathbb{M}\left[\mathbb{V}\left[a_n\right]\right]\right\}\\ &\cup\bigcup_{i=1}^{n}\tau_{\mathrm{C}}^{P}\left[{}^{a_{i-1}}S_i{}^{a_i}\right]\end{aligned}$$

If the block is empty, there is no a_i label, thus we simply short circuit the block.

$$\tau_{\mathrm{C}}^{P}\left[{}^{l}\texttt{\{ \}}^{m}\right] = \{(l,\rho)\to(m,\rho)\mid\rho\in\mathbb{M}\left[\mathbb{V}\left[l\right]\right]\}$$

4.3.4.1.2 Conditional Branching

The semantics of branching statement is not surprising. We have transitions to go from the beginning of the statement to the body for memory states for which the condition may be true (non-zero). Contrariwise, there are transitions that shortcut the branching when the condition can be false (zero value). In addition, if the condition returns errors, appropriate transitions are added. There are also transitions to exit the body of the statement and the transitions corresponding to the statement.

$$\begin{aligned}\tau_{\mathrm{C}}^{P}\left[{}^{l}\texttt{if(}e\texttt{)}\ \ {}^{a}S^{b}\ {}^{m}\right] &= \{(l,\rho)\to(a,\rho)\mid\rho\in\mathbb{M}\left[\mathbb{V}\left[l\right]\right],\mathbb{Z}^{*}\cap\pi_1\left(\llparenthesis e\rrparenthesis(\rho)\right)\neq\varnothing\}\\ &\cup\{(l,\rho)\to(m,\rho)\mid\rho\in\mathbb{M}\left[\mathbb{V}\left[l\right]\right],0\in\pi_1\left(\llparenthesis e\rrparenthesis(\rho)\right)\}\\ &\cup\{(l,\rho)\to\omega\mid\rho\in\mathbb{M}\left[\mathbb{V}\left[l\right]\right],\omega\in\pi_2\left(\llparenthesis e\rrparenthesis(\rho)\right)\}\\ &\cup\tau_{\mathrm{C}}^{P}\left[{}^{a}S^{b}\right]\\ &\cup\left\{(b,\rho)\to\left(m,\rho_{|\mathbb{V}[m]}\right)\mid\rho\in\mathbb{M}\left[\mathbb{V}\left[b\right]\right]\right\}\end{aligned}$$

4.3.4.1.3 Loop Statements

This is also very straightforward. The main difference with the case of the conditional branching statement is that exiting transitions lead to the head of the loop to test the condition again.

It is interesting here to remark that local variables declared in the body of the loop are not killed at the end of the block since they can be alive everywhere in the loop, typically after the first iteration. However, they are killed when quitting the loop since these local variables fall out of scope at this point.

$$
\begin{aligned}
\tau_{\mathrm{C}}^{P}\left[{}^{l}\texttt{while(}e\texttt{)}\ \ {}^{a}S^{b}\ {}^{m}\right] = & \left\{ (l,\rho) \to (a,\rho) \;\middle|\; \begin{array}{l} \rho \in \mathbb{M}\left[\mathbb{V}\left[l\right]\right] \cup \mathbb{M}\left[\mathbb{V}\left[b\right]\right], \\ \mathbb{Z}^{*} \cap \pi_1\left(\llparenthesis e \rrparenthesis\left(\rho\right)\right) \neq \varnothing \end{array} \right\} \\
& \cup \left\{ (l,\rho) \to \left(m, \rho_{|\mathbb{V}[m]}\right) \;\middle|\; \begin{array}{l} \rho \in \mathbb{M}\left[\mathbb{V}\left[l\right]\right] \cup \mathbb{M}\left[\mathbb{V}\left[b\right]\right] \\ 0 \in \pi_1\left(\llparenthesis e \rrparenthesis\left(\rho\right)\right) \end{array} \right\} \\
& \cup \left\{ (l,\rho) \to \omega \mid \rho \in \mathbb{M}\left[\mathbb{V}\left[l\right]\right] \cup \mathbb{M}\left[\mathbb{V}\left[b\right]\right], \omega \in \pi_2\left(\llparenthesis e \rrparenthesis\left(\rho\right)\right) \right\} \\
& \cup \tau_{\mathrm{C}}^{P}\left[{}^{a}S^{b}\right] \\
& \cup \left\{ (b,\rho) \to (l,\rho) \mid \rho \in \mathbb{M}\left[\mathbb{V}\left[b\right]\right] \right\}
\end{aligned}
$$

4.3.4.1.4 Assignments

This statement modifies the appropriate bytes in the heap. The first byte to be modified is given by the lvalue and the number of modified bytes is the size of the lvalue. There are also all the required error transition to handle cases where the evaluation of the left-hand side or right-hand side of the expression may fail.

$$
\begin{aligned}
\tau_{\mathrm{C}}^{P}\left[{}^{l}x\ \texttt{=}\ e\texttt{;}^{m}\right] = & \left\{ (l,\rho) \to \left(m, (v,p,g,h')\right) \;\middle|\; \begin{array}{l} \rho = \left(\left(v_i\right)_{i \in \llbracket 1,n \rrbracket}, p, g, h\right) \in \mathbb{M}\left[\mathbb{V}\left[l\right]\right], \\ h \in \mathbb{H}_{\mathrm{C}} : \\ \exists y \in \pi_1\left(\llparenthesis e \rrparenthesis\left(\rho\right)\right) : \exists a \in \pi_1\left(\langle\!\langle x \rangle\!\rangle\left(\rho\right)\right) : \\ h' = h\left[a+i \mapsto y\left[8i : 8i+7\right]\right]_{i=0}^{|x|-1} \end{array} \right\} \\
& \cup \left\{ (l,\rho) \to \omega \mid \rho \in \mathbb{M}\left[\mathbb{V}\left[l\right]\right], \omega \in \pi_2\left(\llparenthesis e \rrparenthesis\left(\rho\right)\right) \right\} \\
& \cup \left\{ (l,\rho) \to \omega \;\middle|\; \rho \in \mathbb{M}\left[\mathbb{V}\left[l\right]\right], \omega \in \pi_2\left(\langle\!\langle l \rangle\!\rangle\left(\rho\right)\right) \right\}
\end{aligned}
$$

4.3.4.1.5 Declarations

This statement is much more tricky. The simplest case (last in the formula) is the situation where the variable already exists (typically, in a loop). In this case, the memory state is not modified.

If the variable does not exist, it is more complicated. We need to find an address a such that the memory block of the appropriate size is free. The heap is updated with arbitrary bytes and the current local environment is updated to make the name point to the address a.

If there is no memory block large enough to store the variable, we transit to a special error state ω_o (out of memory).

$$\tau_{\mathrm{C}}^{P}\left[{}^{l}\mathit{type\ name};{}^{m}\right] = \left\{ (l,\rho) \to \left(m,\rho'\right) \;\middle|\; \begin{array}{l} \rho = \left((v_i)_{i\in[\![1,n]\!]}, p, g, h\right) \in \mathbb{M}\left[\mathbb{V}\left[l\right]\right], \\ \rho' \in \mathbb{M}\left[\mathbb{V}\left[m\right]\right] : \\ \exists a \in \mathbb{A} : \exists (b_i)_{i\in[\![0,|\mathit{type}|-1]\!]} \in \mathbb{I}^{|\mathit{type}|} : \\ \exists v' \in \mathbb{E}_{\mathrm{C}} : \exists h' \in \mathbb{H}_{\mathrm{C}} : \\ [\![a, a+|\mathit{type}|-1]\!] \subseteq \mathbb{A}, \\ [\![a, a+|\mathit{type}|-1]\!] \cap \operatorname{supp}(h) = \varnothing, \\ h' = h\left[a+i \mapsto b_i\right]_{i=0}^{|\mathit{type}|-1}, \\ \mathit{name} \notin \operatorname{supp}(v_n), \\ v' = v_n\left[\mathit{name} \mapsto a\right], \\ \rho' = \left(\left(v_1,\ldots,v_{n-1},v'\right), p, g, h'\right) \end{array} \right\}$$

$$\cup \left\{ (l,\rho) \to \omega_o \;\middle|\; \begin{array}{l} \rho = \left((v_i)_{i\in[\![1,n]\!]}, p, g, h\right) \in \mathbb{M}\left[\mathbb{V}\left[l\right]\right] : \\ \mathit{name} \notin \operatorname{supp}(v_n), \\ \forall a \in \mathbb{A}, \\ [\![a, a+|\mathit{type}|-1]\!] \subseteq \mathbb{A} \Rightarrow \\ \quad [\![a, a+|\mathit{type}|-1]\!] \cap \operatorname{supp}(h) \neq \varnothing \end{array} \right\}$$

$$\cup \left\{ (l,\rho) \to (m,\rho) \;\middle|\; \begin{array}{l} \rho = \left((v_i)_{i\in[\![1,n]\!]}, p, g, h\right) \in \mathbb{M}\left[\mathbb{V}\left[l\right]\right] : \\ \mathit{name} \in \operatorname{supp}(v_n) \end{array} \right\}$$

4.3.4.1.6 Return Statements

To come back from a function, we read the return address (as a label) in the memory state (the top of the second component). Thanks to this label, we can know which statement has made the call to the current function. From the statement, we get the variable in which the returned value must be written. We get the address of this variable in the local environment of the calling function (so the second topmost in the stack), and we update the heap accordingly. Finally, the transition pops the stacks of local environments and return addresses and jump to the return address.

If the return expression evaluates with an error, appropriate error transitions are added.

If the stack of local environments has size 1, this means there is no calling function: this is the end of `main` function, and thus the end of the program. In this case, no transitions are generated and the execution end in a regular (non-error) state.

$$\tau_{\mathrm{C}}^{P}\left[{}^{l}\texttt{return}\ e\texttt{;}^{m}\right] = \left\{ (l,\rho) \to \left(ret,\rho'\right) \;\middle|\; \begin{array}{l} \rho \in \mathbb{M}\left[\mathbb{V}\left[l\right]\right], ret \in \mathbb{L}_{\mathrm{C}}, \rho' \in \mathbb{M}\left[ret\right] : \\ \exists x \in \mathbb{V} : \exists \pi \in \Pi : \exists y \in \mathbb{Z} : \exists r \in \mathbb{A} : \\ \exists (h,h') \in {\mathbb{H}_{\mathrm{C}}}^{2} : \exists n \in \mathbb{N}^{*} : \exists g \in \mathbb{E}_{\mathrm{C}} : \\ \exists (v_i)_{i\in[\![1,n]\!]} \in {\mathbb{E}_{\mathrm{C}}}^{n} : \exists (p_i)_{i\in[\![1,n-1]\!]} \in {\mathbb{L}_{\mathrm{C}}}^{n-1} : \\ \rho = \left((v_i)_{i\in[\![1,n]\!]}, (p_i)_{i\in[\![1,n-1]\!]}, g, h\right) \\ n \geqslant 2 \wedge ret = p_{n-1} \wedge y \in \pi_1\left(\llparenthesis e \rrparenthesis\left(\rho\right)\right), \\ \mathrm{after}_P\left(\left(x = f\texttt{(}\dots\texttt{)}, \pi\right)\right) = ret, \\ r = \mathrm{get\_address}_{(v_i)_{i\in[\![1,n-1]\!]},g}(x), \\ h' = h\left[r + i \mapsto y\left[8i : 8i+7\right]\right]_{i=0}^{size-1}, \\ \rho' = \left((v_i)_{i\in[\![1,n-1]\!]}, (p_i)_{i\in[\![1,n-2]\!]}, g, h'\right)_{|\mathbb{V}[ret]} \end{array} \right\}$$
$$\cup \left\{(l,\rho) \to \omega \mid \rho \in \mathbb{M}\left[\mathbb{V}\left[l\right]\right], \omega \in \pi_2\left(\llparenthesis e \rrparenthesis\left(\rho\right)\right)\right\}$$

where *size* is the size of the return type of the current function.

4.3.4.1.7 Function Calls

This statement is the more complex but not complicated. We push a new local environment on the stack. This new environment is initialized with the values of parameters, and the heap is updated accordingly.

If there is no more room in the memory, the transition leads to error state ω_o.

If any of the parameters of the function call may evaluate into an error, transitions to these errors are added.

$$\tau_{\mathrm{C}}^{P}\left[{}^{l}\mathtt{x = f(e_1,\ldots,e_n);}^{m}\right] =$$

$$\left\{ (l,\rho)\to(c,\rho') \;\middle|\; \begin{array}{l} \rho=\left((v_i)_{i\in[\![1,n]\!]},(p_i)_{i\in[\![1,n-1]\!]},g,h\right)\in\mathbb{M}\left[\mathbb{V}\left[l\right]\right], \\ c\in\mathbb{L}_{\mathrm{C}},\rho'\in\mathbb{M}\left[c\right]: \\ \exists(a_i)_{i\in[\![1,n]\!]}\in\mathbb{A}^n:\exists(range_i)_{i\in[\![1,n]\!]}\in\mathcal{P}\left(\mathbb{A}\right)^n: \\ \exists v_{n+1}\in\mathbb{E}_{\mathrm{C}}:\exists h'\in\mathbb{H}_{\mathrm{C}}:\exists v'\in{\mathbb{E}_{\mathrm{C}}}^{n+1}: \\ \exists p'\in{\mathbb{L}_{\mathrm{C}}}^{n}: \\ \forall i\in[\![1,n]\!],range_i=[\![a_i,a_i+|x_i|-1]\!], \\ \forall i\in[\![1,n]\!],range_i\subseteq\mathbb{A}\wedge range_i\cap\mathrm{supp}\left(h\right)=\varnothing, \\ \forall i\in[\![1,n]\!],range_i\cap\bigcup\limits_{i\neq j}range_j=\varnothing, \\ \forall i\in[\![1,n]\!],v_{n+1}(x_i)=a_i, \\ \forall i\in[\![1,n]\!],b_i\in\pi_1\left((\!|e_i|\!)\left(\rho\right)\right), \\ h'=h\left[a_i+j\mapsto b_i\left[8j:8j+7\right]\right]_{i=1}^{n}{}_{j=0}^{|x_i|-1}, \\ v'=\left(v_1,\ldots,v_n,v_{n+1}\right),p'=\left(p_1,\ldots,p_{n-1},m\right), \\ \rho'=\left(v',p',g,h'\right),c=\mathrm{at}_P\left(\mathrm{Body}_P\left(f\right)\right) \end{array} \right\}$$

$$\cup\left\{ (l,\rho)\to\omega_o \;\middle|\; \begin{array}{l} \rho=\left((v_i)_{i\in[\![1,n]\!]},(p_i)_{i\in[\![1,n-1]\!]},g,h\right)\in\mathbb{M}\left[\mathbb{V}\left[l\right]\right]: \\ \forall(a_i)_{i\in[\![1,n]\!]}\in\mathbb{A}^n,\forall(range_i)_{i\in[\![1,n]\!]}\in\mathcal{P}\left(\mathbb{A}\right)^n, \\ \forall i\in[\![1,n]\!],range_i=[\![a_i,a_i+|x_i|-1]\!], \\ \exists i\in[\![1,n]\!]: \\ range_i\nsubseteq\mathbb{A}\vee range_i\cap\mathrm{supp}\left(h\right)\neq\varnothing\vee range_i\cap\bigcup\limits_{i\neq j}range_j\neq\varnothing \end{array} \right\}$$

$$\cup\left\{(l,\rho)\to\omega\mid\rho\in\mathbb{M}\left[\mathbb{V}\left[l\right]\right],i\in[\![1,n]\!],\omega\in\pi_2\left((\!|e_i|\!)\left(\rho\right)\right)\right\}\cup\tau_{\mathrm{C}}^{P}\left[\mathrm{Body}_P\left(f\right)\right]$$

where x_i are the names of the parameter of function f.

It is interesting to remark that the functions that are not reachable from the entry point of the program are not translated, and thus generate no transition. It is similar to program linking where functions that are never referenced are not included in the final binary.

4.3.4.2 Initial States

To build the legal initial environments, we have to choose an entry point. The classical choice is to start by executing the function named `main`.

The stack of local environments contains a single empty call frame, the one of the `main` function. The stack of return address is empty. On the other hand, the environment for global variables is already full and the heap is initialized accordingly.

Formally

$$I = \left\{ (l, \rho) \;\middle|\; \begin{array}{l} l \in \mathbb{L}_{\mathrm{C}}, \rho \in \mathbb{E}_{\mathrm{C}} : \\ \exists v \in \mathbb{E}_{\mathrm{C}} : \exists h \in \mathbb{H}_{\mathrm{C}} : \exists g \in \mathbb{E}_{\mathrm{C}} : \exists (a_i)_{i \in [\![1,n]\!]} \in \mathbb{A}^n : \\ \operatorname{supp}(v) = \varnothing, \\ \forall (i,j) \in [\![1,n]\!]^2, i \neq j \Rightarrow \\ \quad [\![a_i, a_i + |t_i| - 1]\!] \cap [\![a_j, a_j + |t_j| - 1]\!] = \varnothing, \\ \operatorname{supp}(h) = \bigcup_{i=1}^{n} [\![a_i, a_i + |t_i| - 1]\!], \\ \operatorname{supp}(g) = \{x_1, \ldots, x_n\}, \\ \forall i \in [\![1,n]\!], g(x_i) = a_i, \\ \rho = ((v), \varepsilon, g, h), \\ l = \mathrm{at}_P(\texttt{main}) \end{array} \right\}$$

where t_i x_i;, with $i \in [\![1,n]\!]$ are the global definitions, where t_i is a type and x_i a variable name.

4.4 Comparison with Real C and Thoughts

Asclepius is clearly inspired by C, but have some differences. Some are significant, others are just a matter of syntactic sugar.

4.4.1 The Place of Undefined, Unspecified and Implementation-Defined Behaviors

There is a lot of cases in which C specifies that the behavior is undefined, that is anything can happen. This is definitely something to avoid. Implementation-defined and unspecified behaviors are less harmful: the standard lets some liberty to the compiler and the environment but it can still specify a set of allowed behaviors. The difference between both notions is that an implementation-defined behavior must be documented by the implementation (typically the compiler), while for an unspecified behavior, the implementation does not have to make a choice. For instance, the number of bits in a byte is implementation-defined, so the compiler must document its choice and stick with it. But the evaluation order of sub-expressions in an expression is unspecified (in most cases), so the compiler can choose any order and does not have to be consistent in any way, neither across several occurrences of identical expressions nor across several evaluations of the same expression. It is still correct for the compiler to specify an unspecified behavior (according to the standard), but we cannot expect it to do so.

These kinds of behaviors are treated in very various ways in Asclepius. First, it is good to note that a lot of such problems in C come from modular programming (like external declaration) or from standard library functions. Since we do not have these features with Asclepius, we do not have to bother about that. Other cases are also out of our scope, like the possible data-race in sub-expression evaluation without sequence point[†], since expressions of the minimal language are pure. Everything related to storage classes or qualifiers is also not applicable. Nevertheless, there is some place left for undefined behavior like dereferencing an illegal pointer or dividing by 0. The default behavior for Asclepius is to transit to an error state. Such situations can be detected dynamically since our abstract language is not subjected to the thousand natural shocks that software engineering is heir to: no architecture constraint, no efficiency problem.... For instance, it is possible to know which addresses are valid by looking which belong to the support of the heap. In practice, this is not possible. Thus, while a real-life dereference might be dangerous, in Asclepius, we can safely decide whether it is safe or not.

From the perspective of unspecified and implementation-defined behaviors, once we have ignored everything that is not applicable, what is left is largely defined in Asclepius. Some strong choices have been made here because of the application: to model interactions of C and assembly, we need a more precise description of some aspects of the language, way beyond the C standard. C is made to be (mostly) platform-independent, and to be efficiently achievable on a large variety of architectures, it needs to leave a lot of details unclear: these are the undefined, unspecified and implementation-defined behaviors. While working on mix of C and assembly, such details are very important, and we perfectly know on which architecture we are working on. We have at the same time both the necessity and the opportunity to define the language much more precisely.

One could argue that such extra hypotheses are not portable and that changing the architecture (or the compiler) may invalid most of the following. This is perfectly true, both conceptually and in practice. Indeed, in OSes, parts that interact with assembly or anything related to the architecture must exist in several versions, for each of supported architectures. The point is to make such parts as small as possible so that most code can be shared using regular platform-independent C or C++ (most of the time). Since we are interested into a specific architecture, we can make these assumptions. Moreover, without them, we would not be able to define a satisfactory semantics since we indeed use the specifics of the architecture.

4.4.2 Desugaring C

C has many features that do not exist in Asclepius, like for-loops, branching with else (and else-if) or switch statements. These features are very handy for the developer but can be easily avoided thanks to a simple desugaring.

C expressions are very rich and their semantics is very complicated. There are two main difficulties: side effects and types. C expressions may include assignments in a lot of places: they are regular expressions; this is not possible with Asclepius where expressions are pure and assignments are statements. C has some very complicated rules

[†]For instance `(*x = *x) | (*y = *y)` is an undefined behavior if `x` and `y` are aliases. From such expression, the compiler may safely assume that `x` and `y` point to disjoint memory blocks.

to deduce the type of expressions (and sub-expressions) based on the types of variables. This is not only a matter of bounds, since the semantics of operators changes depending on the types. For instance, an addition on unsigned integers is computed with modular arithmetic, but an overflow with signed integers is an undefined behavior. Asclepius dissimulates this problem under the chaste veil of n-ary operators in expressions. This set of operators is not fixed, we just need to know their semantics. In particular, we can interpret all implicit transformations in expressions (like integer conversions) in terms of such operators. In the same way, we can have operators for unsigned addition and others for signed addition for each type combination allowed by the C standard (after automatic conversions). Such transformation are performed by most tools working on C semantics, like compilers and analyzers, to clarify the expected behavior.

A missing feature in Asclepius, compared to C, is function pointers. This deserves some comments since we use them intensively with assembly. We can still emulate function pointers by using identifiers instead. So that, rather than writing `p(x)` where `p` is a function pointer, we could write `call(p, x)` where

```c
int call(int id, int x) {
    if(id == 0)
        return f(x);
    if(id == 1)
        return g(x);
    // etc.
}
```

This is a bit long and not modular (since we need to be aware of all existing functions), but it has the expected behavior. Using a bit of sugar, we can make it the general case. In Asclepius' model, functions do not exist in the heap, so that, it is impossible to read or write their code, just as it is forbidden by C. By isolating functions from the memory, we do not let it happens. To add function pointers into Asclepius without compromising this isolation, a strategy is to have a global environment for functions: it is an injective partial mapping from addresses to functions. It is like having randomized identifiers. Calling a function by its address behaves like the `call` function: the semantics finds the requested function and calls it. Any function pointer that is not exactly the address of a function (or a function with a wrong type) is an immediate error.

4.4.3 A Computability Digression

Asclepius uses a finite memory: the heap is finite and environments are not usable as storage from the program's point of view. Moreover, since parameters are stored in the heap, even the recursion depth of a function with at least one parameter is limited. Only the number of calls of function without parameters are unlimited, since return addresses do not require any space. Such calls correspond to empty environments and can only read and write global variables, and thus have limited storage. Thus, we can decide whether such calls terminate in finite time since the number of configurations of global variables is an upper bound on the number of calls while still terminating.

Overall, due to memory limitation, ASCLEPIUS is not TURING-complete, but in the exact same way a computer with finite memory is not TURING-complete either. The amount of memory is fixed because of the application of this work (32-bit x86 architecture without external storage) and so it would mainly be a burden to make the available memory parametric, but could be done easily: we just need to change the size of addresses and extend the heap appropriately. Such parameters can be adapted to make any program run satisfactorily by providing an arbitrarily big memory: terminating programs will eventually be able to terminate without running out of memory and non-terminating programs can run for an arbitrarily long time (allowing, for instance, a non-terminating machine to generate as many decimal of a real number as wanted).

Beyond the concern of computable functions, we can wonder if the number of states is small enough to reasonably allow methods based on exhaustive state enumeration. In our case, it is clearly not reasonable. Moreover, if any restriction in ASCLEPIUS had made any of such method doable, it would not have been a good idea since it would not adapt well to an architecture with more memory or with external storage.

4.4.4 Non-determinism and Memory Usage

The semantics of ASCLEPIUS is non-deterministic in several ways. The most obvious one, and yet the most standard, is that expressions are non-deterministic. A more problematic point is memory allocation: when a variable is declared, it is placed wherever there is available space in the heap. This is a way a bit extreme to avoid strong assumptions on memory allocation policy; since there are many ways to manage memory, it is hard to accept only reasonable ones. But the good side is that what we will be able to prove with such weak hypotheses still hold with more. Making this few assumptions has two main consequences.

Firstly, since variables can be placed with any chaotic layout, we can fill 1 GiB using 2^{30} 1-byte variables spaced out by 3 bytes. If we try to allocate a 4-byte variable, allocation will fail since there is no hole big enough, though we use only a fourth of the memory. If the memory is tidy, we can put all the 2^{30} 1-byte variables at the beginning and still have an available contiguous block of 3 GiB, so that the next allocation will succeed. The moral of this example is that we should not attach too much importance to memory allocation errors. This may look like a limitation of this model but it is much wider. When we analyze some source code, we may consider that memory allocation functions (like `malloc`) can fail, but we rarely take into account that the declaration of a local variable may lead to a stack overflow and thus a crash of a perfectly legitimate program (from the point of view of C standard). To correctly take into account such kind of errors, we must go beyond C standard and know more precisely the memory management policy (are call frames very tight, or do we allocate 1 MiB of memory, even if we use only one integer?) and the available space. Such checks can only been meaningfully performed at a lower level. That is, for instance, what StackAnalyzer [Abs20] does. Another approach is to impose restrictions strong enough so that there is no memory layout messy enough to make allocations fail. For instance, we can prove that if we use a fraction smaller than $1/n$ of the total memory (where n is the byte-size of the wider type), no allocation can fail according to Asclepius' semantics. Such a hypothesis is very sensible for the following: most variables are global and recursion is forbidden, so memory usage of local variables is bounded. To sum up, we have strong hypotheses that encourage us not to worry about allocation errors, but if we really want to, it would need another analysis based on a more precise model.

```c
int main() {
    int x;
    *((int*)42) = 1337;
    return 0;
}
```

Listing 4.1: Dereferencing an arbitrary integer

The second consequence of allocation at arbitrary addresses is that, perchance, a random integer can be a legal address. Let us consider the example shown on Listing 4.1. If `x` is allocated at address 42, `x` receives the value 1337. In any other situations, an illegal memory access happens and the program terminates in an error state. Thus, we need all executions to be valid, whatever the position of each variable in the heap, to declare the program valid. Some approaches avoid this problem by handling pointer more symbolically: by restraining how a pointer can be built and the valid operations on them (typically sum and difference). With this approach, pointers are characterized by the pointed variable and an offset. The non-deterministic part (the address of the pointed variable) is abstracted away. A major downside is that it does not allow arbitrary computations on pointers, like C does. In C, it is possible to convert integer into pointers and conversely (even if the behavior is implementation-defined) and low-level source codes make extensive use of such conversions and perform complex (arithmetic and bitwise) computations on pointers. To represent faithfully such computations, we need to treat pointers like integers, even if it means to pay the price of additional non-determinism.

Chapter 5

The Assembly Language

INTRODUCING ASCLEPIUS as a model of C is the first part toward the modeling of mixed C and assembly. It is now time to study the assembly language of x86 processors. It is very different from a C-like language: it has no structures, there is no arbitrarily complex expressions and no compound statements. Most instructions are quite simple, but they are many of them. Moreover, the notion of undefined behavior is very different: it is more similar to unspecified behavior of C. Implementation-defined behavior is not a relevant notion since we are already at the lower level[*]. For all these differences, our approach of x86 is not the same as for C. We do not need a more precise model, but we are interested only in a fragment of the language since there are too many instructions to treat them all.

An interesting, and almost exhaustive, work on the semantics of x86 assembly can be found in [Das+19]. Unfortunately, it is not very useful here. First, it focuses only on user-level instructions and, while we still need to handle some of them, we are mostly interested into system-level instructions. Though a formalization could have been useful to solve ambiguities of [Int20][†], we mostly needed it about instructions out of the scope of [Das+19]. Moreover, it does not take mixed C and assembly into account, which is our main problem. Finally, in [Das+19], everything is made in the $\mathbb{K}$ framework, which is not at all our context.

We give two semantics for assembly. The first one is very low-level: it sees programs as sequences of bytes in memory. This is a very permissive semantics: we can write everywhere, including in the code, and run data. Actually, there is no formal distinction

[*]There are in fact some degrees of liberty in the x86 architecture left to the implementation (i.e. the actual hardware). This implementation-defined behavior are quite rare, and should rarely be considered as C-style implementation-defined behavior, and are rather the result of history. First, we cannot expect the manufacturer to document it, or the processor to stick with a single behavior. Moreover, unlike a compiler that is useful only when the developer compiles the program (only once in a known environment), the processor is used at each run, and is outside the scope of the developer since it is on the user's computer. Thus, except for very specific cases, one should avoid these implementation-defined behaviors.

[†]Indeed, the semantics is often specified using natural language and might not be univocal. Even in pseudo-code, some elementary operations are written in natural language, especially those that are too complex to be written formally. Unfortunately, they are often also the most ambiguous statements.

between code and data. This semantics is interesting as it is very close to the processor's behavior. Moreover, it is quite simple and some parts are common with the second semantics. It is simpler to define the semantics of some instructions according to the first semantics, and then to make the few adaptations needed for the second one.

The second semantics models code as a sequence of statements that are in memory, but with an unknown layout. This part of memory is not writable, so that the sequence of statements is immutable. However, data (the writable part of memory) is still executable, since this is something that exists in our study case.

5.1 Structure of x86 Assembly Language

As far as we are concerned, assembly code contains instructions, labels and pseudo-instructions (or directives). Instructions are translated by the assembler but the binary encoding is almost equivalent. Labels are used to mark points in the source code. They are removed at assemble-time and replaced by their respective address (or by the offset, if applicable). Pseudo-instructions are instructions executed by the assembler. They allow manually writing bytes in the binary, and they provide a preprocessing mechanism similar to C's preprocessor. We will only use directives that place bytes manually where these directives are written.

Instructions are identified by a keyword (called mnemonic[‡]), followed by comma-separated arguments. Once assembled, the instruction is identified by a binary sequence of 1 or 2 bytes. Different mnemonics can be aliases for the same opcode. Depending on the opcode, there can be 0 to 2 arguments. Labels are identifiers followed by a colon. Directives are the responsibility of the assembler, the syntax is more fluctuating. They may look like instructions whose opcode is not an x86 instruction, or can be prefixed with a single dot. Examples are given on Listing 5.1. Comments begin with a semicolon and stop at the end of the line. Keywords are not case-sensitive.

Instructions may be ambiguous (or at least, hard to fully determine) in various way: size of the assembled instruction, operand size, ambiguous mnemonics.... Such ambiguities exist only in assembly language, but not in machine language (binary encoding): they are solved by the assembler. Consequently, we assume for now that instructions are fully explicit, even if it means that we add some annotations. The resolution of ambiguity will be treated later in chapter 9 "AMICAL" (page 121).

They are three kinds of operands: register, memory and immediate. Register operands designate a register by its name. Memory operands are dereference, optionally with an explicit segment selection. Immediate operands are constant values, corresponding to literal constants of most languages. In practice, some operands may be written as a name, e.g. immediate operands designating an address may be expressed as a label or a memory operand designating a global variable may be expressed as the variable's name.

Memory operands, when not expressed as a variable name, allow some level of complexity in addresses. Allowed address forms are shown on Figure 5.1. The displacement

[‡]From Ancient Greek "μνημονικός" (of memory), since mnemonics are easier to recall than binary encoding.

```
MOV EAX, 5              ; A binary instruction
a_label: push ebp       ; A labeled unary instruction
call a_label            ; An unary instruction
pushad                  ; A zeroary instruction
.byte 0x01, 0x8b        ; -\
.4byte 0x00004242       ; -+-> Two directives.
                        ; They define the sequence 01 8b 42 42 00 00
                        ; which corresponds to the instruction
                        ; add DWORD PTR [EBX + 0x4242], ECX
                        ; which adds the value of ECX into the
                        ; 32-bit variable at address EBX + 0x4242
```

Listing 5.1: Example of assembly language elements

$$\underbrace{\begin{bmatrix}\text{EAX}\\ \text{EBX}\\ \text{ECX}\\ \text{EDX}\\ \text{ESP}\\ \text{EBP}\\ \text{ESI}\\ \text{EDI}\end{bmatrix}}_{\text{Base}} + \left[\underbrace{\begin{pmatrix}\text{EAX}\\ \text{EBX}\\ \text{ECX}\\ \text{EDX}\\ \text{EBP}\\ \text{ESI}\\ \text{EDI}\end{pmatrix}}_{\text{Index}} \times \underbrace{\begin{pmatrix}1\\ 2\\ 4\\ 8\end{pmatrix}}_{\text{Scale}}\right] + \underbrace{\begin{bmatrix}\text{None}\\ \text{8-bit}\\ \text{16-bit}\\ \text{32-bit}\end{bmatrix}}_{\text{Displacement}}$$

Figure 5.1: Allowed memory addresses

is a literal offset to add. For instance, in the address `EDX + EDI * 4 + 0x0ff537`, `EDX` is the base, `EDI` is the index, 4 is the scale and 0ff537H is the displacement. In this example, the displacement is 20-bit large and thus, will only fit in a 32-bit displacement. All of these fields are optional and, in most cases, they are not all used at the same time. The scale provides a nice way to iterate over arrays with 2-, 4- or 8-bytes cells. If the scale is 1, it can be implicit. The base have a special role since it is the one used to choose the segment in which the memory operand should be translated, except when it is explicitly specified.

5.2 Memory Model for Assembly

In assembly, the memory is view through segments and paging (see section 2.3 "Memory Management" (page 17)). We will abstract paging away since it is not always enabled and linear address space is already concrete enough for our concern. We do not need to know how linear address space is mapped over physical memory as long as it behaves like a 4 Gio memory block.

On the other hand we must take segmentation into account, in some extent. Indeed, we have already explained that it is common to have segments that alias each other (see subsubsubsection 2.3.2.2.2 “Segmenting the Linear Address Space” (page 21)). But to know the effect that an assignment has in other segments, we must know precisely how they alias. So, we will only consider the case where all segments start at address 0 and have the same size. This means we should check that this is true when we load a segment register. On the good side, this is a powerful assumption that allows us to almost forget about segments. Segmentation being trivial, working with linear addresses is almost the same as working with physical addresses.

Another touchy point is about memory allocation. It is very different depending on the context. In the system world, at boot, all the memory is its kingdom. Nevertheless, the system must be very careful not to use memory that will be allocated for ordinary programs later. For a user land program, it is the opposite: any memory access that have not been explicitly granted is strictly forbidden. Each piece of memory must be allocated either at program launch (for static and global variables) by the loader, or dynamically (for dynamically allocated blocks) by calling functions such as `malloc`.

In our context, we will avoid allocation questions. We are interested in an embedded OS: it does not perform dynamic allocation for itself, all long-lived structures are global variables. The only dynamic structure is the stack though it has bounded size. We can model it as a static block of memory at the end of which ESP initially points to. It is very similar in Asclepius: the only way to allocate a non-constant amount of memory is thanks to local variables.

In addition to the linear memory, we need to add registers into account. We have to be able to use registers as lvalue while they are not accessible through a pointer. To do that while being as close as possible to the formalism of Asclepius, we enrich the domain of the heap with special values that are not integers, and we forbid to take the address of an lvalue located in such special zone. Moreover, we add special mappings in the global environment for registers.

More formally, let $\mathbb{A}_\mathbb{R}$ be a set of identifiers for each register byte such that $\mathbb{A} \cap \mathbb{A}_\mathbb{R} = \varnothing$. We denote such identifier by the register names, excluding alternate names for shorter parts (e.g. AX, see subsubsection 2.2.1.1 “General-Purpose Registers” (page 10)), with the byte number. That is

$$\mathbb{A}_\mathbb{R} := \left\{ \begin{array}{llllllll} \mathrm{EAX}_0, & \mathrm{EAX}_1, & \mathrm{EAX}_2, & \mathrm{EAX}_3, & \mathrm{EBX}_0, & \mathrm{EBX}_1, & \mathrm{EBX}_2, & \mathrm{EBX}_3 \\ \mathrm{ECX}_0, & \mathrm{ECX}_1, & \mathrm{ECX}_2, & \mathrm{ECX}_3, & \mathrm{EDX}_0, & \mathrm{EDX}_1, & \mathrm{EDX}_2, & \mathrm{EDX}_3 \\ \mathrm{ESI}_0, & \mathrm{ESI}_1, & \mathrm{ESI}_2, & \mathrm{ESI}_3, & \mathrm{EDI}_0, & \mathrm{EDI}_1, & \mathrm{EDI}_2, & \mathrm{EDI}_3 \\ \mathrm{ESP}_0, & \mathrm{ESP}_1, & \mathrm{ESP}_2, & \mathrm{ESP}_3, & \mathrm{EBP}_0, & \mathrm{EBP}_1, & \mathrm{EBP}_2, & \mathrm{EBP}_3 \\ \multicolumn{8}{c}{\ldots} \end{array} \right\}$$

We also need a set of register names $\mathbb{R}$, including alternate names i.e.

$$\mathbb{R} := \left\{ \begin{array}{llll} \text{EAX} & \text{AX} & \text{AL} & \text{AH} \\ \text{ESP} & \text{EBP} & \text{SP} & \text{BP} \\ \text{ESI} & \text{EDI} & \text{SI} & \text{DI} \\ & \multicolumn{2}{c}{\ldots} & \end{array} \right\}$$

We define the set of enriched heaps:

$$\mathbb{H}_{\text{asm}} := \{ f : (\mathbb{A} \uplus \mathbb{A}_{\mathbb{R}}) \rightharpoonup \mathbb{I} \mid \mathbb{A}_{\mathbb{R}} \subseteq \text{supp}\,(f) \}$$

and we add a global constant mapping

$$\begin{aligned} \text{reg} : \mathbb{R} &\to \mathbb{A}_{\mathbb{R}} \\ \text{EAX} &\mapsto \text{EAX}_0 \\ \text{AX} &\mapsto \text{EAX}_0 \\ \text{AL} &\mapsto \text{EAX}_0 \\ \text{AH} &\mapsto \text{EAX}_1 \\ &\vdots \end{aligned}$$

that maps register names to their first byte in $\mathbb{A}_{\mathbb{R}}$, and we extend the function $|\cdot|$ to $\mathbb{R}$ to give as well the byte-size of registers:

$$\begin{aligned} |\cdot| : \mathbb{R} &\to \mathbb{N}^* \\ \text{EAX} &\mapsto 4 \\ \text{AX} &\mapsto 2 \\ \text{AL} &\mapsto 1 \\ \text{AH} &\mapsto 1 \\ &\vdots \end{aligned}$$

On the other hand the environment has the same type as it maps only global variables to their address:

$$\mathbb{E}_{\text{asm}} := \mathbb{E}_{\text{C}} = \mathbb{V} \rightharpoonup \mathbb{A}$$

The memory model is then very close to the one of Asclepius, except that local variables are not relevant here and do not exist in assembly. Thus, we just need an environment for global variables and a heap. Like global variables, the stack has an arbitrary position in the heap at the beginning of the program.

There is the subtle problem of the place of the program in memory. We can indeed take the address of some program point (usually thanks to a label) and use them in control flow instructions. It may be possible to read code as data, depending on code segment permission flags. Moreover, we can write code through the data segment and run it through the code segment. A simple strategy is to forbid it, but even if these manipulations are risky, but they exist, and we must take them into account.

The idea is similar to the allocation of global variables: we choose an arbitrary position in memory for the source code, and we keep it. Thus, we have to represent the program as bytes and to be able to disassemble those bytes to get instructions. For this purpose, we let $\mathrm{Stat}_{\mathrm{asm}}$ the set of assembly statements, and we define the function

$$\mathrm{disassemble} : \mathbb{H}_{\mathrm{asm}} \to (\mathrm{Stat}_{\mathrm{asm}} \times \mathbb{A}) \uplus \Omega$$

that takes a heap (with register), disassembles the instruction at the address in EIP and yields this instruction with the address of the next instruction. We need the address of the next instruction since they have non-constant size in x86. In case of error, an element of Ω is returned instead: ω_a if an attempt to access a non-reserved memory zone is made, and ω_i (illegal instruction) if the byte sequence at the given address is not a legal instruction.

To start correctly the program, we have to make a few assumptions: initially,

– ESP points to the byte after the end of the stack,
– EIP points to the first program instruction.

In addition, we need hypotheses on the state of the system, especially if protection features are used. For instance, if PE (protection enable) flag (bit 0 of CR0) is set:

– GDTR points to a correct GDT,
– CS selects a code segment,
– other segment registers select a data segment.

5.3 The Reduced Instruction Set

The complete instruction set of x86 is huge, even for the relatively old processor we are studying. There are several reasons for that:

– there are a lot of mnemonics providing a very broad set of operations including complex floating-point, binary-coded decimal (BCD) and other complex computations,
– a lot of mnemonics have several variants for various types of arguments,
– the complexity of x86 architecture requires many instructions to control it.

We are not so concerned about the types of arguments as it is mainly an encoding concern, and we are not interested in for now. The multiplicity of instructions is a problem, but there are big categories. There are many instructions that take a lvalue and an operand, perform a binary operation (like the addition or the bitwise and) and write the result in the lvalue. This family includes a lot of arithmetic or logic operations, sometimes unexpected. They are also very close to the `MOV` instruction that copies a value into an lvalue. Another category often distinguished is the set of control flow operations, like jumps, calls and returns.

Technically, control flow instructions might be understood as compound instructions made of `MOV`-like instructions, since a near jump is just an assignment in EIP. The only `MOV` instruction is surprisingly powerful as it is TURING-complete by itself [Dol13; Dom15]. Control flow operations do not have a particular status like they have in most languages. However, they are good didactic examples and thus are worth to be detailed.

On the other hand, the semantics of a lot of complex instructions can be expressed as a composition of simpler transfer functions; they will not be detailed further.

We can thus drastically restrict the instruction set. We will use:

- `MOV`, as it is very fundamental,
- `ADD`, to give an example of binary operation that both reads and writes a lvalue,
- `JMP`, `CALL`, `RET`, to show basic control flow instructions that will be extended when interacting with C code,
- `JZ`, to illustrate conditional control flow,
- `PUSH` and `POP`, as stack operations are very common and good examples.

From there, we trust that we can easily generalize to other instructions.

5.4 A Very Concrete Semantics

In the next chapters we will want to glue semantics of C (approximated by Asclepius) and assembly. For this purpose, it is wise to choose the same formalism, so the semantics of an assembly program is given as a transition system as well (see definition 4.6 "Transition system" (page 49)).

These transition systems have a major difference with the ones for Asclepius programs: the set of states is very different: the memory model is enriched with registers, and we have no label to get the program point. Indeed, it is stored in the EIP register as the address of the next instruction to run.

In this semantics, since we are considering machine code instead of assembly language, there is no name, so no need for an environment.

$$\mathbb{M}^\flat_{\text{asm}} := \mathbb{H}_{\text{asm}}$$

and thus, the set of state simply is

$$Q := \mathbb{M}^\flat_{\text{asm}} \uplus \Omega$$

Another fundamental difference between C and assembly is that, in assembly the program is in the memory, not external. Thus, transitions between states do not depend on an external current statement since the current statement is in the memory. Consequently, there is not a transition system for each program, but a single, huge transition system between memory states.

This semantics is the most concrete possible. We will define another one later that is slightly more constraint as it rejects some unwanted behaviors.

There are two kinds of lvalues (registers and memory references) and three kinds of operand (registers, memory references and immediate operands).

Definition 5.1 – Dereferencing

Let $h \in \mathbb{H}_{\text{asm}}$.

$$\begin{aligned}\text{deref}_h : \mathcal{P}\left(\mathbb{A} \uplus \mathbb{A}_{\mathbb{R}}\right) \times \mathbb{N}^* &\to \mathcal{P}\left(\mathbb{Z}\right) \times \mathcal{P}\left(\Omega\right)\\ (A, \mathit{size}) &\mapsto \left(\bigcup_{a\in A} \pi_1\left(d\right), \bigcup_{a\in A} \pi_2\left(d\right)\right)\\ &\quad\text{where } d = \text{deref}'_h(a, \mathit{size})\end{aligned}$$

where

$$\text{deref}'_h : \left(\mathbb{A} \uplus \mathbb{A}_{\mathbb{R}}\right) \times \mathbb{N}^* \to \mathcal{P}\left(\mathbb{Z}\right) \times \mathcal{P}\left(\Omega\right)$$

$$(a, \mathit{size}) \mapsto \begin{cases} (\varnothing, \{\omega_a\}) & \text{if } \begin{cases} a \in \mathbb{A} \wedge \\ [\![a, a + \mathit{size} - 1]\!] \nsubseteq \text{supp}\left(h\right) \end{cases} \\ \left(\left\{\sum\limits_{i=0}^{\mathit{size}-1} h(a+i) \cdot 2^{8i}\right\}, \varnothing\right) & \text{if } \begin{cases} a \in \mathbb{A} \wedge \\ [\![a, a + \mathit{size} - 1]\!] \subseteq \text{supp}\left(h\right) \end{cases} \\ (\varnothing, \{\omega_a\}) \text{ (absurd)} & \text{if } \begin{cases} a = R_n \in \mathbb{A}_{\mathbb{R}} \wedge \\ \exists i \in [\![0, \mathit{size} - 1]\!] : R_{n+i} \notin \mathbb{A}_{\mathbb{R}} \end{cases} \\ \left(\left\{\sum\limits_{i=0}^{\mathit{size}-1} R_{n+i} \cdot 2^{8i}\right\}, \varnothing\right) & \text{if } \begin{cases} a = R_n \in \mathbb{A}_{\mathbb{R}} \wedge \\ \forall i \in [\![0, \mathit{size} - 1]\!] : R_{n+i} \in \mathbb{A}_{\mathbb{R}} \end{cases} \end{cases}$$

where ω_a is the error raised by an illegal memory access.

The case marked as impossible is it since a register name is always valid, but this case is specified for exhaustiveness.

It may seem a bit restrictive to return an error in case of an illegal memory access since in reality it will raise an exception that can be handled. Actually, the error can be captured and treated in any way by the semantics of instructions. In practice, we do not want such accesses and it is safe to consider them as errors, but it is interesting to remark that returning an error is not an intrinsic requirement.

Definition 5.2 – Semantics of lvalues and expressions

Semantics of lvalues and expressions are defined by mutual recursion.
The semantics of an lvalue l is

$$\langle\!|\, l \,|\!\rangle : \mathbb{M}^{\flat}_{\text{asm}} \to \mathcal{P}\left(\mathbb{A} \uplus \mathbb{A}_{\mathbb{R}}\right) \times \mathcal{P}\left(\Omega\right)$$

$$h \mapsto \begin{cases} \begin{cases} \left(\begin{array}{l}(b + \mathit{scale} \times i + v) \bmod 2^{32}, \\ O_b \cup O_i \cup O_v\end{array}\right) \\ \text{where } (b, O_b) = (\!|\mathit{base}|\!)\left(h\right) \\ \text{and } (i, O_i) = (\!|\mathit{index}|\!)\left(h\right) \\ \text{and } (v, O_v) = (\!|\mathit{imm}|\!)\left(h\right) \end{cases} & \text{if } l = [\mathit{base}{+}\mathit{scale}{*}\mathit{index}{+}\mathit{imm}] \\ \left(\text{reg}\left(l\right), \varnothing\right) & \text{if } l \in \mathbb{R} \end{cases}$$

and the semantics of an expression e is

$$\begin{aligned}
(\!|e|\!) : \mathbb{M}^\flat_{\text{asm}} &\to \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\Omega) \\
h &\mapsto \begin{cases} \left(e \bmod 2^{8\cdot|e|}, \varnothing\right) & \text{if } e \in \mathbb{Z} \\ \begin{cases} \left(V, O \cup O'\right) \\ \text{where } (V, O') = \text{deref}_h\left(A, |e|\right) \\ \text{where } (A, O) = \langle\!|e|\!\rangle\,(m) \end{cases} & \text{if } e = \langle \mathit{lvalue} \rangle \end{cases}
\end{aligned}$$

Now, let build the transition system. We assume we are given a memory space for the stack called *stack*. It is a set of consecutive cells in memory.

Let $h \in \mathbb{M}^\flat_{\text{asm}}$ a memory state. Let $d = \text{disassemble}\,(h)$. If $d \in \Omega$, then d is the only successor of h.

If $d = (\mathit{stat}, \mathit{new\_eip}) \in \text{Stat}_{\text{asm}} \times \mathbb{A}$, we let $h' = h\,[\text{EIP} \mapsto \mathit{new\_eip}] \in \mathbb{M}^\flat_{\text{asm}}$. We perform a case analysis on *stat* to deduce the set of states we can reach from h.

– *stat* = `MOV l, r`:

$$\begin{aligned}
&\left\{ h'' \in \mathbb{M}^\flat_{\text{asm}} \;\middle|\; \begin{array}{l} \exists v \in \mathbb{Z} : \exists a \in \mathbb{A} : \\ v \in \pi_1\left((\!|r|\!)\,(h)\right), a \in \pi_1\left(\langle\!|l|\!\rangle\,(h)\right) : \\ h'' = h'\left[a + i \mapsto v\,[8i : 8i+7]\right]_{i=0}^{|l|-1} \end{array} \right\} \\
\cup &\left\{ h'' \in \mathbb{M}^\flat_{\text{asm}} \;\middle|\; \begin{array}{l} \exists v \in \mathbb{Z} : \exists R_n \in \mathbb{A}_{\mathbb{R}} : \\ v \in \pi_1\left((\!|r|\!)\,(h)\right), R_n \in \pi_1\left(\langle\!|l|\!\rangle\,(h)\right) \\ h'' = h'\left[R_{n+i} \mapsto v\,[8i : 8i+7]\right]_{i=0}^{|l|-1} \end{array} \right\} \\
\cup &\left\{ \omega \in \Omega \;\middle|\; \omega \in \pi_2\left((\!|r|\!)\,(h')\right) \right\} \\
\cup &\left\{ \omega \in \Omega \;\middle|\; \omega \in \pi_2\left(\langle\!|l|\!\rangle\,(h')\right) \right\}
\end{aligned}$$

The bytes of `r` are copied in `l`.

– $\mathit{stat} = \texttt{ADD l, r}$:

$$
\begin{aligned}
&\left\{ h_2 \in \mathbb{M}^\flat_{\text{asm}} \;\middle|\; \begin{array}{l}
\exists (v_1, v_2, \mathit{eflags}) \in \mathbb{Z}^2 : \exists v \in [\![0, 2^{32}-1]\!] : \\
\exists a \in \mathbb{A} : \exists h_1 \in \mathbb{M}^\flat_{\text{asm}} : \\
v_1 \in \pi_1\left(\llparenthesis l \rrparenthesis (h)\right), v_2 \in \pi_1\left(\llparenthesis r \rrparenthesis (h)\right) \\
a \in \pi_1\left(\llangle l \rrangle (h)\right) \\
v = (v_1 + v_2) \bmod 2^{32} \\
h_1 = h' \left[a + i \mapsto v\,[8i : 8i+7]\right]_{i=0}^{|l|-1} \\
\mathit{eflags} = \pi_1\left(\llparenthesis \text{EFLAGS} \rrparenthesis (h_1)\right) \\
h_2 = h_1 \left[\text{EFLAGS}_i \mapsto \text{update\_eflags}(v_1 + v_2, \mathit{eflags})\,[8i : 8i+7]\right]_{i=0}^{3}
\end{array} \right\} \\
\cup &\left\{ h_2 \in \mathbb{M}^\flat_{\text{asm}} \;\middle|\; \begin{array}{l}
\exists (v_1, v_2, \mathit{eflags}) \in \mathbb{Z}^3 : \exists v \in [\![0, 2^{32}-1]\!] : \\
\exists R_n \in \mathbb{A}_{\mathbb{R}} : \exists h_1 \in \mathbb{M}^\flat_{\text{asm}} : \\
v_1 \in \pi_1\left(\llparenthesis l \rrparenthesis (h)\right), v_2 \in \pi_1\left(\llparenthesis r \rrparenthesis (h)\right) \\
R_n \in \pi_1\left(\llangle l \rrangle (h)\right), \\
v = (v_1 + v_2) \bmod 2^{32} \\
h_1 = h' \left[R_{n+i} \mapsto v\,[8i : 8i+7]\right]_{i=0}^{|l|-1} \\
\mathit{eflags} = \pi_1\left(\llparenthesis \text{EFLAGS} \rrparenthesis (h_1)\right) \\
h_2 = h_1 \left[\text{EFLAGS}_i \mapsto \text{update\_eflags}(v_1 + v_2, \mathit{eflags})\,[8i : 8i+7]\right]_{i=0}^{3}
\end{array} \right\} \\
\cup &\left\{ \omega \in \Omega \;\middle|\; \omega \in \pi_2\left(\llparenthesis r \rrparenthesis (h')\right) \right\} \\
\cup &\left\{ \omega \in \Omega \;\middle|\; \omega \in \pi_2\left(\llparenthesis l \rrparenthesis (h')\right) \right\} \\
\cup &\left\{ \omega \in \Omega \;\middle|\; \omega \in \pi_2\left(\llangle l \rrangle (h')\right) \right\}
\end{aligned}
$$

where

$$
\begin{aligned}
\text{update\_eflags} : \mathbb{Z} \times [\![0, 2^{32}-1]\!] &\to [\![0, 2^{32}-1]\!] \\
(v, e) &\mapsto c + p + a + z + s + o \\
&\qquad + \sum_{i \in \{1,3,5\}} 2^i \cdot e\,[i : i] \\
&\qquad + 2^8 \cdot e\,[8 : 10] + 2^{12} \cdot e\,[12 : 31] \\
&\text{where } c = 2^0 \cdot \left[v < 0 \vee v \geqslant 2^{32}\right] \\
&\quad \text{and } p = 2^1 \cdot \left[\sum_{i=0}^{7} v\,[i : i] \equiv 0\,[8]\right] \\
&\quad \text{and } a = 2^4 \cdot [\mathscr{A}] \\
&\quad \text{and } z = 2^6 \cdot \left[v \equiv 0\left[2^{32}\right]\right] \\
&\quad \text{and } s = 2^7 \cdot v\,[31 : 31] \\
&\quad \text{and } o = 2^{11} \cdot \left[v < -2^{31} \vee v \geqslant 2^{31}\right]
\end{aligned}
$$

where $[P]$ is IVERSON's bracket (see section A.2 “Logic” (page 370)) and $\mathscr{A}$ is the condition for AF flag[§].
Bytes of the result of the addition are written into `l`.

– *stat* = `JMP m` where `m` is a memory operand:

$$\left\{ h'' \in \mathbb{M}^{\flat}_{\mathrm{asm}} \;\middle|\; \begin{array}{l} \exists a \in \mathbb{A} : \\ a \in \pi_1\left(\llparenthesis m \rrparenthesis\left(h'\right)\right) \\ h'' = h'\left[\mathrm{EIP}_i \mapsto \left(a \bmod 2^{32}\right)[8i : 8i+7]\right]_{i=0}^{3} \end{array} \right\} \cup \left\{\omega \in \Omega \;\middle|\; \omega \in \pi_2\left(\llparenthesis m \rrparenthesis\left(h'\right)\right)\right\}$$

EIP is updated to the value contained in `m`.

– *stat* = `JMP o` where `o` is an immediate offset:

$$\left\{ h'' \in \mathbb{M}^{\flat}_{\mathrm{asm}} \;\middle|\; \begin{array}{l} \exists (a, eip) \in \mathbb{Z}^2 : \exists d \in \mathbb{A} : \\ a \in \pi_1\left(\llparenthesis o \rrparenthesis\left(h'\right)\right) \\ eip \in \pi_1\left(\llparenthesis \mathrm{EIP} \rrparenthesis\left(h'\right)\right) \\ d = (a + eip) \bmod 2^{32} \\ h'' = h'\left[\mathrm{EIP}_i \mapsto d\,[8i : 8i+7]\right]_{i=0}^{3} \end{array} \right\} \cup \left\{\omega \in \Omega \;\middle|\; \omega \in \pi_2\left(\llparenthesis o \rrparenthesis\left(h'\right)\right)\right\} \cup \left\{\omega \in \Omega \;\middle|\; \omega \in \pi_2\left(\llparenthesis \mathrm{EIP} \rrparenthesis\left(h'\right)\right)\right\}$$

EIP is incremented by of `o`. It is worth noticing that the EIP we increment is already the address of the next instruction.

[§]It is too difficult to formalize here, especially for a flag we do not use. But such a condition exists, that's all we need. Since we do not use this flag[¶], we can simply exclude it from the model.

[¶]It is relevant only for BCD computations.

– *stat* = `CALL m` where `m` is a memory operand:

$$
\left\{ h_3 \in \mathbb{M}^\flat_{\text{asm}} \;\middle|\; \begin{array}{l}
\exists v \in \mathbb{Z} : \exists a \in \mathbb{A} : \exists (h_1, h_2) \in {\mathbb{M}^\flat_{\text{asm}}}^2 : \exists (esp, eip) \in \mathbb{Z}^2 : \\
v \in \pi_1 \left((\!| m |\!) \left(h' \right) \right) \\
a = v \bmod 2^{32} \\
h_1 = h' \left[\text{EIP}_i \mapsto a \left[8i : 8i + 7 \right] \right]_{i=0}^{3} \\
esp \in \pi_1 \left((\!| \text{ESP} |\!) \left(h' \right) \right) \\
eip \in \pi_1 \left((\!| \text{EIP} |\!) \left(h' \right) \right) \\
\left[\!\left[(esp - 1) \bmod 2^{32}, (esp - 4) \bmod 2^{32} \right]\!\right] \subseteq stack \\
h_2 = h_1 \left[\text{ESP}_i \mapsto \left((esp - 4) \bmod 2^{32} \right) \left[8i : 8i + 7 \right] \right]_{i=0}^{3} \\
h_3 = h_2 \left[esp - i - 1 \mapsto \left(eip \bmod 2^{32} \right) \left[8i : 8i + 7 \right] \right]_{i=0}^{3}
\end{array} \right\}
$$
$$
\cup \left\{ \omega_a \in \Omega \;\middle|\; \begin{array}{l}
\exists esp \in \mathbb{Z} : \\
esp \in \pi_1 \left((\!| \text{ESP} |\!) \left(h' \right) \right) \\
\left[\!\left[(esp - 1) \bmod 2^{32}, (esp - 4) \bmod 2^{32} \right]\!\right] \not\subseteq stack
\end{array} \right\}
$$
$$
\cup \left\{ \omega \in \Omega \;\middle|\; \omega \in \pi_2 \left((\!| m |\!) \left(h' \right) \right) \right\}
$$
$$
\cup \left\{ \omega \in \Omega \;\middle|\; \omega \in \pi_2 \left((\!| \text{ESP} |\!) \left(h' \right) \right) \right\}
$$
$$
\cup \left\{ \omega \in \Omega \;\middle|\; \omega \in \pi_2 \left((\!| \text{EIP} |\!) \left(h' \right) \right) \right\}
$$

It is similar to `JMP m` but EIP is pushed on the stack.

– *stat* = `CALL o` where `o` is an immediate offset:

$$\left\{ h_3 \in \mathbb{M}^\flat_{\text{asm}} \;\middle|\; \begin{array}{l} \exists a \in \mathbb{Z} : \exists a' \in \mathbb{A} : \exists (h_1, h_2) \in {\mathbb{M}^\flat_{\text{asm}}}^2 : \exists (esp, eip) \in \mathbb{Z}^2 : \\ a \in \pi_1 \left((\!| o |\!) \left(h' \right) \right) \\ a' = a \bmod 2^{32} \\ esp \in \pi_1 \left((\!| \text{ESP} |\!) \left(h' \right) \right) \\ eip \in \pi_1 \left((\!| \text{EIP} |\!) \left(h' \right) \right) \\ \left[\!\left[(esp - 1) \bmod 2^{32}, (esp - 4) \bmod 2^{32} \right]\!\right] \subseteq stack \\ d = (a' + eip) \bmod 2^{32} \\ h_1 = h' \left[\text{EIP}_i \mapsto d\,[8i : 8i+7] \right]_{i=0}^{3} \\ h_2 = h_1 \left[\text{ESP}_i \mapsto ((esp - 4) \bmod 2^{32})\,[8i : 8i+7] \right]_{i=0}^{3} \\ h_3 = h_2 \left[esp + i \mapsto (eip \bmod 2^{32})\,[8i : 8i+7] \right]_{i=0}^{3} \end{array} \right\}$$

$$\cup \left\{ \omega_a \in \Omega \;\middle|\; \begin{array}{l} \exists esp \in \mathbb{Z}^2 : \\ esp \in \pi_1 \left((\!| \text{ESP} |\!) \left(h' \right) \right) \\ \left[\!\left[(esp - 1) \bmod 2^{32}, (esp - 4) \bmod 2^{32} \right]\!\right] \not\subseteq stack \end{array} \right\}$$

$$\cup \left\{ \omega \in \Omega \,\middle|\, \omega \in \pi_2 \left((\!| o |\!) \left(h' \right) \right) \right\}$$

$$\cup \left\{ \omega \in \Omega \,\middle|\, \omega \in \pi_2 \left((\!| \text{ESP} |\!) \left(h' \right) \right) \right\}$$

$$\cup \left\{ \omega \in \Omega \,\middle|\, \omega \in \pi_2 \left((\!| \text{EIP} |\!) \left(h' \right) \right) \right\}$$

It is similar to `JMP o` but EIP is pushed on the stack.

– *stat* = `RET`:

$$\left\{ h_2 \in \mathbb{M}^\flat_{\text{asm}} \;\middle|\; \begin{array}{l} \exists v \in \mathbb{Z} : \exists (esp, eip) \in \mathbb{Z}^2 : \exists h_1 \in \mathbb{M}^\flat_{\text{asm}} : \\ v \in \pi_1 \left((\!| \texttt{[ESP]} |\!) \left(h \right) \right) \\ esp \in \pi_1 \left((\!| \text{ESP} |\!) \left(h' \right) \right) \\ eip \in \pi_1 \left((\!| \text{EIP} |\!) \left(h' \right) \right) \\ \left[\!\left[esp \bmod 2^{32}, (esp + 3) \bmod 2^{32} \right]\!\right] \subseteq stack \\ h_1 = h' \left[\text{EIP}_i \mapsto \left(v \bmod 2^{32} \right) [8i : 8i+7] \right]_{i=0}^{3} \\ h_2 = h_1 \left[\text{ESP}_i \mapsto ((esp + 4) \bmod 2^{32})\,[8i : 8i+7] \right]_{i=0}^{3} \end{array} \right\}$$

$$\cup \left\{ \omega_a \in \Omega \;\middle|\; \begin{array}{l} \exists esp \in \mathbb{Z} : \\ esp \in \pi_1 \left((\!| \text{ESP} |\!) \left(h' \right) \right) \\ \left[\!\left[esp \bmod 2^{32}, (esp + 3) \bmod 2^{32} \right]\!\right] \not\subseteq stack \end{array} \right\}$$

$$\cup \left\{ \omega \in \Omega \,\middle|\, \omega \in \pi_2 \left((\!| \texttt{[ESP]} |\!) \left(h \right) \right) \right\}$$

$$\cup \left\{ \omega \in \Omega \,\middle|\, \omega \in \pi_2 \left((\!| \text{ESP} |\!) \left(h' \right) \right) \right\}$$

$$\cup \left\{ \omega \in \Omega \,\middle|\, \omega \in \pi_2 \left((\!| \text{EIP} |\!) \left(h' \right) \right) \right\}$$

This instruction pops the top of the stack and stores it in EIP.

– *stat* = `JZ o` where `o` is an immediate offset:

$$\left\{ h'' \in \mathbb{M}^\flat_{\text{asm}} \;\middle|\; \begin{array}{l} \exists (a, eip) \in \mathbb{Z}^2 : \exists d \in \mathbb{A} \\ h'(\text{EFLAGS}_0)\,[6:6] = 1 \\ a \in \pi_1\left(\llparenthesis o \rrparenthesis\,(h')\right) \\ eip \in \pi_1\left(\llparenthesis \text{EIP} \rrparenthesis\,(h')\right) \\ d = (a + eip) \bmod 2^{32} \\ h'' = h'\,[\text{EIP}_i \mapsto d\,[8i : 8i+7]]_{i=0}^{3} \end{array} \right\}$$
$$\cup \left\{ h' \in \mathbb{M}^\flat_{\text{asm}} \;\middle|\; h'(\text{EFLAGS}_0)\,[6:6] = 0 \right\}$$
$$\cup \left\{ \omega \in \Omega \;\middle|\; \omega \in \pi_2\left(\llparenthesis o \rrparenthesis\,(h')\right) \right\}$$
$$\cup \left\{ \omega \in \Omega \;\middle|\; \omega \in \pi_2\left(\llparenthesis \text{EIP} \rrparenthesis\,(h')\right) \right\}$$

This instruction performs a relative jump if ZF flag is set. Otherwise, it does nothing.

– *stat* = `PUSH r`:

$$\left\{ h_2 \in \mathbb{M}^\flat_{\text{asm}} \;\middle|\; \begin{array}{l} \exists (v, esp) \in \mathbb{Z}^2 : \exists h_1 \in \mathbb{M}^\flat_{\text{asm}} : \\ v \in \pi_1\left(\llparenthesis m \rrparenthesis\,(h')\right) \\ esp \in \pi_1\left(\llparenthesis \text{ESP} \rrparenthesis\,(h')\right) \\ \left[\!\left[(esp - 1) \bmod 2^{32}, (esp - |r|) \bmod 2^{32} \right]\!\right] \subseteq stack \\ h_1 = h'\left[\text{ESP} \mapsto ((esp - |r|) \bmod 2^{32})\,[8i : 8i+7]\right]_{i=0}^{3} \\ h_2 = h_1\left[esp + 8i \mapsto (v \bmod 2^{32})\,[8i : 8i+7]\right]_{i=0}^{3} \end{array} \right\}$$
$$\cup \left\{ \omega_a \in \Omega \;\middle|\; \begin{array}{l} \exists esp \in \mathbb{Z} : \\ esp \in \pi_1\left(\llparenthesis \text{ESP} \rrparenthesis\,(h')\right) \\ \left[\!\left[(esp - 1) \bmod 2^{32}, (esp - |r|) \bmod 2^{32} \right]\!\right] \not\subseteq stack \end{array} \right\}$$
$$\cup \left\{ \omega \in \Omega \;\middle|\; \omega \in \pi_2\left(\llparenthesis m \rrparenthesis\,(h')\right) \right\}$$
$$\cup \left\{ \omega \in \Omega \;\middle|\; \omega \in \pi_2\left(\llparenthesis \text{ESP} \rrparenthesis\,(h')\right) \right\}$$

It pushes the value of `r` on the stack.

– *stat* = `POP l`: It pops the top of the stack and stores it `l`.

$$
\begin{aligned}
&\left\{ h_2 \in \mathbb{M}^\flat_{\text{asm}} \,\middle|\, \begin{array}{l} \exists (v, esp) \in \mathbb{Z}^2 : \exists a \in \mathbb{A} : \exists h_1 \in \mathbb{M}^\flat_{\text{asm}} : \\ v \in \pi_1 \left((\!(\,[\mathtt{ESP}]\,)\!) (h) \right), a \in \pi_1 \left(\langle\!| l |\!\rangle (h) \right) \\ esp \in \pi_1 \left((\!(\mathrm{ESP})\!) (h') \right) \\ \left[\!\left[esp \bmod 2^{32}, (esp + |l| - 1) \bmod 2^{32} \right]\!\right] \subseteq stack \\ h_1 = h' \left[a + i \mapsto \left(v \bmod 2^{32} \right) [8i : 8i + 7] \right]_{i=0}^{|l|-1} \\ h_2 = h_1 \left[\mathrm{ESP}_i \mapsto \left((esp + |l|) \bmod 2^{32} \right) [8i : 8i + 7] \right]_{i=0}^{3} \end{array} \right\} \\
\cup &\left\{ h_2 \in \mathbb{M}^\flat_{\text{asm}} \,\middle|\, \begin{array}{l} \exists (v, esp) \in \mathbb{Z}^2 : \exists R_n \in \mathbb{A}_{\mathbb{R}} : \exists h_1 \in \mathbb{M}^\flat_{\text{asm}} : \\ v \in \pi_1 \left((\!(\,[\mathtt{ESP}]\,)\!) (h) \right), R_n \in \pi_1 \left(\langle\!| l |\!\rangle (h) \right) \\ esp \in \pi_1 \left((\!(\mathrm{ESP})\!) (h') \right) \\ \left[\!\left[esp \bmod 2^{32}, (esp + |l| - 1) \bmod 2^{32} \right]\!\right] \subseteq stack \\ h_1 = h' \left[R_{n+i} \mapsto \left(v \bmod 2^{32} \right) [8i : 8i + 7] \right]_{i=0}^{|l|-1} \\ h_2 = h_1 \left[\mathrm{ESP}_i \mapsto \left((esp + |l|) \bmod 2^{32} \right) [8i : 8i + 7] \right]_{i=0}^{3} \end{array} \right\} \\
\cup &\left\{ \omega_a \in \Omega \,\middle|\, \begin{array}{l} \exists esp \in \mathbb{Z} : \\ esp \in \pi_1 \left((\!(\mathrm{ESP})\!) (h') \right) \\ \left[\!\left[esp \bmod 2^{32}, (esp + |l| - 1) \bmod 2^{32} \right]\!\right] \not\subseteq stack \end{array} \right\} \\
\cup &\left\{ \omega \in \Omega \mid \omega \in \pi_2 \left((\!(\,[\mathtt{ESP}]\,)\!) (h) \right) \right\} \\
\cup &\left\{ \omega \in \Omega \,\middle|\, \omega \in \pi_2 \left((\!(\mathrm{ESP})\!) (h') \right) \right\} \\
\cup &\left\{ \omega \in \Omega \,\middle|\, \omega \in \pi_2 \left(\langle\!| l |\!\rangle (h') \right) \right\}
\end{aligned}
$$

This is the transition relation. We need the initial states. Informally, they are the states were global variables, stack and code are disjoint. ESP points at the end of the stack, and EIP points in the code.

5.5 A More Symbolic Semantics

The previous semantics is almost what the processor does. It is interesting because it is quite simple and it helps to describe what we can expect from a higher-level point of view: assembly language source code. Though assembly and machine code are very close they have slightly different constraints that we have to take correctly into account.

We have global variables designated by their names and program points are pointed by labels. Moreover, we do not have access to the machine code, but only the assembly source code. Since the assembling is not unique and a single assembly instruction may have machine encoding of different length, this has some consequences on control flow operations: the destination of a jump whose destination is dynamic is not precisely known. Moreover, if a jump leads to the middle of an instruction, we consider it as an error. Indeed, such jumps are dubious, for the least, and since we do not know the machine code, we do not know what instructions we would get by reading them with some offset**.

At this level, a program is a sequence of statements. Since it has a flat structure, we can use integers to specify a program point: the index of the statement following the program point. There is no need for complicated labels like in ASCLEPIUS. Moreover, labels (in the assembly meaning) are mere aliases for program points. We need to assume that the code lies in memory so instructions have conventional addresses, but we do not want the code to be writable, since the assembly statements are seen as immutable. Nevertheless, we cannot completely get rid of dynamic code, so we need to allow data to be run as code. It is not so often useful, but since it happens in the study case, we need to model it.

```
mov EAX, label          ; EAX <- label
mov EBX, label          ; EBX <- label
and EAX, 0xffff         ; EAX stores the 16 lower bits of label
and EBX, 0xffff0000     ; EBX stores the 16 higher bits of label
or EAX, EBX             ; EAX == label
jmp EAX                 ; jump to label
; ...
label:
```

Listing 5.2: Computing on labels

Executing assembly source code is like executing a single sequence. Running dynamic code (byte sequence in a variable, typically an array) is the same as previously: we disassemble bytes to get corresponding instructions. However, the world of assembly source code is not entirely separated from the data world: each program point correspond to an address in memory. It is similar to the variables in ASCLEPIUS; they can be accessed through their names or by a pointer which may be the result of complicated computations. For instance, Listing 5.2 shows an example where we perform computations on labels but still get a valid destination anyway.

It is not always straightforward to transform an address to a program point for two reasons: the position of the source code is not statically know, and a statement may be assembled in multiple byte sequences with unequal lengths. For instance, the statement `add EAX, 3` can be assembled in three ways:

– `0x05 0x03 0x00 0x00 0x00`: using the opcode that adds a 32-bit immediate operand to EAX;
– `0x81 0xc0 0x03 0x00 0x00 0x00`: using the opcode that adds a 32-bit immediate operand to a register or memory operand;

**In fact, we can find all possible byte sequences and read them starting with some offset. But we would get a titanic number of possibilities, involving very different kinds of instructions, so it is useless to try to handle such behavior that is buggy in legitimate code. Nevertheless, this method is used on purpose in security attacks.

  – `0x83 0xc0 0x03`: using the opcode that adds a 8-bit immediate operand to a register or memory operand.

These sequences have respectively size 5, 6 and 3. Thus, the code in Listing 5.3 is valid only if the assembler choose the 3-bit assembling so, in general, this code is invalid since we cannot guarantee the behavior of the assembler. In fact, most assemblers claim to choose the shortest assembling possible, but we have tested several common assemblers, and they all failed at least once to do so. There are some good reasons to choose another sequence, but the point is that we cannot rely on this assumption. This is slightly more developed in section 9.2 "Assembling" (page 125).

```
jmp 3
add EAX, 3
```

Listing 5.3: A dubious jump

To take this difficulty into account, at the beginning, we choose the length of each statement and the position of the code, just like we choose the position of the stack.

We will not give explicit formal rules like before because it does not make the semantics more understandable. Nevertheless, it is important to detail the type of the set of states of the transition system.

States must keep the program point. This is stored indirectly: the current code pointer is stored in EIP. Moreover, in the initial state, we choose the size of each instruction and the position of the code in memory. This allows deciding whether the code pointer is a valid program point and, in this case, to get it. So, in addition of the heap, we need the base address of the source code and a map of type

$$\text{statement\_size} : [\![0, n-1]\!] \to \mathbb{N}^*$$

where n is the number of statements, such that statement_size (i) is the size of the i^{th} statement. The base address should be such that the last byte of the last statement is still in $\mathbb{A}$. Function statement_size belongs to the state since it is not intrinsic to the program (since there are several possible mapping), but it is constant along an execution trace. This map is chosen in the initial states.

Overall, the memory model for this adapted semantics is

$$\mathbb{M}_{\text{asm}} := \mathbb{H}_{\text{asm}} \times \mathbb{A} \times \mathbb{N}^{*[\![0,n-1]\!]}$$

The first component is the content of the memory strictly speaking. The second is the address of the first instruction in memory. The third component is the size of each instruction, so that we known the position of each instruction in memory. As said before, only the first component can change along an execution trace, the two others are immutable.

5.6 Comparison of Semantics

The first semantics is very close to the way the processor indeed runs the program. It assumes we have chosen a place in memory for the program (for which we have only the binary encoding) and the stack. This transition system is a bit heavy to write, but conceptually simple. Though it is not suited to the formalization of interactions with C. In C (like most programming language), we abstract the actual layout of memory (we see it only through variables) and the code is considered immutable.

The second semantics is very similar up to slight adaptations that aim to solve these problems. The goal is to make it as close as possible of the version we will have in mixed code, while still making sense in pure assembly code. In this semantics, the code is considered as an immutable sequence of instructions in a known region of memory, but with unknown binary content. There are several implications, among them, we can highlight that

– there is no need to disassemble regular code (only when executing data);
– writing into code is illegal and results into error state ω_a (invalid address).

Other changes include using global variables designated by their name as an alias for a memory location (with non-deterministic position in code), and labels in assembly code.

We do not allow the program to modify its own source code, since it is considered buggy, and we should not be able to do it anyway, thanks to memory protection features. But we cannot only rely on them for two reasons. Firstly, such features must be configured and enabled, and before that, there is no such protection, but we would like not to write the code anyway. Moreover, even if such protections are activated, a program that tries to modify a read-only memory (like its code) will be terminated, which is not a desirable behavior. Protections features are still useful for untrusted programs, so that we can ensure they will not modify its own code. For instance, this is useful to avoid dangerous instructions: if they are not in the executable generated by the compiler and the program is unable to modify its code, they will never appear later[††].

Overall, to build the second semantics, we have applied very light changes, but make this model of assembly much closer to what we will find mixed with C. Remaining problems are only about interaction, like operand sharing and control flow.

[††]A typical example is the Pentium F00F bug: the illegal instruction `lock cmpxchg8b EAX` (encoded as F0 0F C7 C8) freezes some Pentium processors rather than raising an exception [Int99; Col98].

Chapter 6

Non-semantic Aspects of Mixed C and Assembly

We have already examined C and assembly separately, we are now interested into mixing them. Such combination of languages is not straightforward at all and problems appear even at syntactic level. This chapter is dedicated to the description of mixed C and assembly code from a non-semantic point of view. Semantics questions will be considered later.

6.1 Syntax of Assembly Blocks

There is a widespread GNU extension that defines a syntax to add assembly blocks in C source code. Sadly, this is not the syntax used by the study case. In our syntax, assembly blocks are opened by "`asm {`" and closed by "`}`". Such blocks can only appear in C functions, not as top-level declarations*. An example of such code is given in Listing 6.1.

```
int c_function() {
    // C code
    asm {
        ; x86 code
    }
    // C code
}
```

Listing 6.1: Generic example of mixed C and assembly

*In some variation of this syntax, blocks can be top-level items, especially to define functions. In our case assembly functions are always wrapped into a dummy C function. We will show an example in Listing 6.8 of section 6.3 "Sharing Symbols" (page 85). Supporting top-level assembly blocks would imply a minor modification of the parser, and a relatively light update of the following.

GNU-style inline assembly requires specifying more than just the assembly code. There are also fields to list the read expressions, written lvalues and modified registers. There are no such things in this syntax. Operands will be detailed in the next section, but they do not need to be given explicitly to the assembly block. This is quite handy for the user but it has an obvious drawback. Either the compiler must parse assembly blocks to infer which register are modified, which variables may be written etc., or it has to compile just as if every assembly block can read and write everything, limiting greatly the optimization potential.

6.2 Sharing Operands

To allow interaction of C and assembly, they must share variables. Registers are not visible from C, consequently it should go the other way: C variables are used in assembly. The way a variable may be used in assembly varies according to its storage class.

6.2.1 Global Variables

The simplest case is the one of global variables. Referencing a global variable is a memory operand whose content is the variable itself. Indeed, global variables are statically allocated into the heap, so naming the global variable is just like dereferencing the address at which the compiler stored it. It is the same in assembly, and the dereferencing is also implicit. Consequently, in Listing 6.2, naming `x` in the assembly code is like writing `[foo]` where `foo` = `&x`. The assembly block in this example is equivalent to the simple C statement `x = 4;`.

Since x86 statements can have at most one memory operand, the example of Listing 6.3 is illegal since the `MOV` instruction has two memory operands: `[EAX]` (the value pointed by EAX) and `x` (a global variable obtained by dereferencing its static address). When we try to assemble such an instruction, GNU Assembler 2.30 emits the error message “too many memory references for mov”.

```
int x;

void f() {
    asm {
        mov x, 4  ; Stores 4 in x
    }
}
```

Listing 6.2: Using a C global variable in an assembly block

```
int x;

void f() {
    asm {
        mov x, [EAX]   ; Illegal
    }
}
```

Listing 6.3: An illegal example involving too many memory operands

6.2.2 Register Storage Class

In C, variables can be declared with the `register` storage class, thanks to the eponymous keyword. This keyword declares an automatic variable[†], but also hints the compiler to store this variable into a register, mainly for performance purpose. This keyword is deprecated in C++, and even unused since C++17. Indeed, compilers are incredibly efficient nowadays and it is almost always better to let them make this kind of choices. In C, its meaning slightly changed: since registers cannot be reached through dereferencing, declaring a variable with the `register` storage class forbids to take its address (C's unary operator &), which can help alias analysis.

However, in our case, this keyword is important: it is not just a hint, but it enforces that the variable is indeed stored in a register. The point is not a performance issue, but that we can use this variable as a register operand in assembly instructions. In particular, it can be used with opcode that can only take a register as parameter (like `MOV` from or to a control register) and be used in a statement whose other argument is a memory operand. For instance, Listing 6.4 is valid since `x` is a register operand here.

```
void f() {
    register int x;
    asm {
        mov x, [EAX]
    }
}
```

Listing 6.4: Using a C variable with `register` storage class in an assembly block

6.2.3 Static Local Variables

Static local variables are local only from the perspective of scopes (where the name of the variable exists), but from the function where it is defined, a static local variable seems global (static storage duration: the lifetime of a static local variable is unbounded).

[†]That is, a local variable, which is automatically deleted at the end of the block.

Indeed, for the value to be preserved, it is stored on the heap like a global variable, but the compiler takes care to check it is only accessed by its name in the function it is declared in, though it can still be accessed through a pointer.

Thus, using a static local variable in assembly is like using a global variable: it is a memory operand and it is used simply by its name. Listing 6.5 shows such an example.

```
void f() {
    static int x;
    asm {
        mov x, 4  ; x is a memory operand
    }
}
```

Listing 6.5: Using a C static local variable in an assembly block

6.2.4 Regular Local Variables

Regular local variables are much weirder. They are stored in the stack thus they are a memory operand, but their addresses are not static since it depends on the size (and the position) of the stack, which adds a difficulty.

```
void f() {
    int x;
    if(?){
        int y;
        l0:  // C code
        // ...
    }
    else {
        l1:  // C code
        // ...
    }
}
```

Listing 6.6: Non-constant set of living local variables

Let us make an aside on compilation. When a function is called in C, the stack contains the parameters, the return address, and the previous copies of ESP and EBP, all of them according to a layout defined by the calling convention. Later, in section 6.5 “Calling Convention” (page 87), we will see more about calling conventions. On the top of this (so, at lower addresses), there are local variables. Usually (but this is up to the compiler), EBP is used to point the first local variable, while ESP still points to the top of the stack. Thus, all local variables are stored between EBP and ESP. The top

```
void f() {
    int x;
    asm {
        mov x[EBP], 4   ; Stores 4 in x
    }
}
```

Listing 6.7: Assigning a C local variable from assembly

of the stack may vary. For instance in the code on Listing 6.6, at label `l0`, there are two living local variable (`x` and `y`), while at label `l1`, `x` is the only living local variable. In this kind of situation, the compiler can act in various way. It can only allocate `x` at the beginning of the function, and then decrease ESP to allocate `y` on demand. With this strategy, if we go by the "else" branch, no additional space is allocated. This is very frugal way to handle memory, but it requires more stack operation. Another way to manage local variable is to get enough room for `x` and `y` in the first place, then we do not have to change ESP again. However, at label `l1`, we waste a few bytes. If `x` is not used once `y` is initialized, another solution is to use the same address for both. Globally, the point is that ESP may change and it is not reliable, but EBP do not move throughout the execution of the function and local variables are always allocated at the same place, relatively to EBP.

The idea is thus to refer to local variables relatively to EBP. Luckily, in our syntax for assembly blocks, the compiler provides a name for the offset between EBP and a local variable: it is the name of the local variable itself. This is illustrated in Listing 6.7: `x` in assembly block is the offset (i.e. a compile-time constant integer value) between EBP and the address of C variable `x`. Thus, `EBP + x` is the address of the local variable `x`, then `[EBP + x]` is the local variable. It is written in the alternative form `x[EBP]` as `a[b]` is an alias for `[a + b]`. It is sometimes more user-friendly since it mimics the syntax for array access in C.

6.3 Sharing Symbols

We have seen how C and assembly can interact through data, using the same variables. There is another kind of interaction: through the control flow.

Or course, as usual, C functions can call C functions, and assembly statements may jump into assembly code. But there are also mixed possibilities: assembly can call a C function, and conversely, C code can call an assembly label just as if it was a C function.

Assembly code sees all C symbols. The converse is not true: C code does not see assembly labels, the only names defined by assembly. However, there is a trick illustrated on Listing 6.8. The assembly label `f` is not directly visible, but an external function with the appropriate type and the same name is declared (but not defined). This way, the C compiler allows C code to call `f`. Later, during linking, the reference to `f` is recognized

```
extern void f(void);

void f_wrapper(){
    asm {
        f:
        ; Assembly code
    }
}
```

Listing 6.8: An assembly label as a function callable from C

as the assembly label.

The way such pseudo-functions get their arguments and provide their return values will be explained in section 6.5 "Calling Convention" (page 87).

6.4 Assembly-Induced Assumptions

```
int f() {
    int a, b, c;
    // ... C code ...
    asm {
        mov EAX, a[EBP];
        add EAX, b[EBP];
        mod c[EBP], EAX;
    }  // Like c = a + b, but with safe integer overflow
    // ... C code ...
}
```

Listing 6.9: Adding with safe wrap-around

To make C and assembly interact nicely, some characteristics of x86 must be applied to C. An important subject is the representation of values: they must be understood in the same way by C and assembly code. This property requires that endianness and representation of signed integers must be shared. Thus, C code uses little endianness and two's complement representation, like x86 processors.

These assumptions go beyond the C standard, but we should be careful about the implications. Two's complement representation allows nice wrap-around when integer overflow occurs in assembly. However, unsigned integer overflows are still undefined in C. If we want to perform an arithmetic operation with wrap-around on overflow, it can be done in assembly, where it is perfectly defined, just like in Listing 6.9. Otherwise, we need to assume the compiler defines this behavior, which does not imply (and once again, is not implied by) the two's complement representation.

6.5 Calling Convention

```
extern int g(int, int);

int g_wrapper(){ /* This name has no importance as it is not meant
                  * to be directly called as a C function */
    asm {
        g:                  ; The label must match the external
                            ; declaration
        mov EAX, [ESP]      ; Top of the stack is the first
                            ; paramater so EAX == 1
        push EBX            ; Saves EBX
                            ; since it is a callee-saved register
                            ; Here, the stack contains the two
                            ; parameters and the saved EBX
        mov EBX, [ESP+8]    ; EBX == 2
        ; ... x86 code ...
        pop EBX             ; Restores EBX
        mov EAX, 4          ; Returned value
        ret
    }
}

int f() {
    return g(1, 2);  // Returns 4
}
```

Listing 6.10: C-to-assembly mixed function

A calling convention describes how functions get their parameters and how they should return their results. This is crucial to allow modular programming: functions defined in different files can call each other as long as they agree on the calling convention. In the same way, it allows code to be shared more globally, through libraries. The calling convention is thus a part of the application binary interface (ABI).

C developers should not really care about calling convention: they work at a higher level of abstraction, and the compiler is in charge of implementing the calling convention. But the calling convention is visible in assembly and to make assembly and C to work together, we have to manually implement it on the assembly side of calls.

There are many conventions. They depend, in particular, on features of the processor and the environment (e.g. the operating system). For instance, on 32-bit x86 processors,

```
int g(int a, int b){
    // Here a == 1 and b == 2
    // ... C code ...
    return 4;
}

int f() {
  int x;
    asm {
        push ECX         ; We might want to save it,
                         ; since it is a caller-saved register,
                         ; even if we do not use it here.
        push 2           ; Second parameter
        push 1           ; First paramter
        call g           ; Call C function
        mov x[EBP], EAX  ; Here EAX == 4
        add ESP, 8       ; Stack clean-up
        pop ECX
    }
    return x;  // Returns 4
}
```

Listing 6.11: Assembly-to-C mixed call

since there are few registers, parameters are usually given on the stack. On processors with more registers (like 64-bit x86 or other architectures), for performance reasons, we can pass parameters in registers. However, even for the given processor, Unix-like and Windows environments use different conventions. For instance, for 64-bit x86 processors, their respective conventions differ on the registers to use. There are usually no objective reasons to favor some registers over others, this is just a matter of history.

The calling convention we are interested in is called cdecl (for “C declaration”). According to this convention, parameters are passed on the stack in right-to-left order, this means that parameters on the right (with respect to the C prototype) must be pushed first, and the leftmost one is the last pushed, and thus is the closest to the top. After the call, parameters are still on the stack and it is the responsibility of the caller to remove them. We say that this is a caller clean-up convention. It is very convenient for variadic functions: the callee does not have to be aware of the number of arguments, and their respective sizes, to infer the number of bytes to clean.

For small non-float types i.e. types that fit in 32 bits (such as integers or pointers), the returned value is stored in register EAX. We will not consider other cases where the returned type is a floating-point type or a type bigger than 32 bits, like aggregate types (array or structures).

Since EAX is used for the returned value, its content is obviously not preserved. But it is not the only one: EAX, ECX and EDX are caller-saved. This means that the called function (the callee) can use them freely and let them in any state at the end[‡]. On the other hand all other registers are callee-saved: the called function must let the registers in the same state as how they initially were (at the beginning of the call). So, the callee has two options: not to use them or to save them (usually on the stack) and restore them at the end.

To respect the calling convention, when an assembly function is called from C, we must check that the assembly callee preserves the value of all other registers. Conversely, when a C function is called from assembly, the value of caller-saved registers are unknown when the callee returns.

Examples of mixed calls are given on Listing 6.10 and Listing 6.11. An interesting fact shown by Listing 6.11 is that `g` cannot see ECX on the stack since it is too deep: `g` has 2 parameters, therefore everything deeper is hidden.

Both these examples show consistent returns: in Listing 6.10, the callee is an assembly function and it returns with the assembly instruction `RET` while in Listing 6.11, the callee is a C function that returns using a C return statement. It is absolutely required to do so: a C callee cannot return through an assembly `RET` instruction, and an assembly callee cannot return using a C return statement. Indeed, C functions have preludes and postludes that handle call frame management. If we call a C function, its prelude is run, some space on the stack is allocated, and ESP and EBP are updated accordingly. The postlude cleans up what the prelude did. But if we simply run a `RET` instruction after the prelude but without executing the postlude, it will not behave as wanted since at this point, the top of the stack is not the return address. Conversely, if we run the postlude in a context where the prelude has not been run, it will free some space that has not been allocated, and it will probably destroy the call frame of the caller. Even if C functions must end by a C return statement, there might be some assembly blocks in it, as long as they preserve the stack.

[‡]For security reasons, for instance in cryptography libraries, the callee might be interested in resetting them anyway. It would be regrettable to leak private data through these registers. The same kind of leak may happen through the stack: if it is not reset, the caller may inspect the state of callee's local variables. Unfortunately, compilers rarely provides an easy-to-use and portable solutions.

Chapter 7

Pathological and Didactic Examples

THE SEMANTICS we will define in the next chapter is very general and is quite tricky. Because the formalization may seem austere and not very explanatory, we will begin by giving examples that illustrate strange cases, justify why the semantics is that complicated, and why we do not exclude apparently undesirable behaviors.

One must keep in mind that these examples are not made up, they are all distilled versions of problems that appear in our study case.

7.1 Assembly Returning from Another Function

In this section, we will explain the Listing 7.1. For the sake of simplicity, all functions have type `void(void)`.

We assume that the entry point of this snippet is `f`. C function `f` calls the pseudo-function* `g`. Then, in `g`, there is a jump to the label `h` that is in an assembly block in another C function. The `ret` instruction is contained in this function, and returns from the call.

The interesting point is that, in contrast with pure C, function enters and exits are not well-parenthesized: the last called function is not necessarily the first returning one. Here, we escape a function without a return. We can see this trick as a horizontal transfer of control.

Let us explain what happens in this snippet. When `g` is called, the caller (`f`) saves appropriate caller-saved registers and may initialize the call frame for the callee. Since `g` is an assembly label, nothing hidden is run, especially, no prelude. At this point, the stack is in the same state as how it was let by `f` at `g()` (line 18). The stack contains what `f` has set and the return address on top of it. The instruction `jmp h` just changes the current program point, thus the stack is preserved. The `ret` instruction pops the return address on the top of the stack and jumps to it, without running any postlude.

*That is, an assembly label declared as external C function.

This return address leads to after the call of g and then, the return part of the calling convention is executed: registers are restored, stack is cleaned up and the remaining of the function runs normally.

```
extern void g(void);

void h_wrapper() {
    asm {
        h:
        ret
    }
}

void g_wrapper() {
    asm {
        g:
        jmp h
    }
}

void f() {
    g();
}
```

Listing 7.1: Returning from another function

7.2 Irregular Usages of Assembly RET

In the example of Listing 7.2, we call an assembly label, g, from a C function f, then we call the assembly label h from assembly. At this point, the top of the stack is the address after call h, under which, there is the return address of g(). Then, from h, we hack to return directly to f. The trick is to pop the top of the stack to expose the return address and perform the ret from there, skipping return to g. The surprising point that undermines the larger part of attempts to define the semantics, is that there is not the same number of CALL and RET. It is not the same as the previous case where the control is horizontally transferred since there still were, at least, the same number of CALL and RET instructions.

Another way to strangely use RET is shown on Listing 7.3. Here, the instructions push h and ret act like jmp h. Then, it is similar to the example shown in section 7.1 “Assembly Returning from Another Function” (page 91). The point is that the first RET executed does not return from a previous call. The control flow cannot be inferred only from the kind of instructions.

```
extern void g(void);

void h_wrapper() {
    asm {
        h:
        add ESP, 4
        ret

    }
}

void g_wrapper() {
    asm {
        g:
        call h
    }
}

void f() {
    g();
}
```

Listing 7.2: A deep return

We need to discover dynamically the role of each control flow instructions. This deduction is based on the destinations which can be computed from the mnemonics, arguments and memory states.

```
extern void g(void);

void h_wrapper() {
    asm {
        h:
        ret
    }
}

void g_wrapper() {
    asm {
        g:
        push h
        ret
    }
}

void f() {
    g();
}
```

Listing 7.3: A return to a new function

7.3 Dynamic Jump

With the previous examples, we have shown that the control flow is not as straightforward as one may think. But, at least, it was static: there was only one execution path possible. Even with conditional branching, the control flow graph can be inferred statically, like in C. Even function pointers have a very limited impact on the CFG, since it can be desugared using the transformation explained in subsection 4.4.2 “Desugaring C” (page 59): it adds relatively few edges, at most one from each call by pointer to each function with correct type. But assembly allows much more complicated control flows: destination of calls can be computed dynamically directly as an address, and not just as a selection of a label. Since each dynamic jump could lead anywhere, a directed acyclic graph (DAG) built without semantic analysis should have an edge from each dynamic jump to every other program point. A CFG is far less useful in such case.

The snippet of Listing 7.4 shows such a tricky example of dynamic jump. It is worth some explanations. Once again, the entry point is `f`, which has a parameter `int n`; this parameter will be forwarded indirectly to `h`. The C function `g` is not useful *per se*, but it hosts a big assembly block that contains only a repetition of `call h`. The first and the second are marked respectively with labels `zero` and `one`. To make this labels visible for C code, they are declared as external functions, but they are never called; they are only used in address computations. Under the assumption that all occurrences of `call h` are assembled in the same way (and thus have the same length), `int d` is initialized with the address of the `n`th occurrence.

When the `n`th occurrence of `call h` is called, the address of the next instruction is pushed on the stack (so the address of the $(\mathtt{n}+1)$th occurrence of `call h`) and control is given to `h`. In `h`, we copy in `int d` the return address and, by a similar computation as the one in `f`, we deduce which instruction was executed (in `int n`). Both local variables are static, so they are globally allocated, and they do not require the preamble to be run before exiting the function.

We can do computations involving `n`, they are skipped here, then we pop the stack to get the return address to `f`, and we return. If the stack is not popped before returning, the control will go the next `call h` and so on.

One should not forget this is a simplified version. There is a good (though incredibly complicated) reason not to directly pass `n` as a parameter of `h` in the real code. And, handling the real code is almost equivalent to handle this polished version.

This shows how complicated can be address computations and that we need to handle it with full generality. This also demonstrate a major difference with C and why assembly cannot be easily compiled to C and then handled for free in any tool that works on C. The C language does not have anything similar. Very close features are functions pointers and label as values (which is a GNU extension, so non-standard). But these features are not so flexible: because of the very nature of C, we cannot jump to a destination that have no name since we cannot predict how C statements will be compiled, and thus, the size they will take.

```
extern void h(void);
extern void zero(void);
extern void one(void);

void h_wrapper() {
    static int d;
    static int n;
    asm {
        h:
        mov EAX, [ESP]   ; return address of n-th call
        mov d, EAX       ; so address of (n+1)-th call
    }
    n = (d - (int)one) / ((int)one - (int)zero);
    asm {  ;       ^ not zero, to get the address of the n-th call
        add ESP, 4
        ret
    }
}

void g() {
    asm {
        zero: call h
        one: call h
        call h
        call h
        call h  ; ... and so on
        ; ...
        call h
    }
}

void f(int n) {
    int d = (int)zero + n * ((int)one - (int)zero);
    asm {
        call d[EBP]
    }
}
```

Listing 7.4: A complicated dynamic jump

7.4 Sub-registers

This example shows a simple and very common hack that adapts very poorly in C. The trick is to get a part (in a bitwise meaning) of a value by writing a register and reading a sub-register. For instance, in Listing 7.5, `short m` is assigned with the 16 lower bits of `int n`. There is no complication here.

```
void f(int n) {
    short m;
    asm {
        mov EAX, n[EBP]
        mov m[EBP], AX
    }
}
```

Listing 7.5: Writing EAX and reading AX

```
struct {
  union {
      struct {
          int8 AL;
          int8 AH;
      } al;
      struct {
          int16 AX;
      } ax;
      int32 EAX;
  } eax;
  /* ... same with other registers ... */
} regs;
```

Listing 7.6: A naive C model of registers

The difficulty comes when one may try to adapt it brutally in C. A naive way is shown in Listing 7.6. It models the state of the register as a C structure, in which each register is a union, allowing to select the whole register or just sub-registers. But C does not behave this way. Writing `regs.eax.EAX` and then reading `regs.eax.ax.AX` is illegal: except if types of initial fields are compatible[†], it is not allowed to read a field of a union type that have not been the last written. Indeed, there is no guarantee that the fields are located as we may think, for instance, AX may be located in the middle of EAX. Making the hypotheses required to make this model works may lead to accept

[†]That is, they have the same types, but may have different names.

some illegal behavior in pure C.

It may seem like a mere technical difficulty, but it is another argument that dissuades us to try to reduce assembly into C.

7.5 Peroration

We have seen various difficulties that undermine efforts to understand assembly as a C-like language: we cannot compile assembly into C (even enriched with registers), and we cannot assume assembly has a nice C-like control-flow structure. In particular, we cannot understand `CALL` and `RET` instructions as the C concepts of calls and returns, and we cannot assume calls and returns are well-parenthesized (or even in equal numbers).

Of course, this is not an exhaustive list of problems we found in our study case, but it is a sufficient subset to justify the choices of the following chapter. More importantly, we can handle them using relatively few (and reasonable) assumptions on the compiler and the environment. Without these assumptions, we could do almost nothing.

Our study case contains other issues that are not illustrated by these examples, and not quite handled by the following semantics. It can be easily adapted to handle them, but only using very strong assumptions. Since we want the semantics to stay as portable as possible, we will not add these assumptions from the beginning. Moreover, such code is not in the analyzed fragment of the study case, thus we may ignore these problems for now. Such problems, with solutions, will be explained in section 8.4 “Summary and Extension” (page 117).

Chapter 8

Semantics of Mixed C and Assembly

THIS CHAPTER is dedicated to a formalization of the semantics of mixed C and assembly. In previous chapters, we introduced all the concepts useful here, especially, the semantics of a C-like language, the semantics of assembly and what to expect from the semantics of the mix. The time has come to formalize this last semantics.

This formalization is not so meant to be an explanatory approach of mix C and assembly, but to be the theoretical background for the following. As we want to analyze such code, we will design abstractions and formally establish their soundness. Yet, soundness is always relative to the concrete semantics of the language. This is what we will define here. Abstractions will be explained in Part III "Abstraction of Mixed C and x86 Assembly" (page 173).

8.1 The Formal Language

In chapter 4 "A Minimal C-like Language" (page 43), we defined ASCLEPIUS to be a model of C. We will now enrich it with assembly blocks. We call this enriched version PANACEA [*], for Plump ASCLEPIUS as a Neat Approximation of C Enriched with Assembly. Moreover, in ancient Greek mythology, PANACEA (Πανάκεια) is the daughter of ASCLEPIUS and a goddess of universal remedy. With that name, we express the hope that PANACEA is the final formalization solving all the problems, after a lot of trials and errors.

Definition 8.1 – Statements of PANACEA

We extend the non-terminal symbol $\langle c\_stat \rangle$ of the syntax of ASCLEPIUS (see definition 4.3 "Statements" (page 44)) as follows:

$\langle c\_stat \rangle \quad ::= \quad \ldots$

[*] The note about ASCLEPIUS (see page 43) applies also about PANACEA.

	|	'asm' '{' $\langle asm\_block \rangle$ '}'
$\langle asm\_block \rangle$	::=	{ $\langle asm\_item \rangle$ }
$\langle asm\_item \rangle$	::=	$\langle asm\_statement \rangle$
	|	$\langle asm\_label \rangle$
$\langle asm\_label \rangle$	::=	$\langle identifier \rangle$':'

where

$$\langle asm\_label \rangle \in \mathrm{Stat}_{\mathrm{asm}}$$

Definition 8.2 – Statements and programs

We denote Stat the set of PANACEA statements.
We denote $\mathbb{P}$ the set of PANACEA programs.

We need to extend the labels of ASCLEPIUS to PANACEA. For this purpose, we first need to define components of PANACEA programs.

Definition 8.3 – Components

Given $\pi \in \Pi$, we extend $\mathrm{Cmp}^{\pi} : \mathrm{Stat} \to \mathcal{P}(\mathrm{Stat} \times \Pi)$ of definition 4.5 "Components" (page 46) by

$$\begin{aligned}
\mathrm{Cmp}^{\pi}(\texttt{asm } \{S_1 \dots S_n\}) &:= \{(\texttt{asm } \{S_1 \dots S_n\}, \pi)\} \cup \bigcup_{i=1}^{n} \mathrm{Cmp}^{\pi.i}(S_i) \\
\mathrm{Cmp}^{\pi}(\langle asm\_stat \rangle) &:= \{(\langle asm\_stat \rangle, \pi)\} \\
\mathrm{Cmp}^{\pi}(\langle asm\_label \rangle) &:= \varnothing
\end{aligned}$$

Likewise, we extend

$$\begin{aligned}
\mathrm{Cmp}_P : \mathbb{F}[P] &\to \mathcal{P}(\mathrm{Stat} \times \Pi) \\
f &\mapsto \mathrm{Cmp}^{(f,\varepsilon)}(\mathrm{Body}_P(f))
\end{aligned}$$

and

$$\begin{aligned}
\mathrm{Cmp} : \mathbb{P} &\to \mathcal{P}(\mathrm{Stat} \times \Pi) \\
P &\mapsto \bigcup_{f \in \mathbb{F}[P]} \mathrm{Cmp}_P(f)
\end{aligned}$$

It is interesting to remark that labels are not components since they disappear at compile-time. Indeed, we do not want to put labels around assembly labels: assembly labels mark program points, they are not instructions.

```
asm {
    mov EAX, 1
    a_label:
    mov EAX, 2
}
```

Listing 8.1: Two instructions separated by a label

We can remark that the last part of paths of assembly statements separated by a label are not consecutive integers. For instance, in Listing 8.1 if the assembly block is located at path $\pi \in \Pi$, the instruction `mov EAX, 1` is located at path $\pi.1$, and `mov EAX, 2` is located at path $\pi.3$, but path $\pi.2$ does not exist since labels are omitted from components. This does not matter for the following since we never use paths explicitly, and it makes the formalization much simpler.

In addition to Asclepius labels (elements of $\mathbb{L}_\mathrm{C}$), we need to label assembly statements. Since assembly blocks are only a sequence of statements, we just need labels at the beginning of the block, at the end and between each pair of consecutive statements. We can identify such labels as an ordered pair (l, i) where $l \in \mathbb{L}_\mathrm{C}$ is the label of the assembly block (as an Asclepius component) and $i \in \mathbb{N}$ is the number of the program point in this block (which implies we skip labels). Formally, we define

$$\mathbb{L}_\mathrm{asm} := \mathbb{L}_\mathrm{C} \times \mathbb{N}$$

and

$$\mathbb{L} := \mathbb{L}_\mathrm{C} \uplus \mathbb{L}_\mathrm{asm}$$

Likewise, we extend

$$\begin{aligned} \mathrm{at}_P &: \mathrm{Cmp}\,(P) \to \mathbb{L} \\ \mathrm{after}_P &: \mathrm{Cmp}\,(P) \to \mathbb{L} \\ \mathrm{in}_P &: \mathrm{Cmp}\,(P) \to \mathcal{P}\,(\mathbb{L}) \end{aligned}$$

We need to add a labeling axiom to handle the case of assembly blocks.

Axiom 8.1 – Labeling of programs

We are given a program $P \in \mathbb{P}$. We have the hypotheses of labeling of Asclepius programs (see axiom 4.1 “Labeling of programs” (page 47)).

Moreover, for $C \in \mathrm{Cmp}(P)$, if $C = \left(\texttt{asm \{}(a_i)_{i\in[\![1,m]\!]}\texttt{\}}, \pi\right)$:

$$\begin{aligned}
\mathrm{in}_P(C) &= \{\mathrm{at}_P(C), \mathrm{after}_P(C), (\mathrm{at}_P(C), 0)\} \\
&\cup \bigcup_{i=1}^{m} \mathrm{in}_P((s_i, \varphi(i))) \\
\{\mathrm{at}_P(C), \mathrm{after}_P(C)\} &\subseteq \mathbb{L}_{\mathrm{C}} \\
\forall i \in [\![1,n]\!], \mathrm{in}_P((s_i, \pi.\varphi(i))) &= \{\mathrm{at}_P((s_i, \pi.\varphi(i))), \mathrm{after}_P((s_i, \pi.\varphi(i)))\} \subseteq \mathbb{L}_{\mathrm{asm}} \\
\forall i \in [\![1,n]\!], \mathrm{at}_P((s_i, \pi.\varphi(i))) &= (\mathrm{at}_P(C), i-1) \in \mathbb{L}_{\mathrm{asm}} \\
\forall i \in [\![1,n]\!], \mathrm{after}_P((s_i, \pi.\varphi(i))) &= (\mathrm{at}_P(C), i) \in \mathbb{L}_{\mathrm{asm}}
\end{aligned}$$

where $(s_i)_{i\in[\![1,n]\!]}$ is the extracted sequence of a that keeps only the statements, that is, there exists $n \in [\![1,m]\!]$ and $\varphi : [\![1,n]\!] \to [\![1,m]\!]$ increasing (i.e. φ is an extraction) such that

$$\begin{aligned}
&\forall i \in [\![1,n]\!], s_i = a_{\varphi(i)} \\
&\forall i \in [\![1,m]\!], a_i \in \mathrm{Stat}_{\mathrm{asm}} \Leftrightarrow i \in \mathrm{Im}(\varphi)
\end{aligned}$$

Even if the assembly block contains no instruction (it is empty or contains only assembly labels), there still is a label (in $\mathbb{L}_{\mathrm{asm}}$) inside. It is useful in the case where an assembly block only contains an assembly label: the Panacea label marks the program point that can be the destination of jumps.

Notation 8.1 – Labeled empty assembly blocks

We extend the notation 4.3 "Labeled statement" (page 47) for the case of empty assembly blocks.
Given a program $P \in \mathbb{P}$, and three labels $(k, l, m) \in \mathbb{L}^3$ if

$$\exists \pi \in \Pi : \begin{cases} & \mathrm{at}_P((\texttt{asm \{ \}}, \pi)) &= l \\ \wedge & \mathrm{after}_P((\texttt{asm \{ \}}, \pi)) &= m \\ \wedge & \mathrm{in}_P((\texttt{asm \{ \}}, \pi)) &= \{k, l, m\} \end{cases}$$

then we write

$$^{l}\texttt{asm \{ } ^{k} \texttt{ \}}^{m}$$

This rule is much more simple and constructive than other labeling axioms. If we look at Listing 8.1 "Two instructions separated by a label" (page 101) again, and we call l the label before the assembly block, the program point before the first instruction will be $(l, 0)$. The program point between both instructions is labeled $(l, 1)$ (and is a synonymous for "`a_label`") and the last program point in the block is $(l, 2)$. Labeling of assembly blocks is that simple because they have a flat structure.

```
// Label: $l \in \mathbb{L}_C$
asm {
    ; $(l, 0) \in \mathbb{L}_{asm}$
    mov EAX, 1
    a_label: ; $(l, 1) \in \mathbb{L}_{asm}$
    mov EAX, 2
    ; $(l, 2) \in \mathbb{L}_{asm}$
}
```

Listing 8.2: Explicitly labeled version of Listing 8.1

8.2 Memory Model

We need a memory model that allows both pure C and pure assembly to be run together; it is necessarily a mix of memory models of both languages. There are two approaches: defining an almighty memory model to use in C and assembly, or using a memory model for C program points (where there are no registers) and another for assembly program points, and transiting between both when entering or leaving assembly blocks.

In subsection 4.3.1 “Memory Model” (page 48), we defined a C memory state as a 4-tuple containing a stack of local environments, a stack of return addresses, a global environment and a heap. In section 5.2 “Memory Model for Assembly” (page 65), we defined assembly memory states as an enriched heap, that models both the main memory and registers. In pure assembly, no distinction was made between registers, but in C, we need to distinguish general-purpose registers from others (see subsection 2.2.1 “Registers” (page 10)). Indeed, general-purpose registers are subject to be modified arbitrarily, while system registers (such as segment registers) are not modified by C code. The EFLAGS register is a special case: status flags may be modified but system and control flags are constant. In the first approach (an almighty memory model), we would keep general-purpose registers alive all the time, but assigning them arbitrary values at each step in C code. In the second approach (two specialized memory models), general-purpose registers would exist only in assembly blocks, but absent from the C-specific memory model.

The correspondence between both approaches is quite trivial. Since C statements modify arbitrarily registers and are not accessible, we can forget it. Conversely, in a model without registers at C program points, we can consider they have arbitrary values. However, there is a non-obvious difference that favors a formalization based on a single almighty model: when an assembly block is immediately followed by another one, nothing is run between them, and value of registers are preserved, as illustrated on Listing 8.3. It is exactly as if there was only one block. To simplify this case, we choose to keep the registers at all times.

We thus keep the enriched heap (heap with registers). We also need an environment for global variables (the address of global variables). The remaining component of the memory model in Asclepius makes the call stack: there are local environments and

```
void f(void) {
    asm {
        mov EAX, 42
    }
    asm {
        cmp EAX, 42   ; Will set ZF status flag since EAX == 42 here
    }
}
```

Listing 8.3: Two immediately successive assembly blocks

return labels. We cannot use directly the same kind of stack since an assembly can be called from C, just like a C function, but at the destination point, there is no local environment. On the other hand, the stack is explicit. The solution is to have a stack in memory that is divided into call frames. An assembly call frame is an explicit array, while a C call frame is a black box of unknown content. The only thing we know about C call frames is that they contain the local variables, but with unknown padding, order. . . . The call frame contains also other data, such as the return address, saved registers, etc. These bytes must stay unchanged not to risk corrupting the call frame, unlike local variables that can be safely modified, either modified by name or pointer.

First, we need to define the set of environments that allow variables declared with appropriate keyword to be stored in registers, that is

$$\mathbb{E} := \mathbb{V} \rightharpoonup (\mathbb{A} \uplus \mathbb{A}_{\mathbb{R}})$$

A C call frame is a 4tuple (v, a, s, f) where $v \in \mathbb{E}$ is the local environment, $a \in \mathbb{L}$ is the return address given as a label, $s \in \mathbb{N}$ is the size of the call frame and $f \in \mathbb{F}$ is the function whose prelude built this call frame. The content of an assembly call frame is explicitly defined in memory, and we just need to remember its size. Since call frames are contiguous, and we know the base address of the stack, we can compute the base address of each call frame. Both kind of call frames still miss a component to correctly handle mixed calls. For calls from C to assembly, we need to check that assembly restore the registers as required by the calling convention and touched anything in C call frames, except local variables. For calls from assembly to C, we need to be able to retrieve the original value of registers, as ensured by the calling convention. To perform these tasks, we keep a copy of the heap. Overall, call frames are elements of the set

$$((\mathbb{E} \times \mathbb{L} \times \mathbb{N} \times \mathbb{F}) \uplus \mathbb{N}) \times \mathbb{H}_{\text{asm}}$$

Thus, the set of memory state is

$$\mathbb{M} := \mathbb{E}_{\text{C}} \times \mathbb{H}_{\text{asm}} \times (((\mathbb{E} \times \mathbb{L} \times \mathbb{N} \times \mathbb{F}) \uplus \mathbb{N}) \times \mathbb{H}_{\text{asm}})^{+} \times \mathbb{A} \times \mathbb{N} \times \mathbb{A} \times \mathbb{N} \times \mathbb{A}^{\mathbb{L}}$$

The first component ($\mathbb{E}_{\text{C}}$) is the global environment, the second ($\mathbb{H}_{\text{asm}}$) is the enriched heap. The third component ($(((\mathbb{E} \times \mathbb{L} \times \mathbb{N} \times \mathbb{F}) \uplus \mathbb{N}) \times \mathbb{H}_{\text{asm}})^{\star}$) is the sequence of call stacks with saved registers (in the $\mathbb{H}_{\text{asm}}$ component). The fourth and fifth components ($\mathbb{A}\times\mathbb{N}$) are respectively the base address of the stack and the size (counted downward) of the stack. The sixth and seventh components ($\mathbb{A}\times\mathbb{N}$) are respectively the base address and the size of the code (which is a write-only zone). The eighth component ($\mathbb{A}^{\mathbb{L}}$) is a map that gives the memory address of each program point. These addresses must be contained in the memory block dedicated to the code and be consistent with instruction encoding, i.e. the distance between the addresses of consecutive assembly program points must be the size of a possible encoding for the instruction between these two program points. This question is detailed further in chapter 9 "Amical" (page 121).

We can remark that every C call frame has a return address, including the first one (the call frame of `main` function). This is to simplify this complex enough definition. We shall simply use an arbitrary value since it will never be used. We can also remark that the first call frame is always a C call frame.

8.3 Semantics

We will not present the semantics of Panacea with the same level of formalization as Asclepius. Indeed, it was already tedious and not so clear for Asclepius; for Panacea, it would be pantagruelian. The semantics is given in a more operational way. We always start by giving the intuition of what we shall do in the first paragraphs, before formalizing it. The reader can jump directly to the next title to skip formal details without missing any explanation.

Formally, we define the semantics of Panacea as a transition system whose states are ordered pairs of a label (in $\mathbb{L}$) and a memory state (in $\mathbb{M}$), that is, this is a transition system whose set of states is $\mathbb{L}\times\mathbb{M}$.

We can distinguish cases according to the kind of statement we consider: some of them are easy.

8.3.1 Where it is Legal to Write

Global variables are always accessible and writable. But, in the case of local variables, this is not so straightforward. Local variables are located in the stack, but the stack contains many things: it is split in call frames. Assembly call frames are very explicit: we know what they contain. On the other hand, C call frames are guaranteed to contain local variables, but not only. Modifying local variables of deeper call frames is legal; these variables may be accessed through pointer. But writing in a C call frame, anywhere else is considered an immediate error (ω_a: invalid address).

Lastly, it is illegal to write in the region of memory that contains code. Of course, the code, the stack and global variables cannot alias each other in any way.

We can formalize whether a region is writable. We start by a utility function that gives the size of a call frame.

$$\begin{array}{rcll} \text{cs\_size} : (\mathbb{E} \times \mathbb{L} \times \mathbb{N} \times \mathbb{F}) \uplus \mathbb{N} & & \to & \mathbb{N} \\ (\_, \_, n, \_) & \in \ \mathbb{E} \times \mathbb{L} \times \mathbb{N} \times \mathbb{F} & \mapsto & n \\ n & \in \ \mathbb{N} & \mapsto & n \end{array}$$

Then we can easily check where a region of size s, starting at address a is writable in the memory state m.

$$\begin{aligned} \text{is\_writable} : \mathbb{M} \times \mathbb{A} \times \mathbb{N}^* &\to \{ff, tt\} \\ (m, a, s) &\mapsto \textbf{let } ((c_i, \_))_{i \in [\![1,n]\!]} := \pi_3(m) \textbf{ in} \\ &\qquad \exists v \in \mathbb{V} : [\![a, a+s-1]\!] \subseteq [\![\pi_1(m)(v), \pi_1(m)(v) + |v| - 1]\!] \\ &\qquad \vee\ \exists i \in [\![1,n]\!] : c_i \in \mathbb{N} \wedge 0 \leqslant \pi_4(m) - a - \sum_{j=1}^{i-1} \text{cs\_size}(c_j) < c_i \\ &\qquad \vee\ \exists i \in [\![1,n]\!] : c_i \in \mathbb{E} \times \mathbb{L} \times \mathbb{N} \wedge \textbf{let } (e, \_, \_) := c_i \textbf{ in} \\ &\qquad\qquad \exists v \in \mathbb{V} : [\![a, a+s-1]\!] \subseteq [\![e(v), e(v) + |v| - 1]\!] \end{aligned}$$

It simply checks whether the given region is in a global or local variable, or in an assembly call frame.

This check must be performed when getting the address of an lvalue (that is computing $\langle\!| l |\!\rangle$), when we intend to write it. If the result of is_writable(m, a, s) is ff, the transition gives only ω_a.

8.3.2 Pure C and Pure Assembly Statements

We start by examining the case of pure C and assembly statement. The idea is to transform pure C or assembly transitions to Panacea transitions.

8.3.2.1 Pure Assembly Statements

In the set of assembly statements, this family includes non-control statements or control statements whose destination is in the current assembly block. For these kinds of statements, the transitions are very similar as the one for pure assembly, adapted for the new memory model.

We assume we are given a program $P \in \mathbb{P}$. Let $(m, m') \in \mathbb{M}^2$ be two memory states such that $\forall i \in \{1\} \cup [\![4, 8]\!], \pi_i(m) = \pi_i(m')$. Let $(l, l') \in {\mathbb{L}_{\text{asm}}}^2$ such that there exists an instruction $s \in \text{Stat}_{\text{asm}}$ such that $\exists l'' \in \mathbb{L}_{\text{asm}} : {}^{l}s^{l''}$. Let $a := \pi_8(m)(l)$ and $s := \pi_2(m)(l'') - \pi_2(m)(l)$. Then the transition $(l, m) \to (l', m')$ is valid if and only if:

- the transition $(\pi_2(m), a, (0 \mapsto s)) \to (\pi_2(m), a, (0 \mapsto s))$ is valid in pure assembly for the program containing only the instruction s at address a,
- lvalues pass the is_writable check,
- topmost call frames of m and m' are assembly call frame,
- $(\!|ESP|\!)(m) + s = (\!|ESP|\!)(m') + s'$ where s (resp. s') is the size of the topmost call frame of m (resp. m'); this guarantee the size of the call frame is updated correctly when a stack instruction occurs,

– EIP needs to be updated. Since it will be a common operation, we define a function to do that:

$$\begin{aligned}\text{update\_EIP} : \mathbb{H}_{\text{asm}} \times \mathbb{L} \times \mathbb{A}^{\mathbb{L}} &\to \mathbb{H}_{\text{asm}} \\ (h, l, pp) &\mapsto h\left[\text{EIP}_i \mapsto pp(l)\left[8i : 8i+7\right]\right]_{i=0}^{3}\end{aligned}$$

We shall have

$$\forall i \in \llbracket 0, 3 \rrbracket, \text{update\_EIP}(\pi_2(m), l', \pi_8(m))(\text{EIP}_i) = \pi_2(m')(\text{EIP}_i)$$

We shall be careful about `RET` statements. Indeed, we cannot syntactically check whether the return address is an assembly program point, since it is contained on the stack. If the return address is indeed an assembly address, the previous criterion is valid, otherwise, this is likely a return from a C-to-assembly mixed call, which will be explained later (see subsubsection 8.3.4.1 “Calls from C to Assembly” (page 110)). To know if the given return address ra is a regular return, we shall just test whether there is an assembly program point with this address, that is

$$\exists l \in \mathbb{L}_{\text{asm}} : \pi_8(m)(l) = ra$$

We may also go even further. If the address is between two program points in the same assembly block, the return address is at the middle of an assembly instruction. This means it cannot be a return from a mixed call and the `RET` is not valid. Formally, if

$$\exists (l, l') \in {\mathbb{L}_{\text{asm}}}^2 : \exists c = (s, \pi) \in \text{Cmp}(P) : \begin{cases} \text{at}_P(c) = l \wedge \text{after}_P(c) = l' \wedge s \in \text{Stat}_{\text{asm}} \\ \wedge\, \pi_8(m)(l) < ra < \pi_8(m)(l') \end{cases}$$

then the return address is in the middle of an assembly instruction, and thus, we transit immediately to the error state ω_a.

8.3.2.2 Pure C Statements

It is similar for Asclepius statements, the only non-trivial transitions are at the interface between languages: beginning and end of assembly blocks, and calls to assembly pseudo-functions. All other transitions are very close to the one we defined for Asclepius, but with a precaution: general-purpose registers may be used arbitrarily. Thus, to adapt Asclepius transitions to Panacea, we make them to modify arbitrarily all the general-purpose registers. Indeed, not only registers are unknown in C, but they can change at each step.

Let us formalize that, too. Let $(m, m') \in \mathbb{M}^2$ be two memory states such that $\forall i \in \{1\} \cup \llbracket 4, 8 \rrbracket, \pi_i(m) = \pi_i(m')$. Let $(l, l') \in {\mathbb{L}_{\text{C}}}^2$ such that there exists an instruction $s \in \text{Stat}_{\text{C}}$ such that $\exists l'' \in \mathbb{L}_{\text{C}} : {}^{l}s^{l''}$. Let $h := \pi_2(m)_{|\mathbb{A}}$ and $h' := \pi_2(m')_{|\mathbb{A}}$ be the non-enriched heaps. We assume that $\forall r \in R, \pi_2(m)(r) = \pi_2(m')(r)$ where $R \subseteq \mathbb{A}_{\mathbb{R}}$ is the set of non-general-purpose registers that are guaranteed to be untouched by C, that is R contains segment registers ($(\text{CS}_i)_{i \in \llbracket 0,1 \rrbracket}$, $(\text{DS}_i)_{i \in \llbracket 0,1 \rrbracket}$, …), system registers ($(\text{CR3}_i)_{i \in \llbracket 0,3 \rrbracket}$,

…) and other special registers. Let $((c_i, h_i))_{i \in [\![1,n]\!]} := \pi_3(m)$ and $((c'_i, h'_i))_{i \in [\![1,p]\!]} := \pi_3(m')$, be sequences of call frames without saved heaps. We assume that $n - p \in [\![-1, 1]\!]$ and $\forall i \in [\![0, \min(n,p)]\!], (c_i, h_i) = (c'_i, h'_i)$. Let $((v_i, a_i, \_, \_))_{i \in [\![0,\alpha]\!]}$ the sequence of C call frames in c and $((v'_i, a'_i, \_, f'_i))_{i \in [\![0,\beta]\!]}$ the sequence of C call frames in c'. Then the transition $(l, m) \to (l', m')$ is valid if and only if:

– the transition

$$\left(l, \left((v_i)_{i\in[\![0,\alpha]\!]}, (a_i)_{i\in[\![1,\alpha]\!]}, \pi_1(m), h\right)\right) \to \left(l', \left(({v'}_i)_{i\in[\![0,\alpha]\!]}, ({a'}_i)_{i\in[\![1,\alpha]\!]}, \pi_1(m), h'\right)\right)$$

is valid in pure Asclepius for P without assembly blocks (thus, with the same context of global and local variables),

– lvalues pass the is_writable check,

– label l' belongs to function $f'_\beta \in \mathbb{F}$,

–

$$\forall i \in [\![0, 3]\!], \text{update\_EIP}(\pi_2(m), l', \pi_8(m))(\text{EIP}_i) = \pi_2(m')(\text{EIP}_i)$$

There is a special case to treat: returns. The problem is similar as the one for `RET` statements explained before. We can determine which C calls lead to a C function rather than an assembly label, since we do not have function pointers. But this is not the case for return statements, whose destination is known by the content of the call frame. The additional check to perform is that the return address in the current call frame must be a label in C code, and the function name in the current call frame must be the function containing the `return` statement. This is a protection to avoid returning from a function whose prelude has not been run. It is not possible to do so in pure Asclepius, but assembly gives the power to jump between functions. We will treat later the behavior if the return address is not in $\mathbb{L}_C$ (see subsubsection 8.3.4.2 "Calls from Assembly to C" (page 114)). If the current function is not the one specified in the call frame, the program transits to ω_i.

8.3.3 Entering and Leaving Blocks

Another family of transitions are very easy: entering and exiting assembly blocks. In fact, these transitions do not even involve C or assembly statements, they just update the current program point, without modification on the memory. This way, two consecutive assembly blocks behave just like one.

8.3.3.1 Entering Blocks

Formally, let us consider the component ${}^{a}\texttt{asm\{}{}^{b}S_1 \ldots S_n{}^{c}\texttt{\}}^{d}$. We assume that the current program point is a, and the memory state is m. The memory state can be decomposed:

$$\left(g, h, (c_i)_{i\in[\![0,p]\!]}, sb, ss, cb, cs, pp\right) := m$$

We start by changing EIP to its new value:

$$h_1 := \text{update\_EIP}(h, b, pp)$$

We craft a new assembly call frame, initially empty:

$$c_{p+1} := (0, \mathfrak{h})$$

where $\mathfrak{h}$ is an arbitrary heap chosen non-deterministically. Once we entered the block, the new state is

$$(b, m')$$

where

$$m' := \left(g, h_1, (c_i)_{i \in [\![0,p+1]\!]}, sb, ss, cb, cs, pp\right)$$

8.3.3.2 Leaving Blocks

Exiting the block is slightly more subtle: it depends on whether the next transition is entering a block again. If not, in addition to changing the program point and updating EIP, we need to check if the assembly call frame is empty before removing it. We assume that the current state is (c, m). Let

$$\left(g, h, (c_i)_{i \in [\![0,p]\!]}, sb, ss, cb, cs, pp\right) := m$$

and

$$(size, \_) := c_p$$

If $size \neq 0$, then the system transits to ω_i. Otherwise, we update EIP:

$$h_1 := \text{update\_EIP}(h, d, pp)$$

and remove the topmost call frame:

$$m' := \left(g, h_1, (c_i)_{i \in [\![0,p-1]\!]}, sb, ss, cb, cs, pp\right)$$

to get the new memory state. The new state is

$$(d, m')$$

8.3.3.3 Bridging Blocks

If the next transition enters an assembly block (that is, there are two immediately consecutive assembly blocks), we only update the program point (and EIP), without removing the call frame. Likewise, to enter the next assembly block, we do not add a new call frame. There are two ways to do that. Either, we statically detect consecutive assembly blocks, and we merge them, or adapt the transition between them. Otherwise, we could keep the call frame when exiting the block, but we remove it lazily, if anything happens except entering a block. This is simpler to implement because it is very local: we do not have to check the next statements. But for the sake of the formalization, we prefer to do it statically. Thus, in the special case ^{l}`}` `asm{`${}^{l'}$ where $l \in \mathbb{L}_{\text{asm}}$, in memory state m, we simply update EIP

$$\left(g, h, (c_i)_{i \in [\![0,p]\!]}, sb, ss, cb, cs, pp\right) := m$$

$$h_1 := \text{update\_EIP}(h, l', pp)$$

The new memory state is

$$m' := \left(g, h_1, (c_i)_{i \in [\![0,p]\!]}, sb, ss, cb, cs, pp\right)$$

and with the new program point, the new state is

$$(l', m')$$

It allows not only keeping the state of general-purpose registers, but also the stack suffix must not be empty when leaving a block to immediately enter another one.

8.3.4 Mixed Calls

Mixed calls allow C code to call an assembly function and assembly code to call a C function. This is a nice way to mix C and assembly; this is even the most common way to interface multiple languages, usually through a foreign function interface (FFI). This way of interfacing allows both languages to be strictly separated with explicit interaction (the call). As it is quite clean, we examine these cases now, and we will later see a more vicious interaction between C and assembly: inter-blocks jumps.

8.3.4.1 Calls from C to Assembly

8.3.4.1.1 Calling

When an assembly function is called from C, a size for the current call frame is non-deterministically chosen. Accordingly with the calling convention, the parameters are pushed on the stack in right-to-left order, followed by the return address. The return address is an arbitrary 4-byte integer that does not alias with any variable and is not located in an assembly block (i.e. between the address of the beginning of a block and the address of the end of the block). At this point, we make a copy of the current enriched heap in the caller's call frame. Here, general-purpose registers have arbitrary value but, for callee-saved registers, the same value must be restored at the return.

Formally, we have a call ^{l}`v = f(`$e_1, \ldots, e_n$`)`${}^{l'}$ in a memory state m where `f` is an assembly label. The precondition is (l, m). We call $f \in \mathbb{L}_{\text{asm}}$ the PANACEA label at assembly label `f`. Let us deconstruct m:

$$\left(g, h, (c_i)_{i \in [\![0,p]\!]}, sb, ss, cb, cs, pp\right) := m$$

We start by adding a call frame of size $4(n+1)$: 4 bytes for each parameter and for the return address. The base address of this call stack is

$$csb := sb - \sum_{i=0} p\,\mathrm{cs\_size}(c_i)$$

The call frame should be contained in the stack region of the memory, thus if $csb - 4(n+1) \leqslant sb - ss$, then we transit to ω_a. We assume we have adapted $\langle\!| l |\!\rangle$ and $(\!| e |\!)$ to work with this memory model of type $\mathbb{M}$. This is simply the given definition where we ignore irrelevant parts of the new memory state type.

$$h_1 := h\left[csb - 4i + j - 3 \mapsto (\!| e_{i+1} |\!)\,(m)\,[8j : 8j + 7]\right]_{i=0,j=0}^{n-1,3}$$

We need to add the return address. The return address ra is chosen non-deterministically. It should be consistent with pp: it must be greater than $pp(l)$ and not greater than $pp(l')$. It is not necessarily $pp(l')$ since the caller might want to restore caller-saved registers before running the next instruction. Moreover, there are several constraints: it is located in the code region, that is in $[\![cb, cb + cs - 1]\!]$, but which is not contained in an assembly block. That is,

$$\forall c \in \mathrm{Cmp}\,(P)\,, \exists(\lambda, \lambda') \in {\mathbb{L}_{\mathrm{asm}}}^2 : {}^{\lambda}s^{\lambda'} \wedge s \in \mathrm{Stat}_{\mathrm{asm}} \Rightarrow ra \notin\subseteq [\![pp(\lambda), pp(\lambda')]\!]$$

The new heap is

$$h_2 := h_1\left[csb - 4n + j - 3 \mapsto ra\,[8j : 8j + 7]\right]_{j=0}^{3}$$

The new call stack is an assembly one, thus, it is simply its size, with a dummy heap:

$$c_{p+1} := (4(n+1), \mathfrak{h})$$

where $\mathfrak{h}$ is an arbitrary heap chosen non-deterministically. We save the current heap in the caller call frame, and for that we redefine c_p:

$$c_p := (\pi_1\,(c_p)\,, h_2)$$

Now, we need to update EIP register:

$$h_3 := \mathrm{update\_EIP}(h_2, f, pp)$$

The new memory state is

$$m' := \left(g, h_3, (c_i)_{i \in [\![0, p+1]\!]}, sb, ss, cb, cs, pp\right)$$

thus, the new state in the transition system is

$$(f, m')$$

Let us emphasize that this is not a single state: several values, such as the return address ra, are chosen non-deterministically. Thus, there is in fact a set of possible states.

8.3.4.1.2 Returning

Such a call can only be ended by a `RET` when the stack has the same height as at call time and the top of the stack is the return address, but the parameters may have been modified. Equivalently, we can also jump to the return address with a stack that is 4-byte shorter than the initial one.

A `RET` instruction with an assembly destination is performed as a jump (see subsubsection 8.3.2.1 "Pure Assembly Statements" (page 106) for intra-block jumps and subsection 8.3.5 "Other Control Statements" (page 116) otherwise) and, in any other cases, it fails in ω_a.

At the end of the call, we check whether callee-saved registers have indeed the same value as initially, EAX register is used as returned value and the execution of the C function continues. If an assumption of the calling convention is not matched, the system makes a transition to ω_i (illegal instruction).

Let us formalize this return. We run the statement ${}^{l}\texttt{RET}^{l'}$, in the memory state m. Thanks to the criterion explained in subsubsection 8.3.2.2 "Pure C Statements" (page 107), we can tell if this is a return from a mixed call. If the return address is not a C program point, this is a classical assembly instruction. Let us deconstruct m:

$$\left(g, h, (c_i)_{i\in[\![0,p]\!]}, sb, ss, cb, cs, pp\right) := m$$

We also deconstruct the topmost call frame, this is necessarily an assembly call frame:

$$(size, \_) := c_p$$

and we need the stack saved in the caller's call frame, which is always a C call frame:

$$(\_, hs) := c_{p-1}$$

We check that the stack has the correct height:

$$\forall i \in [\![0,3]\!], hs(\mathrm{ESP}_i) = h(\mathrm{ESP}_i)$$

If this condition is not matched, the system transits to the immediate error ω_i. If it is correct, we know the top of the stack contains 4 bytes for each parameter and 4 bytes for the return address. The state of parameters is irrelevant, but the return address must be unaltered: we shall check if the top of the stack is the same as in hs. The base of the call stack is

$$csb := sb - \sum_{i=0} p - 1\, \mathrm{cs\_size}(c_i)$$

We check whether

$$\forall i \in [\![0,3]\!], h(csb - 4(size - 1) - i) = hs(csb - 4(size - 1) - i)$$

If this condition is false, the return address has been altered, and the system transits to ω_a. Now, we shall check that callee-saved registers are unchanged or have been correctly restored. We check whether

$$\forall r \in \{\mathrm{EBX}, \mathrm{EDX}, \mathrm{ESI}, \mathrm{EDI}, \mathrm{EBP}\}, \forall i \in [\![0,3]\!], h(r_i) = hs(r_i)$$

If this condition is not matched, the system transits to ω_i. Now, we can perform the actual return. Let us determine the return program point. The 4 bytes at the top of the stack is the return address, that is

$$ra := \sum_{i=0}^{3} 2^{8i} h(csb - 4(size - 1) - i)$$

Thanks to pp map, we find the appropriate program point, that is the program point rpp such that $pp(rpp) = ra$. We look for the call instruction just before this label, that is the call such that

$$-\texttt{v = f(}e_1, \ldots, e_n\texttt{)}^{l''}$$

Such a call necessarily exists since a C return address is stored in the saved heap only when executing a call. We copy the returned value from EAX to v. The actual address of v is found in the global environment or the caller's call stack, we can use $\langle\!\langle \texttt{v} \rangle\!\rangle$ for that purpose. We need a memory state where caller's local are visible. We build this state

$$m_1 := (g, h, (c_i)_{i\in[\![0,p-1]\!]}, sb, ss, cb, cs, pp)$$

where the assembly call frame is removed. We can now update the return variable:

$$h_1 := \quad h\left[\langle\!\langle l \rangle\!\rangle(m) + i \mapsto h(EAX_{i+1})\right]_{i=0}^{3}$$

The last thing to do is to change arbitrarily some registers. In particular, we change general-purpose registers and EIP. Let

$$R := \{\mathrm{EAX}, \mathrm{ECX}, \mathrm{EDX}, \mathrm{EBX}, \mathrm{EDI}, \mathrm{ESI}, \mathrm{EBP}, \mathrm{ESP}\}$$

an non-deterministically chosen map $x : R \times [\![0,3]\!] \to \mathbb{I}$, and

$$h_2 := h_1\left[r_i \mapsto x(r,i)\right]_{r\in R, i\in[\![0,3]\!]}$$

We update EIP:

$$h_3 := \mathrm{update\_EIP}(h_2, rpp, pp)$$

The resulting memory state is

$$m' := \left(g, h_3, (c_i)_{i\in[\![0,p-1]\!]}, sb, ss, cb, cs, pp\right)$$

and the new state in the transition system is

$$(rpp, m')$$

Since this way of calling function does not go through a C function prelude, no local variables can be declared as long as the stack ends with an assembly call frame. Calling a C function from assembly is a solution to get back into a nice setup with local variables. In the same idea, since we do not want to run a postlude: an Asclepius return in a context where the current call frame is an assembly call frame is an immediate error (ω_i).

8.3.4.2 Calls from Assembly to C

8.3.4.2.1 Calling

The converse operation consists of an assembly `CALL` to a C function. The number of parameters of the callee is known from its prototype. If the assembly call frame is not deep enough (that is, it is smaller than the cumulative size of arguments), the system terminates with error ω_i. Caller's `CALL` instruction is executed: it pushes the current return address on the stack and jump to the beginning of C function. At this point, before executing the callee, the heap is saved in caller's call frame.

Let us formalize that. We want to run the statement ${}^{l}\texttt{call f}^{l'}$ where `f` is a C function with n parameters called x_1 to x_n, with respective types t_1 to t_n Let us call l'' the first label in function `f`. We call m the current memory state. We ritually deconstruct m:

$$\left(g, h, (c_i)_{i \in [\![0,p]\!]}, sb, ss, cb, cs, pp\right) := m$$

The topmost call frame is an assembly one. We decompose it:

$$(size, \_) := c_p$$

First, if $size < \sum_{i=1}^{n} |t_i|$, the system transits to ω_i, otherwise the execution continues. We need to build the new call frame. First, let us choose a size $s \in \mathbb{N}$. It should be big enough to store local variables and the return address. The base address of the new call stack is

$$csb := sb - \sum_{i=0}^{p} \text{cs\_size}(c_i)$$

Let us initialize the new local environment:

$$e : \{x_i \mid i \in [\![1, n]\!]\} \to \mathbb{A}$$
$$x_i \mapsto csb + 1 + \sum_{j=1}^{i-1} |t_j|$$

Overall, the new call frame is

$$c_{p+1} := ((e, l', s, \mathtt{f}), \mathfrak{h})$$

and we save the heap in the caller's call frame:

$$c_p := (\pi_1 (c_p), h)$$

Now, we simply need to make general-purpose registers unknown, just like before:

$$h_1 := h \left[r_i \mapsto x(r, i)\right]_{r \in R, i \in [\![0,3]\!]}$$

where $x : R \times [\![0,3]\!] \to \mathbb{I}$ is a non-deterministically chosen arbitrary map. We update EIP:

$$h_2 := \text{update\_EIP}(h_1, l'', pp)$$

The resulting memory state is

$$m' := \left(g, h_2, (c_i)_{i\in[\![0,p+1]\!]}, sb, ss, cb, cs, pp\right)$$

and the new state in the transition system is

$$(l'', m')$$

8.3.4.2.2 Returning

Such a calls end with an Asclepius return statement. According to the calling convention, we copy back the original value of callee-saved registers from the saved heap to registers. The return value is copied into EAX and the stack is popped so that only the parameters are left. We can remark that the callee may have modified the parameters since they are local variables from its point of view.

Once again, we have to first check if this is a return from a mixed call, or a regular Asclepius call. This has been already explained: we just need to see if the return label is an assembly program point. Otherwise, see subsubsection 8.3.2.2 “Pure C Statements” (page 107).

We want to run ^{l}`return e;`${}^{l'}$ in the memory state m. Let

$$\left(g, h, (c_i)_{i\in[\![0,p]\!]}, sb, ss, cb, cs, pp\right) := m$$

and

$$((v, rl, size, f), \_) := c_p$$

First, let us check we will indeed run the correct postlude: the label l must be in the function f. If is not, the system transits to ω_i. If everything is correct, we may copy the returned value to EAX, as commanded by the calling convention.

$$h_1 := h\left[EAX_i \mapsto (\!|e|\!)\,(m)\,[8i : 8i+7]\right]_{i=0}^{3}$$

The calling convention also orders the callee to restore some registers. We define this set of registers:

$$\mathcal{R} := \{\text{EBX}, \text{EBP}, \text{ESP}, \text{ESI}, \text{EDI}\}$$

We also need the enriched heap saved in the caller’s call frame:

$$(\_, hs) := c_{p-1}$$

Thanks to this saved heap, we can restore registers:

$$h_2 := h_1\left[r_i \mapsto hs(r_i)\right]_{r\in\mathcal{R}, i\in[\![0,3]\!]}$$

We now need to update EIP:

$$h_3 := \text{update\_EIP}(h_3, rl, pp)$$

The new memory state is

$$m' := \left(g, h_3, (c_i)_{i\in[\![0,p-1]\!]}, sb, ss, cb, cs, pp\right)$$

and the new state is

$$(rl, m')$$

8.3.5 Other Control Statements

In assembly, there is no fundamental difference between `JMP`, `RET` and `CALL`. We can break down the two latter as stack operations and a `JMP`. The stack operation may succeed or fail the same way as real stack operations. Thus, we will just consider the case of `JMP`.

A `JMP` instruction simply updates the current instruction pointer. Thanks to the mapping between assembly program points and addresses, we can deduce whether the destination is indeed a legal program point or if it is outside assembly blocks or in the middle of an assembly instruction. If the destination is not legal, this is an immediate error (ω_a). However, even if the destination is legal, the jump may lead to other problems. There are two cases: whether the destination is in an assembly block in the same function or not.

If the destination is in the same function, local variables may still be accessible. In fact, it depends on how the compiler allocates local variables. If they are all allocated at the beginning of the function, they are all accessible at any point of the body, but if they are allocated when declared, we can access local variables if the sets of local variables alive at the source and at the destination are identical.

To formalize that, we need stronger assumptions on the compiler. Unlike many assumptions made until now, these would not be reasonably generalizable. In our application, C is limited to ANSI C (C89), where local variables should be declared only at the beginning of the function: in the outermost local scope, before the first statement. Because of these assembly blocks, the compiler is very little optimizing, and local variables are always allocated; even if two variables have disjoint lifetime, they will not share a memory cell. The nice consequence is that all intra-function jumps are valid. Formally, to tun ${}^{l}\texttt{jmp k}^{l'}$ in memory environment m, with

$$\left(g, h, (c_i)_{i\in[\![0,p]\!]}, sb, ss, cb, cs, pp\right) := m$$

we simply have to update EIP

$$h_1 := \text{update\_EIP}(h, k, pp)$$

where $k \in \mathbb{L}_{\text{asm}}$ is the label at assembly label `k`. The new memory state is

$$m' := (g, h_1, (c_i)_{i\in[\![0,p]\!]}, sb, ss, cb, cs, pp)$$

and the new state is

$$(k, m')$$

If the destination is in another function, since the stack is not updated, local variables of the destination function are not allocated since we did not run the prelude. Local variables of the source function are not in the current scope and thus cannot be accessed by their name; this is not the concern of the semantics: we can check syntactically the scope of variables. Thus, in the destination function, we can only access global variables, and, if the control eventually returns to the source function, the stack must be in the same state to ensure that the execution continues correctly.

Once again, we need extra hypotheses for such tricky jumps. The safer way to handle them is to forbid it. Sadly, this is not always possible: system developers' hands are more stained that Lady MACBETH's (see [Sha06]); they fear no hack to achieve their goals. In our case, we restrict the problem to functions that have no local variables[†]. Thanks to this restriction, we may handle inter-function jumps exactly like intra-function jumps. It is worth noticing that we allow static local variable, as shown in Listing 7.4 "A complicated dynamic jump" (page 96) since they are statically allocated, and do no depend on a prelude/postlude pair.

If we need to support functions with local variables, we must instrument the semantics to know whether local variables have been allocated: if they have not been allocated while an expression tries to access them, the system must transit to an error state.

8.4 Summary and Extension

We saw how to handle mixed code. We have chosen a single memory model compatible with both C and assembly code. We have seen that transitions can be categorized in a few classes:

- Intra-language transitions can be easily adapted to this mixed code.
- Entering an assembly block requires only to update the current program point and add an empty assembly call frame. Leaving a block is more subtle: if the next transition is to enter another one, we only update the program point, otherwise, we check is the current assembly call frame is empty, and we remove it.
- Mixed calls is a lot of work, but ultimately, the interface is clean. This is very dependent on the calling convention.
- Assembly jumps between blocks require very precise assumptions on the compiler. With our assumptions, these cases are quite easy.

States of the transition system are pairs of a program point (as an element of $\mathbb{L}$) and a memory state. The memory state contains the value of EIP register (instruction pointer). Since the memory state also keeps the mapping between program points and addresses, having both is redundant. In fact, we need the label since it is a simpler way

[†]In section 6.1 "Syntax of Assembly Blocks" (page 81), we mentioned that, in a syntactic extension, assembly functions can be declared as top-level items. Since there is no C function wrapping them, there is no way to declare C local variables. Excluding local variable is not a strong hypothesis: it comes naturally when assembly blocks are wrapped in dummy C functions.

to specify the current program point, and we modeled EIP for completeness. In fact, EIP is never used by its own and will be ignored from abstractions. When its value is needed, we will only use the label symbolically. This is similar to pointers: we do not care about the actual value of a pointer (and we often do not know it), but only to which variable it points. In the same way, the content of EIP is useless; we are only interested to know which program point it designates.

```
 1  void g(void) {
 2      asm {
 3          mov EAX, X[EBP]   ; What should be X to retrieve 42 ?
 4      }
 5  }
 6
 7  void f(void) {
 8      asm {
 9          push 42
10          call g
11      }
12  }
```

Listing 8.4: Accessing deep assembly call frames

We have mentioned in section 7.5 “Peroration” (page 98) that some issues are not handled by this semantics. For instance, we cannot reach a deeper assembly call frame since we do not know the size of C call frames, and thus, the number of bytes to skip. For instance, in Listing 8.4, at line 3, we have a stack made of 4 parts (from the topmost call frame):

- an empty assembly call frame for the assembly block in `g`;
- the C call frame of `g`;
- the assembly call frame of the assembly block in `f` (8-byte long and containing 42 and the return address of the call);
- the C call frame of `f`.

From the assembly call frame in `g`, we would like to access the assembly call frame in `f`. It could be done easily using a pointer: if we had saved the value of ESP after line 9, it would still point to the desired value. But in the study case, it is simply done by using an offset relative to current EBP: we need to skip `g`’s call frame, and the return address pushed by `call g` instruction. To do so, we need to know the size of call frame of C functions, even if they have local variables: for the sake of simplicity `g` does not in our example, but could. This behavior is easy to handle in the semantics: rather than choosing a non-deterministic value for the size of C call-frames, they get a statically chosen size. Of course, it requires more hypotheses on the compiler to be able to ensure that the size of C call frames of each function is constant, and to indeed predict it.

We can go slightly further and try to handle the access to deep C call frames, as illustrated on Listing 8.5. To access a local variable in a deep call frame, we need to precisely know the layout of C call frames. This is a very strong assumption, and it might be tempting to simply make all the stack explicit, up to a few opaque segments (like where register are saved) of known size. But if we do so, we may still ensure that C statements sees variables independently, otherwise, it would wrongly allow some undefined behavior, like array overflow. Just like knowing the representation of signed integers does not allow signed modular arithmetic (see section 6.4 "Assembly-Induced Assumptions" (page 86)), knowing the layout does not allow overflow. For instance, in Listing 8.6, even if we know that `n` is located just after `a`, the loop is still faulty. Indeed, the compiler can assume array overflow does not happen, and generate optimized code accordingly. For instance, since it sees the loop iterates over `a`, and performs at least `4` steps, it may infer that it clears all (and only) the array `a`, and generate an SIMD[‡] instruction to write 16 bytes at once, and not modify `n` at all.

```
void g(void) {
    asm {
        mov EAX, X[EBP]   ; What should be X to retrieve n ?
    }
}

void f(void) {
    int n = 42;
    asm {
        call g
    }
}
```

Listing 8.5: Accessing deep assembly call frames

```
void f(void) {
    int a[4];
    int n = 42;
    int i;
    for(i = 0; i < 5; i++) {
        a[i] = 0;
    }
}
```

Listing 8.6: Array overflow

This particular problem is a bit strange to handle: for signed arithmetic with two's

[‡]single instruction, multiple data

complement, it is easy to distinguish two kinds of addition (wrapping around or not); it is simply a different function to call. Here, it is more technical to know the layout but forbid array overflow. For pure C statements, the concrete structure of the stack must be totally forgotten, and we keep only the stack of environments. But if the heap has still a constant structure, we will accept faulty programs such as Listing 8.6. So, in addition, we shall scramble the heap to randomize the address of variables: we have to put blinders on C not to allow extra behaviors.

We did not solve these problems directly in the semantics since it requires stronger assumptions that we do not want to make mandatory. Moreover, it is not on the analyzed fragment of the study case, and we have no satisfactory abstraction to handle such codes.

Chapter 9

Amical

DETERMINING the size of instructions is a crucial matter to infer the destination of dynamic jumps. This is the job of a library called AMICAL (for AsM Intel in oCAmL)[*], we wrote specially for ASTRÉE as part of this work. Moreover, it can disassemble byte sequences (see subsubsection 2.4.2.1 "A Syntactic Digression" (page 30)), even partly symbolic. This is useful when the opcode is manually specified in the source code using assembler directives. AMICAL also has a standalone version that takes assembly instructions and returns the possible encodings. This mode is very limited since there is no C context (e.g. providing variables), but very useful to debug.

AMICAL solves another problem: determining the operand size. While it can be written explicitly[†], it is usually implicit. In fact, in our study case, such annotations are explicitly written only when they are required. For instance, the instruction

```
mov [EAX], 1
```

is ambiguous because there is no way to know how many bytes are dereferenced by the memory operand; we could write 1, 2 or 4 bytes at once. An assembler would fail to process it, for instance GNU Assembler 2.30 will emit the error message "ambiguous operand size for mov". If we want to write 4 bytes, we can write one of these possibilities:

- `mov DWORD PTR [EAX], 1`
- `mov [EAX], DWORD PTR 1`
- `mov DWORD PTR [EAX], DWORD PTR 1`

It is worth mentioning that this problem exists only with INTEL's syntax, since AT&T's syntax makes the operand size a required suffix of the mnemonic. In AT&T's syntax, the previous example would be written:

```
movl $1, (%eax)
```

in which the suffix `l` (that stands for "long") specifies that the instruction has 4-byte operands. Nevertheless, the problem of instruction size still applies.

There is a major semantic difference between this instruction

[*]Amical is french for "friendly". Indeed, AMICAL is a nice tool that takes care of a painful problem for us.

[†]Using keywords `BYTE PTR` (1 byte), `WORD PTR` (2 bytes), `DWORD PTR` (4 bytes), in INTEL syntax.

```
mov DWORD PTR [EAX], 1
```

and this one

```
mov BYTE PTR [EAX], 1
```

The latter writes 1 in a single byte, while the former writes 1 in lower byte and 0 in the 3 other bytes. Thus, determining the size of memory operands is crucial. Sizes of register operands is trivial, since each register name comes with a size: 4 bytes for registers prefixed by `E`, 1 for registers matching `[A-D][HL]`, 2 for the others.

Clearly, the size of memory operands is important since it has a semantic impact. But, maybe surprisingly, operand size of immediate operands is also crucial, and no more trivial to infer. For instance, the instruction

```
add BL, 256
```

does not add 256 in BL: BL is a 8-bit register, and the immediate operand is expected to be 8-bit long as well. 256 does not fit in a single byte (it is 1 0000 0000B), thus, only the result of the modulo is kept in the binary: 0. This is different from adding 256 in modular arithmetic. Indeed, adding 256 always causes an overflow (and sets EFLAGS register accordingly), which never happens when adding 0. This is a reason why the size of immediate operands has a semantic impact as well.

Moreover, the size of an immediate operand is not always the same as the size of the other operand. For instance, the instruction

```
add DWORD PTR [EAX], 1
```

can be assembled in different ways: the immediate operand can be 1-byte (and sign-extended[‡]) or 4-byte. Such special rules exist to propose smallest encoding for usual operations[§], like adding small numbers. But the instruction

```
add DWORD PTR [EAX], 256
```

does not raise this issue: since 256 does not fit in a single byte, this instruction will be assembled using a 4-byte immediate operand. Since the destination is indeed 32-bit wide, the value of the immediate operand is not affected. While the operand size has no effect on the semantics in the penultimate case, it changes the legal encodings, and thus the possible total lengths. This is an example of how the problem of the instruction size and the problem of operands sizes are linked.

We can remark that these problems appear only because we are working on source code. If we were working with the binary, assembling is already performed, and thus operand size would be explicit and instruction lengths would have been already chosen by the assembler. On the other hand, in binary, syntactic structures have been lost, which

[‡]That is, extended to the 32-bit encoding with the same value when interpreted as an integer with two's complement representation.

[§]Smaller encoding allows smaller executable file size, and quicker decoding, thus faster execution.

are very useful to analyze C. The question of simplified compiled program¶ vs. original source code is a leitmotiv in program analysis. As a rule of thumb, we often choose the original source code since it contains all syntactic information used by heuristics to improve precision. Indeed, to use heuristics on compiled programs, we often need some steps of decompilation to try to retrieve original structures. Similar questions will be discussed later.

9.1 The x86 Instruction Encoding

Before explaining how assembling and disassembling work, we will explain the instruction format. The following does not include anything from the 64-bit generation as it is irrelevant here.

Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate

Figure 9.1: Instruction format

Each instruction is encoded according to the general pattern shown on Figure 9.1. Let us detail each field.

There are up to 4 1-byte prefixes, at most one of each prefix class, and they are all optional. The 4 classes of prefixes are:

– Lock and repeat prefixes. They are mainly out of our scope but quickly discussed in section 16.5 “String Instructions” (page 204).
– Segment override prefixes. They allow overriding the segment in which a memory operand is resolved, rather than using register-directed disambiguation.
– Operand-size override prefix. It allows 16-bit parameters while working in 32-bit mode, and conversely.
– Address-size override prefix. It allows 16-bit addressing style in 32-bit mode, and conversely.

The opcode identifies the instruction. It is 1 or 2 bytes long. As the opcode defines the operation, it specifies what kind of parameters is expected and their size.

ModR/M and SIB are two 1-byte fields used to encode register and memory operands. These fields are optional since they are not needed when the instruction requires no such operand. It is important to clarify what can be encoded here. These bytes can store a register-only operand (denoted r) and an operand that can be either a register or a memory operand (denoted r/m). For instance, in the instruction `mov EAX, EBX`, there are two register operands, thus one of them is encoded as a register-only operand, and the other as a register-or-memory operand.

Displacement field can be 1, 2 or 4 bytes large (depending on ModR/M and SIB fields) and represent a literal offset in a memory operand.

¶It can be a full compilation in the usual meaning, or a mere simplification to remove redundant constructs of C.

Immediate field is also 1, 2 or 4 bytes large (depending on the opcode) and stores an immediate operand, that is a literal value.

Yet, there are irregularities to this general layout. Two of them are quite interesting in our scope. For some opcode, additional precision can be stored in the ModR/M byte. Specifically, this byte holds 3 bits to specify a register operand that may be used to precise the opcode, in which case, this instruction cannot have a register-only operand. The most important example is the case of opcodes 80H, 81H and 83H. Depending on the supplementary bits in the ModR/M byte, it can either be instructions `ADD`, `OR`, `ADC` (add with carry), `SBB` (subtract with borrow), `AND`, `SUB`, `XOR` and `CMP`. All these variants of these instructions have no register-only operand but a register-or-memory operand (r/m) and an immediate operand. The second irregularity is somehow the opposite: the opcode contains the argument. A classic example is the instruction `DEC` (decrement). There are two variants: one with a register/memory operand that has a regular encoding, and one with register-only operand. For the latter variant, the 5 upper bits of the opcode are 01001B and the 3 lower bits encode the register to decrement. Thus, 01001000H is `DEC EAX`, 01001001H is `DEC ECX` (sic), and so on. We can consider this family of instructions as 8 zeroary opcode to interpret this case as 8 regular encoding rules.

It is interesting to note that arguments are not ordered. For instance, the only difference between `mov EAX, [EBX]` and `mov [EBX], EAX` is the opcode, but the parameters will be encoded in the same way.

Assembling rules are provided as on Table 9.1. The first column describes the binary encoding: the first number is the opcode in hexadecimal and the following specifies the fields used in the encoding. For instance, ib, iw and id are respectively immediate arguments of size 8, 16 and 32. The specifier /r means that there is a ModR/M byte (and optionally a SIB byte and a displacement) to encode a register-only operand and a register-or-memory operand.

The second column is about the syntax of the assembly instruction. For instance, the first line of Table 9.1 allows `AL` as left-hand side operand and any 8-bit immediate operand on the right-hand side. The description column is for documentation purpose.

The correspondence between the argument and the encoding is not explicitly stated, but there is no ambiguity since each kind of argument may only have one of each kind. For instance, for the instruction `add [EAX], EBX`, the only possible is "ADD r/m32, r32". Since the ModR/M byte can encode exactly one register operand and one r/m operand, there is no ambiguity.

The assembling rule is subject to ambiguity. For instance, `add EAX, EBX` can be assembled using the rule "ADD r/m32, r32" or "ADD r32, r/m32". In the first case we result with encoding 01 D8 while, with the latter rule, we get 03 C3. Here, both encoding have the same size (2 B), so, in our case, it makes no difference since we are only interested in the size of instructions. But other cases may have several encoding of various sizes. We already gave the example of `add EAX, 3` in section 5.5 "A More Symbolic Semantics" (page 77).

Assembling has another purpose: deciding the size of memory operands. The size of register operands is unambiguous, since it is specified by their name, for instance EAX is

Opcode			**Instruction**		**Description**
04	ib		ADD	AL, imm8	Add imm8 to AL
05	iw		ADD	AX, imm16	Add imm16 to AX
05	id		ADD	EAX, imm32	Add imm32 to EAX
80	/0	ib	ADD	r/m8, imm8	Add imm8 to r/m8
81	/0	iw	ADD	r/m16, imm16	Add imm16 to r/m16
81	/0	id	ADD	r/m32, imm32	Add imm32 to r/m32
83	/0	ib	ADD	r/m16, imm8	Add sign-extended imm8 to r/m16
83	/0	ib	ADD	r/m32, imm8	Add sign-extended imm8 to r/m32
00	/r		ADD	r/m8, r8	Add r8 to r/m8
01	/r		ADD	r/m16, r16	Add r16 to r/m16
01	/r		ADD	r/m32, r32	Add r32 to r/m32
02	/r		ADD	r8, r/m8	Add r/m8 to r8
03	/r		ADD	r16, r/m16	Add r/m16 to r16
03	/r		ADD	r32, r/m32	Add r/m32 to r32

Table 9.1: Assembling rules for `ADD`

always a 4-byte operand while AX is a 2-byte one. But the size of memory operands may be unclear without context. One may specify it explicitly, it is even required in some cases. For example, the instruction `add [EAX], 1` may be assembled using five rules (all rules of the form "ADD r/m?, imm?"). But these rules does not have the same semantics since they do not dereference the same size in memory, thus we should specify the size of the memory operand so as to decide which rule to choose. We can, for instance, write `add DWORD PTR [EAX], 1` to specify that memory operands are 32-bit long and thus legal rules have the form "ADD r/m32, imm?". There are still two legal rules, but we can decide which to choose according to the immediate operand, and if both rules are still legal, they have the same semantics. But even when it is mandatory to specify the size, it may be not so trivial, for instance, we can write `add [EAX], DWORD PTR 1`, where we specify the size of the immediate operand which is enough to decide the size of the memory operand. Even when specifying the size is not required (when there is no ambiguity), it is not always obvious to decide the size since it depends on the mnemonics and the other operand (if applicable).

The following sections will explain how we exploit these rules to deduce the size of instructions and to disassemble binary sequences.

9.2 Assembling

Amical is in charge of deducing the size of instructions and memory operands. But it goes even further: it assembles completely so that it produces the final machine code. Indeed, once we have done everything to deduce the size of an instruction, computing the corresponding machine code is quite easy. Moreover, it helps to debug since we can see the binary sequences behind each size. In particular, we can use mainstream tools, to disassemble the sequence obtained with Amical, and check if it matches the input instruction. During Amical development, we assembled binary sequences using GNU Assembler 2.30 (using directive to manually define put bytes), and disassemble the resulting object file using GNU objdump 2.30.

Amical works in 4 mains step:

1. It selects all the assembling rules that match the mnemonic and the operands.
2. Among the rules selected previously, it excludes some of them according to additional assumptions. Here, it already knows the operand sizes according to the remaining rules.
3. It unifies the operands with the fields of the binary encoding. At this point, we can compute the size.
4. It puts the fields in the appropriate order to get the binary encoding.

To select the assembling rules, we first select the rules with the appropriate mnemonic. This is really easy when the rules are stored in an associative container whose keys are the mnemonics and values are set of rules. This step is fast and restrains greatly the set of rules to examine.

For the remaining rules, we must look at each one independently. It is an acceptable cost since they are relatively few rules for a given mnemonic[**]. For each rule, we check if each parameter of the actual instruction matches the corresponding operand pattern. For instance `1` can be `imm8`, `imm16` or `imm32`, and `EAX` can be `r32` or `r/m32`.

At this point, we have selected all the rules that match the instruction. It is important to get all rules because we do not know which one the assembler would choose. Indeed, while assemblers claim to always select the smallest encoding possible, this is not true[††]. Sometimes this is just a bug, but sometimes it is because of alignment or to generate relocatable code (or any technical reason)[‡‡]. Yet there might be good reasons, it is hard to predict when the assembler will use a suboptimal encoding. Moreover, a longer encoding could be chosen because of a reason that may not always apply[§§]. So, to stay on the safe side, we must consider all possible encodings. However, sometimes, we have selected rules that are actually discarded by additional hypotheses. For instance, the instruction `push [EAX]`, in 32-bit mode, pushes a 32-bit value on the stack, while in 16-bit mode (or 32-bit with operand size override), it pushes only a 16-bit value. The assembler assumes that it knows the current execution mode and that we want the default behavior, i.e. to push a 32-bit value. But Amical would have selected both rules “PUSH r/m16” and “PUSH r/m32”. At this point, we discard the 16-bit version since assumptions of the assembler avoid an ambiguity here. More generally, Amical needs to know the current execution mode and it will only select rules that are available in this execution mode if there are any, discarding rules that require a operand-size override prefix. If there is no rule for the current execution mode, Amical keeps all

[**]Up to 23 for rules implemented in Amical.

[††]Or at least, not true without more constraints.

[‡‡]Given a set of such constraints, some assemblers may be optimal, though it would be wise to stay skeptical.

[§§]Assembler usually have very limited static analysis abilities, making them usually unable to decide whether a suboptimal encoding is indeed required.

rules previously selected. This allows AMICAL to reject rules that would require an operand-size override that has not been explicitly required, while still accepting them if the operand size is constrained. For instance, the instruction `push WORD PTR [EAX]` explicitly specifies that the operand size is 16 bits. Thus, only the rule "PUSH r/m16" would match, and not "PUSH r/m32". Since they are no rule for the current execution mode, the other one is selected.

Here, we have selected all the rules we will work with. Since patterns of operands specify the size, we already know the possible sizes of all operands, but we still need to compute the size of the encoded instruction. For each rule, we determine which fields are allowed based on the specifiers of the "Opcode" column of Table 9.1, and infer how the arguments are arranged in these fields. Moreover, we must detect whether this rule needs an operand-size override or an address-size override prefix. This is mostly a matter of case analysis. We have here computed the content of all allowed fields in each rule. The sum of the sizes of all fields is the size of this encoding. But, to allow easier debugging, we put these fields in a row in the right order to give the correct instruction.

AMICAL is able to assemble instructions that contain labels and C variable names. Once again, it is only possible with additional hypotheses, e.g. the size of C variables must be provided. For labels, if they are in other blocks, we have no idea how far they are, and thus, whether the offset fits in a 8-bit, 16-bit or 32-bit integer, and we must consider that everything is possible. If the label is in the same block as its usage, we may compute possible distances as sums[¶¶] of possible sizes of all instructions in between. Thanks to the set of possible distances, we can discard some possibilities: when the label is at least 129 B away, the distance cannot fit in a 8-bit relative offset; or when it is at least 32 769 B away, the relative offset can only by 32-bit wide. Contexts containing the position of labels can be provided to AMICAL or computed automatically when it works on a sequence of assembly instructions and labels.

9.3 Disassembling

We already mentioned that we also need to disassemble binary sequences. This was briefly discussed in subsubsection 2.4.2.1 "A Syntactic Digression" (page 30) and illustrated on Listing 2.5 "Writing a far call using defines" (page 31). We will not linger on the reasons that lead the developer to write instructions in such a displeasing way; we will only explain how we solve that problem and find the correct instruction. Indeed, the internal representation of instructions in AMICAL can be as precise as the machine code (and thus more precise than assembly language), thus even instructions that must be written in binary in the source code can be expressed unambiguously in the internal format.

The problem of disassembling is much easier than assembling. First, we precisely know the size of the instruction. Moreover, finding the corresponding statement is very linear.

[¶¶]More precisely, the MINKOWSKI sum.

To disassemble, Amical takes a list of bytes (or symbolic bytes, such as labels). Then, they are consumed in very simple steps that follow the structure of instruction format as shown on Figure 9.1 “Instruction format” (page 123).

1. We read prefixes: they are 12 of them in 4 categories. Each category may appear up to once.
2. We read one byte of the opcode: if it is 0FH, the opcode is made of 2 bytes, and we read a second byte; otherwise the opcode is a single byte.
3. Based on the prefixes, the opcode and the current execution mode (which is a part of the configuration), we can find the corresponding assembling rule. Since different rules give different encodings, we are sure there is only one possible rule.
4. Using the specifier of the “Opcode” column, we know which fields are present: we fill them with the appropriate number of bytes. At this point it is interesting to remark that literal bytes, such as a label, must all be consumed at once and cannot be split.
5. From the value of each field and the form of the instruction, we can infer by which field is encoded each parameter, and thus we can rebuild the whole instruction. Leftover bytes belongs to the next instruction.

Disassembling is really the easy way: there is only one involved rule, and we know the most precise representation, the binary sequence.

9.4 A more Concise Syntax for Assembling Rules

For each opcode, there can be many assembling rules, but they are often several of them that are very similar and follow quite regular patterns. Let us look again at the case of `ADD` instruction, the rules are given in Table 9.1 “Assembling rules for `ADD`” (page 125): there are 14 cases. But we can observe some regularity that is even documented in [Int20]. First, 16-bit and 32-bit versions share the same opcode. Moreover, between the 8-bit and the 16/32-bit version, the difference is only the last bit of the opcode. When the two operands are writable (typically, a register-only operand and a register-or-memory operand), the direction is given by bit 1 (counting from 0), so between “ADD r32, r/m32” and “ADD r/m32, r32”, only one bit in the opcode is flipped.

To be able to feed all rules to Amical without having to write too much manually (with the associated risk of mistakes, especially coming from copy-pastes), we developed a language to exploit patterns in assembly rules such that each line expand to several rules.

For instance, the 14 rules of `ADD` are reduced to 4 combined rules, as show on Table 9.2. The first, third and fourth lines expand into 3 rules each, and the second line expands to 5 rules.

The details are not relevant here, but we can give the overall idea. Since rules can differ from very few, we make these variations parameterizable. After the opcode, these parameters allow changing some bits, by default from the lower bits, but it can be specified otherwise. After a specifier or an operand pattern, it allows changing the size by using the “w” parameter (byte-size or full-size) and the “osa” parameter (operand

Opcode			**Instruction**		**Description**
04.w	i.w.osa		ADD	A__.w.osa, imm.w.osa	Add imm to A__
80.s.w	/0	i.w.osa.s	ADD	r/m.w.osa, seimm.w.osa	Add (possibly sign-extended) imm to r/m
00.w	/r		ADD	r/m.w.osa, r.w.osa	Add r to r/m
02.w	/r		ADD	r.w.osa, r/m.w.osa	Add r/m to r

Table 9.2: Combined assembling rules for `ADD`

size attribute: 16- or 32-bit).

Opcode			**Instruction**		**Description**
	70.tttn	cb	J.tttn	rel8	Conditional jump with 8-bit offset
0F	80.tttn	c.osa	J.tttn	rel.osa	Conditional jump with 16/32-bit offset

Table 9.3: Combined assembling rules for `Jcc`

There are 3 other parameters to allow rules to be compressed even more efficiently where possible. We will not detail everything and the subtleties of the syntax, but we can give a remarkable example: the case of conditional jumps. There are 16 kinds of conditional jumps and, as some of them have aliases, there are 30 mnemonics. Since each kind of condition is always encoded in the same way and use the same kind of suffix[***], we can compress them efficiently. The result is shown in Table 9.3. The first line expand to 30 rules, while the second line, which has an additional parameter (16- or 32-bit), is expanded to 60 rules. To achieve such compression, we allow the mnemonic to be also parameterizable: the parameter tttn is understood as the encoding of the condition when it is part of the opcode, but in the mnemonic, it is interpreted as the condition name (and various aliases).

Of course, a rule without parameter will not be expended, and kept as is. This allows specifying rules explicitly, just like in [Int20].

[***]Conditions are encoded in a 4-bit field called tttn in [Int20].

Part II

Abstract Interpretation and the Astrée Analyzer

Chapter 10

Introduction

To GUARANTEE the safety of a program we must check that all executions are correct. As a consequence of RICE's theorem [Ric53], this is impossible to do automatically. A solution to still check all executions without giving up the benefit of an automatic process is to check a bigger set of executions: by adding some not-actually-possible executions, we can make a bigger set on which the problem is decidable. The drawback is clear: if some impossible executions we added are faulty while all possible executions are error-free, we detect an error that cannot actually happen. Moreover, for every way to choose these additional impossible executions, there exists correct programs on which the analysis will unjustly detect errors. On the bright side, if the program is correct even with the extra traces, possible executions are necessarily also correct.

The way we add extra executions is not entirely artificial. Since we cannot automatically deduce the exact set of possible states of a program, we only work with a partial knowledge of possible states, thus allowing impossible states. This technique is called abstract interpretation. Before diving into mathematical details, let us illustrate abstract interpretation thanks to simple examples.

The most canonical example, given in [CC77], is about the sign of the result of arithmetic operations given the sign of the operands. For instance, if an operand is negative and the other is positive, the result of the multiplication is negative. All the cases are summed up in Table 10.1. With a partial knowledge of the input, as we just know the sign but not the value, we can already deduce properties on the output. Of course, the result is also imprecise: we cannot get an exact result, but hopefully, we are only interested in the sign.

$\times$	$+$	0	$-$
$+$	$+$	0	$-$
0	0	0	0
$-$	$-$	0	$+$

Table 10.1: Rule of signs for multiplication

With another operation, the partial information may be insufficient. Let us consider the addition instead of the multiplication. We know that the sum of two positive (resp. negative) numbers is positive (resp. negative). But when a term is positive and the other is negative, we cannot say the sign of the result in general. In addition to the sign, we would need to know which has the bigger absolute value. The rule is given by Table 10.2. In this table ? means that we cannot say anything about the result: it can be positive, negative or null, i.e. the partial knowledge of the input is not precise enough to deduce anything. This special value (the complete absence of information) is usually denoted $\top$. We can remark that for the addition, if one of the term has an unknown sign, we cannot tell anything on the result. This is not an intrinsic property of $\top$, for instance, the product of ? and 0 is always 0. Nevertheless, getting this most imprecise value is often a bad omen, and it usually spreads quickly. When we lose all precision, we cannot deduce anything non-trivial (something that is not always true or always false).

$+$	$+$	0	$-$	?
$+$	$+$	$+$	?	?
0	$+$	0	$-$	?
$-$	?	$-$	$-$	?
?	?	?	?	?

Table 10.2: Rule of signs for addition

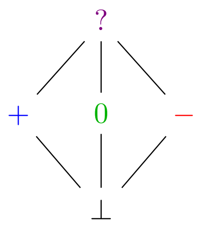


Figure 10.1: Sign lattice

The partial knowledge of a value is called an abstract value. Each abstract value represents a set of possible numbers, for instance $+$ stands for all positive numbers, i.e. $\mathbb{N}^*$ (or $\{1, 2, 3, 4, \ldots\}$). The interpretation of an abstract value is the concretization. Some abstract values are more precise than others, for instance ? (that stands for all integers i.e. $\mathbb{Z}$) is coarser than $+$; but some are not comparable, like $+$ and $-$. We can present the allowed abstract values as on Figure 10.1: the higher an abstract value is, the less precise it is. At the very bottom, the special value $\perp$ stands for the set of all integers that are simultaneously positive, negative and null: this represents the empty set, since no concrete value matches this constraint. Such a representation is called a Hasse diagram.

Another simple example is searching a word in a dictionary. The standard method is a binary search: we start somewhere in the dictionary (usually, hinted by the rank of the first letter in the alphabet, but this is not a requirement), and depending on the word we see, we know if the one we search is before or after. The principle behind is that we know that all the words that come before a given word are a subset of all the sequences of letters that are smaller (according to lexicographic order) than the given word. We do not need to know the exact set of words: we may not know all the words of the dictionary or, on the contrary, some words may be missing. But if the word we search is not in the huge set of all sequences that would come before, it can only be after (if it is in the dictionary). Here, concrete values are sets of words and abstract values are properties of the form “all words before (resp. after) the word w”.

Abstract interpretation can be also found in disguise in many computer science methods. For example, let us consider the maximum satisfiability problem (MAX-SAT): given a formula in conjunctive normal form, what is the maximum number of clauses that can be satisfied by an assignment of Boolean variables, or alternatively, is it possible to satisfy at least K clauses (for a given K). This problem is well-known to be NP-complete, so we do not know polynomial algorithm to solve it, but we can strive to speed up resolution algorithms. Exhaustive search for a formula with V Boolean variables requires trying 2^V assignments, which is quickly prohibitive. Fortunately, this may be improved easily. We use by a DPLL-like algorithm [DLL62]: it works by selecting an unassigned variable*, assigning it a truth value, simplifying the formula, proceeding recursively to find a solution, and eventually backtracking to try the other truth value for the selected variable. From a partial assignment, we can tell if a clause is satisfied (if at least a literal is *tt*), refuted (if all literals are *ff*), or has still an unknown truth value (if all assigned literals are *ff* and there is at least one unassigned literal). Let us assume that we have already found a solution that satisfies m clauses, but we have not yet finished the search. Let us also assume that with the current partial assignment, i clauses are satisfied, j are refuted and l have still an unknown truth value. In the best case, at most $i + l$ clauses can be satisfied by completing the current partial assignment. If $i + l \leqslant m$, it is useless to continue the search with this partial assignment: it cannot give a better solution. Here, without knowing exactly which clauses will be satisfied by deeper search, we can tell whether the current state might lead to a better solution (otherwise, it definitely cannot), allowing us to reduce the search space. This is an example of branch and bound algorithm. The common idea in all these algorithms is to cut some branches of a search tree by recognizing early when an inner node cannot have leaves that beat the current best solution. This can be understood as abstract interpretation as inner nodes encode partial knowledge of a candidate solution†, and we only want to deduce bounds on the objective function.

In the context of program analysis, we want to approximate the semantics of the program: we run the program using imprecise states. At each step, from an approximate description of the precondition, we compute an approximation of the postcondition. It allows us to decide whether forbidden values (like a null variable just before dividing by this variable) are indeed avoided or if we cannot exclude them (though it does not mean they are really possible.).

*The selection order might be static, heuristic-directed or even coming from an oracle.

†Thus, this is a set of candidate solutions.

The following of this part will explain the bases of abstract interpretation in general. First, we will explain the theory of abstract interpretation, from the order theory. We will focus on parts that are useful afterwards (in this part and the following ones), we will not give a comprehensive overview of abstract interpretation. Since the general theory does not provide an effective way to build abstract analyzers, we will then study the construction of complex abstract domains by product and reduction. Finally, we will explain the general architecture of ASTRÉE, and some implementation details relevant for the following, either to explain some abstraction choices, or to give some context to subsequent comments on the implementation.

Chapter 11

Abstract Interpretation Theory

THIS CHAPTER explains bases of abstract interpretation. The erudite reader that is aware of this matter may skip it safely.

11.1 Ordering and Lattices

This introductory chapter may be quite dry, but it requires little background. We start by elementary definitions and properties of order theory.

Definition 11.1 – Poset

A poset (for partially ordered set) is a tuple $(E, \sqsubseteq)$ where E is a set and $\sqsubseteq$ an order relation over E, that is a relation which satisfies:

- reflexivity: $\forall x \in E, x \sqsubseteq x$
- antisymmetry: $\forall (x, y) \in E^2, (x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y$
- transitivity: $\forall (x, y, z) \in E^3, (x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z$

In abstract interpretation, elements of posets are usually properties about the state of a program (or its execution trace). If an element x is smaller than an element y, then x is more precise than y, x carries more data. Indeed, it is always easy to say that anything can happen (the biggest element), while knowing the exact values of each variable is way more interesting.

Proposition 11.1 – Duality of posets

Let $\mathcal{E} = (E, \sqsubseteq)$ be a poset. Let $\sqsupseteq$ the relation defined by:

$$\forall (x, y) \in E^2, a \sqsupseteq b :\Leftrightarrow b \sqsubseteq a$$

The tuple $(E, \sqsupseteq)$ is a poset, called the dual poset of $\mathcal{E}$.

Proof. – Reflexivity. Let $x \in E$. We have $x \sqsubseteq x$, thus $x \sqsupseteq x$.

– Antisymmetry. Let $(x, y) \in E^2$. We assume that $x \sqsupseteq y \wedge y \sqsupseteq x$. Thus, $y \sqsubseteq x \wedge x \sqsubseteq y$, then $x = y$.
– Transitivity. Let $(x, y, z) \in E^3$. We assume that $x \sqsupseteq y \wedge y \sqsupseteq z$. We have $y \sqsubseteq x \wedge z \sqsubseteq y$, thus $z \sqsubseteq x$ and $x \sqsupseteq z$.

□

The notion of duality allows theorems to be easily transposed when reversing the order relation.

Notation 11.1

Given a poset $\mathcal{E}$, we denote $\overline{\mathcal{E}}$ the dual poset of $\mathcal{E}$.

Proposition 11.2

For all poset $\mathcal{E}$,

$$\overline{\overline{\mathcal{E}}} = \mathcal{E}$$

Proof. Let $\mathcal{E} = (E, \sqsubseteq)$ a poset. The dual is $\overline{\mathcal{E}} = (E, \sqsupseteq)$ where

$$\forall (x, y) \in E^2, a \sqsupseteq b :\Leftrightarrow b \sqsubseteq a$$

The dual of the dual is $\overline{\overline{\mathcal{E}}} = (E, \sqsubseteq')$ where

$$\forall (x, y) \in E^2, a \sqsubseteq' b :\Leftrightarrow b \sqsupseteq a$$

Thus

$$\forall (x, y) \in E^2, a \sqsubseteq' b :\Leftrightarrow a \sqsubseteq b$$

□

Notation 11.2

Let $(E, \sqsubseteq)$ a poset, $A \subseteq E$, and $x \in E$. We denote

$$\begin{aligned}
A^{\downarrow\sqsubseteq} &:= \{x \in E \mid \exists y \in A : x \sqsubseteq y\} \\
A^{\uparrow\sqsubseteq} &:= \{x \in E \mid \exists y \in A : y \sqsubseteq x\} = A^{\downarrow\sqsupseteq} \\
x^{\downarrow\sqsubseteq} &:= \{x\}^{\downarrow\sqsubseteq} \\
x^{\uparrow\sqsubseteq} &:= \{x\}^{\uparrow\sqsubseteq}
\end{aligned}$$

This notation will be useful in Part IV "A Reduced Product with Ghost Variables" (page 229).

Definition 11.2 – Total order

Let $(E, \sqsubseteq)$ be a poset. We say that $\sqsubseteq$ is a total order if

$$\forall (x, y) \in E^2, x \sqsubseteq y \vee y \sqsubseteq x$$

Posets are rarely totally ordered in abstract interpretation, this is a much too strong structure. In a totally ordered set, there are only various levels of precision, we cannot express two unrelated states, with the incomparable precision. Total orders are useful only on subsets of posets. Such orders appear (indirectly) in the hypotheses of theorems about loop stabilization (fixpoint theorems). This is because the semantics of loops keeps previous iterations, thus we get an increasing sequence of states, that is, a sequence of states less and less precise, totally ordered.

Definition 11.3 – Extremum

Let $(E, \sqsubseteq)$ be a poset, $A \subseteq E$ and $x \in A$. x is a minimum of A if

$$\forall y \in E, x \sqsubseteq y$$

x is a maximum of A if

$$\forall y \in E, y \sqsubseteq x$$

Proposition 11.3 – Unicity of extremum

Let $(E, \sqsubseteq)$ be a poset, $A \subseteq E$ and $(x, y) \in A^2$. If x and y are minima (resp. maxima) of A, then $x = y$.

Proof. We assume that x and y are minima of A. Because x is a minimum and $y \in A$, we have $x \sqsubseteq y$. Symmetrically, we prove $y \sqsubseteq x$. By antisymmetry, $x = y$. □

Notation 11.3 – Extremum

Let $(E, \sqsubseteq)$ be a poset, $A \subseteq E$. The minimum of A (if it exists) is denoted

$$\min_{\sqsubseteq} A$$

or

$$\min A$$

if the order is clear in the context.
Similarly, the maximum (if it exists) is denoted The minimum of A is denoted

$$\max_{\sqsubseteq} A$$

or

$$\max A$$

if the order is clear in the context.

Definition 11.4 – Least upper bound

Let $(E, \sqsubseteq)$ be a poset, $A \subseteq E$ and $x \in E$.
x is an upper bound of A if

$$\forall y \in A, y \sqsubseteq x$$

x is a least upper bound of A if

$$(\forall y \in A, y \sqsubseteq x) \wedge (\forall z \in E, (\forall y \in A, y \sqsubseteq z) \Rightarrow x \sqsubseteq z)$$

Hence, the least upper bound is, as indicated by its name, an upper bound that is less than any other upper bound. We see in the next result that when it exists, it is unique hence it can be called *the* least upper bound. In abstract interpretation, when we know that either the property x or y is true, any upper bound of $\{x, y\}$ is correct (since it is less precise than x and y), and the least upper bound is the best of these approximations.

Proposition 11.4 – Unicity of least upper bound

Let $(E, \sqsubseteq)$ be a poset, $A \subseteq E$ and $(x, y) \in E^2$.
If x and y are least upper bounds of A, then $x = y$ (i.e. the least upper bound is unique).

Proof. Since x is a upper bound of A and y a least upper bound, then $y \sqsubseteq x$. Symmetrically $x \sqsubseteq y$. Thus, by antisymmetry of $\sqsubseteq$, $x = y$. □

Notation 11.4 – Least upper bound

When it exists, the least upper bound of A in the poset $\mathcal{E}$ is denoted $\sqcup_{\mathcal{E}} A$.

We can omit the subscript, and chose a notation that matches the symbol for the order of the poset. For instance, if the order is denoted $\subseteq$, the least upper bound can be noted $\cup$; if the order is denoted $\leqslant$, the least upper bound may be written $\vee$.

Definition 11.5 – Greatest lower bound

Let $(E, \sqsubseteq)$ be a poset, $A \subseteq E$ and $x \in E$.
x is a lower bound of A if

$$\forall y \in A, x \sqsubseteq y$$

x is a greatest lower bound of A if

$$(\forall y \in A, x \sqsubseteq y) \wedge (\forall z \in E, (\forall y \in A, z \sqsubseteq y) \Rightarrow z \sqsubseteq x)$$

We can remark that the greatest lower bound of A in a poset $\mathcal{E}$ is the least upper bound of A in $\overline{\mathcal{E}}$. The meaning in abstract interpretation is also dual: if a lower bound of $\{x, y\}$ is true, then x and y are both true. In particular, the greatest lower bound is the coarsest value that allows this deduction.

Proposition 11.5 – Unicity of greatest lower bound

Let $(E, \sqsubseteq)$ be a poset, $A \subseteq E$ and $(x, y) \in E^2$.
If x and y are greatest lower bounds of A, then $x = y$ (i.e. the greatest lower bound is unique).

Proof. Similar to proof of proposition 11.4, by duality. □

Notation 11.5 – Greatest lower bound

The greatest lower bound of A in poset $\mathcal{E}$ is denoted $\sqcap_{\mathcal{E}} A$.

The comment about the notation of the least upper bound applies here also.

Definition 11.6 – Complete lattice

A complete lattice is a poset $\mathcal{E} = (E, \sqsubseteq)$ such that $\forall A \subseteq E$, $\sqcup_{\mathcal{E}} A$ exists.

Proposition 11.6 – Alternative definition

Let $\mathcal{E} = (E, \sqsubseteq)$ a complete lattice, thus $\forall A \subseteq E$, $\sqcap_{\mathcal{E}} A$ exists.

Proof. Indeed, $\sqcap_{\mathcal{E}} A = \sqcup_{\mathcal{E}} \{y \in E \mid \forall x \in A, y \sqsubseteq x\}$. □

Proposition 11.7 – Duality of complete lattices

Let $\mathcal{E}$ be a poset. If $\mathcal{E}$ is a complete lattice, so is $\overline{\mathcal{E}}$.

Proof. By duality, assuming that the greatest lower bound always exists implies that the least upper bound exists. □

This is an alternative definition of complete lattices.

Proposition 11.8

Let $\mathcal{E} = (E, \sqsubseteq)$ be a complete lattice; E is not empty.

Proof. $\varnothing \subseteq E$, thus $\sqcap_{\mathcal{E}} \varnothing$ exists and belongs to E. □

Notation 11.6 – Extrema in a complete lattice

In a complete lattice $\mathcal{E} = (E, \sqsubseteq)$, we denote $\bot := \sqcup_{\mathcal{E}} \varnothing = \sqcap_{\mathcal{E}} E$ and $\top := \sqcup_{\mathcal{E}} E = \sqcap_{\mathcal{E}} \varnothing$

Thanks to these properties, we can prove many properties on complete lattices. They are very nice structures for abstract interpretation: the least upper bound is useful when a program point is accessible from multiple points, greatest lower bound allows refining a state given a known constraint.

Definition 11.7 – Lattice

A lattice is a poset $\mathcal{E} = (E, \sqsubseteq)$ such that $\forall (x, y) \in E^2$, $\sqcup_{\mathcal{E}}\{x, y\}$ and $\sqcap_{\mathcal{E}}\{y, x\}$ exist.

Proposition 11.9

Every complete lattice is a lattice.

Proof. Indeed, if the least upper bound exists for any subset, it exists a fortiori for any pair. □

Lattices are weaker than complete lattices, some theorems do not hold on them, but some structures used in abstract interpretation may not have the structure of a complete lattice.

Definition 11.8 – Monotone maps

Let $(E, \sqsubseteq)$ be a poset and $f : E \to E$. f is monotone if and only if

$$\forall (x, y) \in E^2, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

In semantics, monotone maps are a hypothesis of many theorems, and fortunately, they are very common. Indeed, it is very natural to think that if the precondition is less precise (greater), the postcondition will be less precise as well. It would be odd to gain precision on the result by losing some on the input. More inputs means more outputs.

We can note that this is true for the concrete execution, but not necessarily for an approximation of the execution. For instance, if approximate values are in a subset of concrete values, we can approximate a state by picking any bigger one that is an allowed approximate value. If the state is itself an allowed approximation, there is nothing to do. But if it is not, we may be very bad at finding an approximation, and not necessarily find the better one: with a slightly more precise input, we can obtain a worse output.

Definition 11.9 – Chain

Let $(E, \sqsubseteq)$ be a poset. Let $A \subseteq E$. We say that A is a chain if $\min_{\sqsubseteq} A$ exists and $(A, \sqsubseteq)$ is totally ordered.

Definition 11.10 – Continuous maps

Let $\mathcal{E} = (E, \sqsubseteq)$ be a poset and $f : E \to E$. f is continuous if and only if

$$\forall \text{ chain } A \subseteq E, \begin{cases} \sqcup_{\mathcal{E}}\, f(A) \text{ exists whenever } \sqcup_{\mathcal{E}}\, A \text{ exists} \\ \text{and } f(\sqcup_{\mathcal{E}} A) = \sqcup_{\mathcal{E}} f(A) \end{cases}$$

Definition 11.11 – Contractive and extensive maps

Let $(E, \sqsubseteq)$ be a poset and $f : E \to E$.
f is contractive if

$$\forall x \in E, f(x) \sqsubseteq x$$

f is extensive if

$$\forall x \in E, x \sqsubseteq f(x)$$

Definition 11.12 – Fixpoint

Given $(E, \sqsubseteq)$ and a map $f : E \to E$. Let $x \in E$.

- If $f(x) = x$, then x is a fixpoint of f.
- If $x \sqsubseteq f(x)$, then x is a pre-fixpoint of f.
- If $f(x) \sqsubseteq x$, then x is a post-fixpoint of f.

Notation 11.7 – Fixpoint

Let $(E, \sqsubseteq)$ be a poset and $f : E \to E$, we denote $\text{fp}\, f$ the set of fixpoint of f, that is:

$$\text{fp}\, f := \{x \in E \mid f(x) = x\}$$

Let $x \in E$. We denote $\text{lfp}_x\, f$ the least fixpoint of f greater than x if it exists, that is:

$$\text{lfp}_x\, f := \min_{\sqsubseteq} \{y \in \text{fp}\, f \mid x \sqsubseteq y\}$$

The least fixpoint of f is denoted $\text{lfp}\, f$, that is:

$$\text{lfp}\, f := \text{lfp}_{\bot}\, f$$

Dually, we denote $\text{gfp}_x\, f$ the greatest fixpoint of f lower than x if it exsits, that is:

$$\text{gfp}_x\, f := \max_{\sqsubseteq} \{y \in \text{fp}\, f \mid y \sqsubseteq x\}$$

The greatest fixpoint of f is denoted $\text{gfp}\, f$, that is:

$$\text{gfp}\, f := \text{gfp}_{\top}\, f$$

Fixpoints are a central problem in analysis: they are useful in loops semantics. Nevertheless, we know that loop termination is undecidable, and equivalently, there is no automatic way to find a fixpoint in finite time. Thus, we shall focus on finding approximation of fixpoints.

Lemma 11.1

Let $\mathcal{E} = (E, \sqsubseteq)$ a complete lattice and $f : X \to X$ a monotonic map. Then, $\text{lfp}\, f$

and $\operatorname{gfp} f$ exist. Moreover,

$$\operatorname{lfp} f = \sqcap_{\mathcal{E}}\{x \in E \mid f(x) \sqsubseteq x\}$$
$$\operatorname{gfp} f = \sqcup_{\mathcal{E}}\{x \in E \mid x \sqsubseteq f(x)\}$$

Proof. Let $P := \{x \in E \mid f(x) \sqsubseteq x\}$ and $m := \sqcap_{\mathcal{E}} P$. We will prove that $m = \operatorname{lfp} f$.

$$\begin{array}{lll} & \forall x \in P,\quad m \sqsubseteq x & \wr \text{ by definition of } \sqcap_{\mathcal{E}} \ \wr \\ \Rightarrow & \forall x \in P,\quad f(m) \sqsubseteq f(x) & \wr \text{ since } f \text{ is monotonic } \wr \\ \Rightarrow & \forall x \in P,\quad f(m) \sqsubseteq f(x) \sqsubseteq x & \wr \text{ since } x \in P \ \wr \end{array}$$

Since $f(m)$ is lower than everything in P, by transitivity, $f(m) \sqsubseteq m$.

$$\begin{array}{lll} & f(m) \sqsubseteq m & \\ \Rightarrow & f(f(m)) \sqsubseteq f(m) & \wr \text{ since } f \text{ is monotonic } \wr \\ \Rightarrow & f(m) \in P & \wr \text{ by definition of } P \ \wr \\ \Rightarrow & m \sqsubseteq f(m) & \wr \text{ since } m \text{ is a lower bound of } P \ \wr \end{array}$$

By antisymmetry, we have $m = f(m)$, i.e. $m \in \operatorname{fp} f$.

Moreover, we have $\operatorname{fp} f \subseteq P$. Since m is the greatest lower bound of P and is a fixpoint, $m = \operatorname{lfp} f$.

Dually, we prove the existence of $\operatorname{gfp} f$. □

Theorem 11.1 – TARSKI's theorem

Let $\mathcal{E} = (E, \sqsubseteq)$ a complete lattice and $f : X \to X$ a monotonic map. $\operatorname{fp} f$ is a complete lattice.

Proof. Let $A \subseteq \operatorname{fp} f$, we prove that $\operatorname{lfp}_{\sqcup_{\mathcal{E}} A} f$ exists.

Consider $F := \{x \in E \mid \sqcup_{\mathcal{E}} A \sqsubseteq x\}$. $(F, \sqsubseteq)$ is a complete lattice and $f(F) \subseteq F$ since f is monotonic. We consider $g := f_{|F}$. Thanks to lemma 11.1, so g has a least fixpoint, and $\operatorname{lfp} g = \operatorname{lfp}_{\sqcup_{\mathcal{E}} A} f$.

$\operatorname{lfp}_{\sqcup_{\mathcal{E}} A} f$ exists and is smaller than any fixpoint that is an upper bound of F.

Dually, we can construct $\operatorname{gfp}_{\sqcap_{\mathcal{E}} A} f$.

Thus $(\operatorname{fp} f, \sqsubseteq)$ is a complete lattice. The lowest element is $\operatorname{lfp} f$ and the bigger is $\operatorname{gfp} f$. Least upper bound application is $X \in \mathcal{P}(\operatorname{fp} f) \mapsto \operatorname{lfp}_{\sqcup_{\mathcal{E}} X} f$ and greatest lower bound application is $X \subseteq \operatorname{fp} f \mapsto \operatorname{gfp}_{\sqcap_{\mathcal{E}} X} f$. □

11.2 Galois Connections Framework

Abstraction can be formalized in many frameworks; among them we highlight two: concretization-only or Galois connections. Both formalisms will be explained as they have their own advantages and drawbacks. Eventually, we will work rather with the concretization framework, but we explain Galois connections as it is a very popular formalism and it is interesting to understand its limitations. This section is dedicated to abstractions through Galois connections.

Definition 11.13 – Galois connection

A Galois connection is a tuple $((C, \leqslant), (A, \sqsubseteq), \gamma, \alpha)$ where:

- $(C, \leqslant)$ is a poset called concrete poset,
- $(A, \sqsubseteq)$ is a poset called abstract poset,
- $\gamma : A \to C$,
- $\alpha : C \to A$,

such that

$$\forall (a, c) \in A \times C, \alpha(c) \sqsubseteq a \iff c \leqslant \gamma(a)$$

Galois connections are defined using posets. Yet, many theorems need stronger structures, like lattices, CPOs (see definition 11.15 "CPO (complete partial order)" (page 150)) or complete lattices.

Notation 11.8 – Galois connection

A Galois connection $((C, \leqslant), (A, \sqsubseteq), \gamma, \alpha)$ is denoted

$$(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$$

A Galois connection is a tight correspondence between the abstract and the concrete world. It means that each concrete element c has a best abstraction $\alpha(c)$. In other words, $\alpha(c)$ is smaller than every other sound abstraction of c.

A Galois connection is a powerful tool, but it may not be the adequate framework. Indeed, many abstract domains do not define a Galois connection. While there are real-world such domains, it is easy to build one artificially: we just need to duplicate an arbitrary abstract element into a fresh (and not comparable) element with the same concretization. These two abstract elements are equally good abstractions of their concretization, thus there is no best abstraction anymore.

Proposition 11.10 – Duality of Galois connections

Let

$$\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$$

We have

$$\overline{\mathcal{A}} \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \overline{\mathcal{C}}$$

Proof. We want to prove that

$$\forall (c, a) \in C \times A, \gamma(a) \geqslant c \iff a \sqsupseteq \alpha(c)$$

Which is exactly the hypothesis $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ up to the symmetry of $\Leftrightarrow$, simply using the definition of $\geqslant$ (resp. $\sqsupseteq$) from $\leqslant$ (resp. $\sqsubseteq$). $\square$

Proposition 11.11 – Alternative definition of Galois connections

Let $(C, \leqslant)$ and $(A, \sqsubseteq)$ be posets, $\alpha : A \to C$ and $\gamma : C \to A$ be maps.
The four properties:

- α is monotone,
- γ is monotone,
- $\alpha \circ \gamma$ is contractive,
- $\gamma \circ \alpha$ is extensive,

are true if and only if

$$(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$$

Proof. $(\Rightarrow)$ We assume that α and γ are monotone, $\alpha \circ \gamma$ is contractive and $\gamma \circ \alpha$ is extensive. Let $c \in C$ and $a \in A$. We assume that $\alpha(c) \sqsubseteq a$.

$$\begin{aligned} \alpha(c) \sqsubseteq a &\Rightarrow \gamma(\alpha(c)) \leqslant \gamma(a) && \wr \text{ since } \gamma \text{ is monotone } \int \\ &\Rightarrow c \leqslant \gamma(a) && \wr \text{ since } \gamma \circ \alpha \text{ is extensive and } \leqslant \text{ transitive } \int \end{aligned}$$

Dually, we can prove than $c \leqslant \gamma(a) \Rightarrow \alpha(c) \sqsubseteq a$.

$(\Leftarrow)$ We assume that $(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$.
Let $c \in C$.

$$\begin{aligned} c = c &\Rightarrow \alpha(c) = \alpha(c) && \wr \text{ Leibniz's law } \int \\ &\Rightarrow \alpha(c) \sqsubseteq \alpha(c) && \wr \text{ Reflexivity of } \sqsubseteq \int \\ &\Rightarrow c \leqslant \gamma(\alpha(c)) && \wr \text{ Definition of Galois connection } \int \end{aligned}$$

Thus $\gamma \circ \alpha$ is extensive. Dually, we can prove that $\alpha \circ \gamma$ is contractive.
Let $(a, a') \in A^2$. We assume that $a \sqsubseteq a'$.

$$\begin{aligned} &\alpha \circ \gamma(a) \sqsubseteq a && \wr \text{ Contractivity of } \alpha \circ \gamma \int \\ &\Rightarrow \alpha \circ \gamma(a) \sqsubseteq a' && \wr \text{ Transitivity of } \sqsubseteq \int \\ &\Rightarrow \gamma(a) \leqslant \gamma(a') && \wr \text{ Definition of Galois connection } \int \end{aligned}$$

Thus γ is monotone. Dually, we can prove that α est monotone as well.

$\square$

Proposition 11.12 – Iteration of adjoints

Let $(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$. We have
- $\gamma \circ \alpha \circ \gamma = \gamma$
- $\alpha \circ \gamma \circ \alpha = \alpha$

Proof. We will only prove the first point. The second can be obtained by duality.

Let $c \in C$, $a = \alpha(c)$ and $c' = \gamma(a)$. By reflexivity of $\leqslant$, we have $c' \leqslant \gamma(a)$. By the definition of GALOIS connections, we have $\alpha(\gamma(\alpha(c))) = \alpha(c') \sqsubseteq a$.

Moreover, $\gamma \circ \alpha$ is extensive. Thus, $c \leqslant \gamma \circ \alpha(c)$. And since α is monotone, $\alpha(c) \sqsubseteq \alpha \circ \gamma \circ \alpha(c)$.

By antisymmetry, $\gamma \circ \alpha \circ \gamma = \gamma$. □

Definition 11.14 – Soundness

Given a GALOIS connection $(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, we say that $a \in A$ is a sound approximation of $c \in C$ if

$$c \leqslant \gamma(a)$$

Let $f : C \to C$ and $f^\sharp : A \to A$. We say that $f^\sharp$ is a sound approximation of f if

$$f \circ \gamma \leqslant \gamma \circ f^\sharp$$

More generally, let $n \in \mathbb{N}$. Let $f : C^n \to C$ and $f^\sharp : A^n \to A$. We say that $f^\sharp$ is a sound approximation of f if

$$\forall (a_1, \dots, a_n) \in A^n, f(\gamma(a_1), \dots, \gamma(a_n)) \leqslant \gamma \circ f^\sharp(a_1, \dots, a_n)$$

Here, we have tacitly lifted $\leqslant$ pointwise: $f \circ \gamma \leqslant \gamma \circ f^\sharp$ means that

$$\forall a \in A, f \circ \gamma(a) \leqslant \gamma \circ f^\sharp(a)$$

We will do so in the following.

Proposition 11.13 – Alternative definition of soundness

Given a GALOIS connection $(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, $f : C \to C$ and $f^\sharp : A \to A$. We have

$$f \circ \gamma \leqslant \gamma \circ f^\sharp \iff \alpha \circ f \circ \gamma \sqsubseteq f^\sharp$$

Proof.

($\Rightarrow$) We assume that $f \circ \gamma \leqslant \gamma \circ f^\sharp$. Let $a \in A$.

$$\begin{aligned} f \circ \gamma(a) \leqslant \gamma \circ f^\sharp(a) &\Rightarrow \alpha \circ f \circ \gamma(a) \sqsubseteq \alpha \circ \gamma \circ f^\sharp(a) && \{\text{since } \alpha \text{ is monotone}\} \\ &\Rightarrow \alpha \circ f \circ \gamma(a) \sqsubseteq f^\sharp(a) && \left\{\begin{array}{l}\text{since } \alpha \circ \gamma \text{ is contractive} \\ \text{and } \sqsubseteq \text{ transitive}\end{array}\right\} \end{aligned}$$

($\Leftarrow$) We assume that $\alpha \circ f \circ \gamma \sqsubseteq f^\sharp$. Let $a \in A$.

$$\begin{aligned} \alpha \circ f \circ \gamma(a) \sqsubseteq f^\sharp(a) &\Rightarrow \gamma \circ \alpha \circ f \circ \gamma(a) \leqslant \gamma \circ f^\sharp(a) && \wr \text{ since } \gamma \text{ is monotone } \wr \\ &\Rightarrow f \circ \gamma(a) \leqslant \gamma \circ f^\sharp(a) && \left\wr \begin{array}{l} \text{since } \gamma \circ \alpha \text{ is extensive} \\ \text{and } \leqslant \text{ transitive} \end{array} \right\wr \end{aligned}$$

$\square$

This introduction may justly seem a bit arid and inelegant. The goal is to quickly come to the requirement of an abstraction to be able to state the interests and drawbacks of this framework.

To build an abstraction, we would like a way to abstract soundly loops that are defined by a least fixpoint. For this purpose, there are various fixpoint transfer or fixpoint approximation theorems.

To actually build an analyzer, we expect the abstract semantics to be inductively defined. In particular, if we denote $[\![S]\!]$ the semantics of a statement S, and $[\![S]\!]^\sharp$ its abstract semantics, as we have $[\![S_0; S_1]\!] = [\![S_1]\!] \circ [\![S_0]\!]$, we need to have $[\![S_0; S_1]\!]^\sharp = [\![S_1]\!]^\sharp \circ [\![S_0]\!]^\sharp$.

The first theorem is very strong: under constraining assumptions, the least fixpoint of the abstract transfer function is the abstraction of the least fixpoint of the concrete transfer function. This is much stronger than stating it is a sound approximation.

Theorem 11.2 – Exact fixpoint transfer

Let $(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ be a Galois connection between complete lattices. Moreover, let $f : C \to C$, $f^\sharp : A \to A$, $c \in C$ and $a \in A$ such that

- f is continuous
- $f^\sharp$ is monotone
- $\alpha \circ f = f^\sharp \circ \alpha$
- $\alpha(c) = a$

then

$$\alpha(\mathrm{lfp}_c\, f) = \mathrm{lfp}_a\, f^\sharp$$

We will not prove this theorem since we will not use it: its assumptions are not realistic. The requirement of $f^\sharp$ being monotone is generally not matched, especially when abstracting loops. Moreover, the requirement $\alpha \circ f = f^\sharp \circ \alpha$ is also utopian: asking $f^\sharp$ to preserve best abstractions is not realistic for expressive abstractions.

Even if we relax this last point, the resulting theorem is not so usable.

Theorem 11.3 – Approximate fixpoint transfer

Let $(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ be a Galois connection between complete lattices. Moreover, let $f : C \to C$, $f^\sharp : A \to A$, $c \in C$ and $a \in A$ such that

- f is continuous
- $f^\sharp$ is monotone

– $\alpha \circ f \sqsubseteq f^\sharp \circ \alpha$
– $\alpha(c) = a$

then

$$\alpha(\mathrm{lfp}_c\, f) \sqsubseteq \mathrm{lfp}_a\, f^\sharp$$

Even the requirement $\alpha \circ f \sqsubseteq f^\sharp \circ \alpha$ is too strong in practice. Assuming we have $\alpha \circ [\![S_0]\!] \sqsubseteq [\![S_0]\!]^\sharp \circ \alpha$ and $\alpha \circ [\![S_1]\!] \sqsubseteq [\![S_1]\!]^\sharp \circ \alpha$.

$$\begin{aligned} \alpha \circ [\![S_1]\!] \circ [\![S_0]\!] &\sqsubseteq [\![S_1]\!]^\sharp \circ \alpha \circ [\![S_0]\!] && \wr \text{ by hypothesis } \int \\ &\sqsubseteq [\![S_1]\!]^\sharp \circ [\![S_0]\!]^\sharp \circ \alpha && \wr \text{ by hypothesis and if } [\![S_1]\!]^\sharp \text{ is monotone } \int \end{aligned}$$

Thus, for the abstraction to be compositional, we need all the elementary abstract transfer functions to be monotone, and not just the compound transfer functions of loop bodies.

To summarize, the Galois connection framework requires to have a best abstraction map. Even with realistic abstractions, the best abstraction may not exist. Moreover, the approximate fixpoint transfer theorem needs the monotony of all abstract transfer functions. On the plus side, the characterization of soundness using α ($\alpha \circ f \circ \gamma \sqsubseteq f^\sharp$) defines a lower bound on $f^\sharp$. Thus, the best $f^\sharp$ possible is this lower bound. We get a generic way to derive the best abstract transfer function from the concrete transfer function and the abstraction relation.

11.3 Concretization Framework

A related approach is to only use a concretization map. In this case, the concretization explains the concrete meaning of an abstract element. By dropping the abstraction map, we only have a way to check whether an abstraction is sound, but no generic way to deduce the abstract transfer function.

In this framework, we have an abstract complete lattice $(A, \sqsubseteq)$ and a concrete complete lattice $(C, \leqslant)$. We also have a monotone map $\gamma : A \to C$ to interpret abstract elements. The notion of soundness of definition 11.14 “Soundness” (page 147) still stands. Galois connections enforce a relation between the order of abstract and concrete lattices. Of course, even without abstraction function, we need to have some constraints as well, in particular: $\sqcap$ is a sound approximation of $\wedge$.

Fixpoint transfer theorems (theorem 11.2 “Exact fixpoint transfer” (page 148) and theorem 11.3 “Approximate fixpoint transfer” (page 148)) are not applicable here, we need another one.

Theorem 11.4 – Fixpoint approximation

Let $(A, \sqsubseteq)$ and $(C, \leqslant)$ be complete lattices, and $\gamma : A \to C$ a monotone map. Let $f : C \to C$ a monotone map, $f^\sharp : A \to A$ and $a \in A$. If
– $f \circ \gamma \leqslant \gamma \circ f^\sharp$: $f^\sharp$ is a sound approximation f,

– $f^\sharp(a) \sqsubseteq a$: a is a post-fixpoint of $f^\sharp$,

then a is a sound approximation of lfp f, i.e.

$$\text{lfp } f \leqslant \gamma(a)$$

Proof. We assumed, $f^\sharp(a) \sqsubseteq a$.

$$\begin{array}{rcll} f^\sharp(a) \sqsubseteq a & \Rightarrow & \gamma(f^\sharp(a)) \leqslant \gamma(a) & \wr \text{ by monotony of } \gamma \,\wr \\ & \Rightarrow & f(\gamma(a)) \leqslant \gamma(a) & \left\wr \begin{array}{l}\text{since } f^\sharp \text{ is a sound approximation} \\ \text{of } f \text{ and by transitivity of } \leqslant\end{array} \right\wr \end{array}$$

The proof of the lemma of Tarski's theorem (lemma 11.1) states that

$$\text{lfp } f = \bigwedge \{x \in C \mid f(x) \leqslant x\}$$

Thus

$$\text{lfp } f \leqslant \gamma(a)$$

□

Of course, this theorem is still valid in the context of Galois connections, but this context introduces more hypotheses (and thus, constraints) that are useless in this case.

We can note that the monotony requirement on the concrete transfer function is not prohibitive: usually, concrete transfer functions map set of states to set of states, using a bigger set of preconditions may only returns a bigger set of postconditions.

We can also check that the soundness condition is compositional. If we have $[\![S_0]\!] \circ \gamma \leqslant \gamma \circ [\![S_0]\!]^\sharp$ and $[\![S_1]\!] \circ \gamma \leqslant \gamma \circ [\![S_1]\!]^\sharp$.

$$\begin{array}{rcll} [\![S_1]\!] \circ [\![S_0]\!] \circ \gamma & \leqslant & [\![S_1]\!] \circ \gamma \circ [\![S_0]\!]^\sharp & \left\wr \begin{array}{l}\text{by soundness of } [\![S_0]\!]^\sharp \\ \text{and monotony of } [\![S_1]\!]\end{array} \right\wr \\ & \leqslant & \gamma \circ [\![S_1]\!]^\sharp \circ [\![S_0]\!]^\sharp & \wr \text{ by soundness of } [\![S_1]\!]^\sharp \,\wr \end{array}$$

There is no monotony requirements on abstract transfer functions anymore.

Here, the difficulty is to get a post-fixpoint of $f^\sharp$ in a constructive and automatic way.

Definition 11.15 – CPO (complete partial order)

A poset $\mathcal{E} = (E, \sqsubseteq)$ is a CPO (complete partial order) if

$$\forall \text{ chain } C \subseteq E, \sqcup_{\mathcal{E}} C \text{ exists}$$

Proposition 11.14

Every complete lattice is a CPO.

Proof. In a complete lattice $\mathcal{E} = (E, \sqsubseteq)$, for all $X \subseteq E$, $\sqcup_{\mathcal{E}} X$ exists. Thus, this is true for chains, in particular. □

Theorem 11.5 – KLEENE fixpoint theorem

Let $\mathcal{E} = (E, \sqsubseteq)$ be a CPO, $f : E \to E$ and $a \in E$. If $a \sqsubseteq f(a)$ and f is continuous, then $\operatorname{lfp}_a f$ exists and

$$\operatorname{lfp}_a f = \sqcup_{\mathcal{E}}\{f^n(a) \mid n \in \mathbb{N}\}$$

where f^n is the n^{th} iterate of f.

Proof. By hypothesis $a \sqsubseteq f(a)$. Since f is monotone, it preserves the order, thus

$$\forall n \in \mathbb{N}, f^n(a) \sqsubseteq f^{n+1}(a)$$

hence

$$\forall (m,n) \in \mathbb{N}^2, m \leqslant n \Rightarrow f^m(a) \sqsubseteq f^n(a)$$

Thus, $I := \{f^n(a) \mid n \in \mathbb{N}\}$ is a chain and, since E is a CPO, $\sqcup_{\mathcal{E}} I$ exists.

$$\begin{array}{rcll} f(\sqcup_{\mathcal{E}} I) & = & \sqcup_{\mathcal{E}}\left\{f^{n+1}(a) \,\middle|\, n \in \mathbb{N}\right\} & \wr \text{ since } f \text{ is continuous } \int \\ & = & \sqcup_{\mathcal{E}}\left\{\sqcup_{\mathcal{E}}\left\{f^{n+1}(a) \,\middle|\, n \in \mathbb{N}\right\}, a\right\} & \wr \text{ since } \forall n \in \mathbb{N}, a \leqslant f^{n+1}(a) \int \\ & = & \sqcup_{\mathcal{E}} I & \end{array}$$

So $\sqcup_{\mathcal{E}} I \in \operatorname{fp} f$.

Moreover, $\forall x \in \operatorname{fp} f, a \sqsubseteq x \Rightarrow (\forall n \in \mathbb{N}, f^n(a) \sqsubseteq x)$. So $\sqcup_{\mathcal{E}} I = \operatorname{lfp}_a f$ □

This theorem has a stronger hypothesis than TARSKI's, but it provides a constructive way to get the fixpoint. Sadly, this requires countable iteration (so infinite in general). But, we can get a sound approximation of the limit with finite iteration. For that, we need an over-approximation of the union that guarantees convergence in finite number of steps.

Definition 11.16 – Widening

Let $(C, \leqslant)$ be a concrete complete lattice, $(A, \sqsubseteq)$ an abstract poset and $\gamma : A \to C$ the concretization map. A widening operator over A is a binary function $\nabla : A^2 \to A$ such that:

- $\forall (x,y) \in A^2, \gamma(x) \vee \gamma(y) \leqslant \gamma(x \nabla y)$ (over-approximation of $\vee$)
- for all sequence $(a_i)_{i\in\mathbb{N}} \in A^{\mathbb{N}}$ of abstract elements, the sequence $(b_i)_{i\in\mathbb{N}} \in A^{\mathbb{N}}$ defined by

$$b_i := \begin{cases} a_0 & \text{if } i = 0 \\ b_{i-1} \nabla a_i & \text{otherwise} \end{cases}$$

  is stationary.

In fact, in general, a widening works on sequence prefixes and previous result, but in practice, they just work with the last element of each prefix, making the widening a binary function.

Theorem 11.6 – Widening-based fixpoint approximation

Let $(C, \leqslant)$ be a concrete complete lattice, $(A, \sqsubseteq)$ an abstract poset, $\gamma : A \to C$ the concretization map and ∇ a widening operator on A. Let $f : C \to C$ be a monotone map and $f^\sharp : A \to A$ be a sound approximation of C.
We define the sequence $(a_i)_{i \in \mathbb{N}} \in A^{\mathbb{N}}$ by

$$a_i := \begin{cases} \bot & \text{if } i = 0 \\ a_{i-1} \nabla f^\sharp(a_{i-1}) & \text{otherwise} \end{cases}$$

then

- a is stationary, and we denote l its limit,
- $\operatorname{lfp} f \leqslant \gamma(l)$.

Proof. From the definition of widening, it is very direct that a is stationary. We let $N \in \mathbb{N}$ such that $\forall n \in \mathbb{N}, n \geqslant N \Rightarrow a_n = l$.

We have

$$\gamma(l) \leqslant \gamma(l) \vee \gamma\left(f^\sharp(l)\right)$$

by definition of $\vee$. Since the widening is a sound approximation of $\vee$, we have

$$\gamma(l) \vee \gamma\left(f^\sharp(l)\right) \leqslant \gamma\left(l \nabla f^\sharp(l)\right)$$

And since l is the limit of a, we have $l = l \nabla f^\sharp(l)$, thus, $\gamma(l) = \gamma\left(l \nabla f^\sharp(l)\right)$. By transitivity of $\leqslant$:

$$\gamma(l) \leqslant \gamma(l) \vee \gamma\left(f^\sharp(l)\right) \leqslant \gamma\left(l \nabla f^\sharp(l)\right) = \gamma(l)$$

and by antisymmetry $\gamma(l) = \gamma(l) \vee \gamma\left(f^\sharp(l)\right)$, thus $\gamma\left(f^\sharp(l)\right) \leqslant \gamma(l)$.

By soundness of $f^\sharp$, we have

$$f(\gamma(l)) \leqslant \gamma\left(f^\sharp(l)\right)$$

and by transitivity

$$f(\gamma(l)) \leqslant \gamma(l)$$

that is $\gamma(l)$ is a post-fixpoint of f.

Thanks to Tarski's theorem's lemma, we know that f indeed have a least fixpoint, and

$$\operatorname{lfp} f = \bigwedge \{x \in C \mid f(x) \leqslant x\}$$

Thus,

$$\operatorname{lfp} f \leqslant \gamma(l)$$

□

Here, we used a very permissive definition of the widening: it is a sound approximation of the concrete $\vee$. The drawback is that l is not necessarily an abstract post-fixpoint. We may strengthen the definition of the widening by replacing the condition

$$\forall(x, y) \in A^2, \gamma(x) \vee \gamma(y) \leqslant \gamma(x \nabla y)$$

by

$$\forall(x, y) \in A^2, x \sqcup y \sqsubseteq x \nabla y$$

which implies that we have an abstract lattice (and not simply a poset). Since $\sqcup$ is a sound approximation of $\vee$, by transitivity, this version implies the definition 11.16. Moreover, we can prove directly that l is an abstract post-fixpoint, and we obtain the result by using the theorem 11.4 "Fixpoint approximation" (page 149).

We can remark that the proof of theorem 11.4 uses the fact that l is an abstract post-fixpoint only to prove that $\gamma(l)$ is a concrete post-fixpoint; so it may look dubious to use a stronger hypothesis to make the same proof, with additional steps. In the context we presented here, having an abstract post-fixpoint is indeed not very interesting, but it is if we want to check the result of an analysis: we cannot do much from the only knowledge that $\gamma(l)$ is a concrete post-fixpoint, but we can check that l is an abstract post-fixpoint without knowing anything about the concrete world, or the widening.

To ensure termination, widening may lead to a loss of precision. To refine our approximation, there is a similar notion that is an over-approximation of the intersection and enforce convergence in finite time: narrowing operators. It is good to emphasize that narrowing operators are not dual widening: a widening must go past the least fix point, while a narrowing must stay above. A dual widening would go under the greatest fixpoint. Narrowing will not be detailed much more, as they are not relevant for the following.

Of course, being able to perform sound approximations of union, intersection and least fixpoint is mandatory, but the semantics contains probably more than that. Commonly, the semantics is expressed by composition of such operators and elementary transfer functions, e.g. assigning a variable, of enforcing a condition. For each transfer function, we need a corresponding sound approximation. The set of all these approximate transformers, with the explicative concretization, forms an abstract domain. The concretization is useful to justify the soundness of the analysis, but it does not appear in the implementation of an abstract domain.

Chapter 12

Product of Abstract Domains

Abstract domains only keep partial information about the state of the program. Necessarily, the imprecise knowledge of the state may be too coarse to be able to check the interest properties.

Imprecision has two main sources. Firstly, abstract domains may be not as good as they can be. There are two possible reasons: abstract transfer functions may not be as precise as possible, or widening may be too brutal: by enforcing a very quick termination, we may have a very imprecise approximation of the least fixpoint. These are two kinds of loss of precision, but they can be solved by improving the domain, without changing the kind of property it handles. Depending on the case, we can improve abstract transfer functions, or use a more cautious widening whose termination will be slower, but approximation better. This kind of imprecision is common: usually, we do not implement abstract transfer functions optimally, but only well enough for existing cases. Likewise, widening is always heuristic, and depends on the application. Fortunately, improving a domain is usually easy. But sometimes, this is not enough: the domain is simply not suited to the problem.

The other kind of imprecision is more fundamental: an abstract domain is bound to a certain kind of properties. One may remember the range of variables or parity of variables, for instance. An abstract domain may also handle linear relations between variables, or modular equalities. We can abstract any kind of properties but, *a priori*, each domain takes care of one kind of property.

```c
void f() {
    unsigned int x = random_between(0, 10);
    unsigned int y = 42;
    unsigned int z = y + x;
    z = z - x;
    assert(z == 42);
}
```

Listing 12.1: When merging two instructions can improve precision

When precision is lost between two statements, we may either consider more expressive properties that allows domains to keep their precision, or smash the two statements in a single call to a complex abstract transfer function. Indeed, the abstraction of the sequence of two statements can be more precise than the composition of the abstraction of each statement. For instance, in Listing 12.1, if we can only remember an interval of possible values for each variable, after line 4, `z` is in the interval $[42, 52]$. After line 5, we would get the interval $[32, 52]$, thus we do not manage to prove the assertion. If we merge statements of lines 4 and 5, we can simplify them to `z = y`, which make the assertion easily provable. This method can work, but the number of transfer functions would explode, and the adequate grouping of statements may depend on domains. Thus, we prefer to handle more expressive properties, while keeping a constant instruction-wise granularity.

If the imprecision must be solved using a property that is not representable in the abstract domain we use, we need a new adequate domain for this kind of properties. In the example of Listing 12.1, we need to remember the linear equality $z = y + x$. Just adding an abstract domain do not solve this issue: they must cooperate. If they do not, from the point of view of the other domains, nothing changed and the same imprecision will happen. This chapter is about how we can make domains to cooperate. One can find further references in [CC79; CCF13].

12.1 Example: When Reduction is Needed

```
void f() {
    int v = 1;
    while(v <= 10) {
        v = v + 2;
    }
    if(v >= 12) {
        // error
    }
}
```

Listing 12.2: A program that requires two domains

We consider the program on Listing 12.2. We assume we have a very common domain: the interval domain. This domain represents the value of each variable by an interval of possible values. We look at the values of `v` at the top of the loop (before testing the condition). At the first iteration, it is $[\![1, 1]\!]$. Let us run the loop using only the union (without widening) to get most precision. We can do that here since it will stabilize in finite time. Another approach is to use a widening that will perform n unions at the beginning before returning $\mathbb{Z}$, and we choose n big enough.

After one iteration, `v` has value 3; the value at the top of the loop is now $[\![1, 3]\!]$. All values in the interval are lower than 10 thus everything enters the loop. After another iteration, we get $[\![1,5]\!]$, and so on. After 5 iterations, we have $[\![1,11]\!]$. Something is different here: only $[\![1,10]\!]$ enters the loop again, and $[\![11,11]\!]$ exits the loop. Since the previous state was $[\![1,9]\!]$, we do not have a fixpoint yet: 10 is a new possible value. It is interesting to remark that 10 is not possible in the concrete execution since v is always odd. We perform another iteration, and we end with $[\![1,12]\!]$ at the top of the loop. Here $[\![1,10]\!]$ enters again and $[\![11,12]\!]$ exits. We reached a fixpoint. The Table 12.1 sums up the abstract properties at each iteration. In green, we show the inductive invariant that holds at the beginning of loop's body.

Number of iterations	0	1	2	3	4	5	6
Interval at the top of the loop	$[\![1,1]\!]$	$[\![1,3]\!]$	$[\![1,5]\!]$	$[\![1,7]\!]$	$[\![1,9]\!]$	$[\![1,11]\!]$	$[\![1,12]\!]$
Interval staying in the loop	$[\![1,1]\!]$	$[\![1,3]\!]$	$[\![1,5]\!]$	$[\![1,7]\!]$	$[\![1,9]\!]$	$[\![1,10]\!]$	$[\![1,10]\!]$
Interval exiting the loop	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$[\![11,11]\!]$	$[\![11,12]\!]$

Table 12.1: Abstract properties at each iteration

Before the line 6, possible values, according to interval domain are $[\![11,12]\!]$. Thus, it seems possible to enters the body of the if-statement, which is an error. However, since `v` is always odd, the value at the end of the loop is in fact only $[\![11,11]\!]$, but interval domain is unable to handle holes in intervals.

We need another domain: parity domain. This domain remembers whether variables are even, odd, or might be both. Let us run the example with this domain. Initially, `v` is odd. In the loop, `v` stays odd: fixpoint is reached in only 1 iteration. Before the line 6, `v` can only be odd. This analysis still detect wrongly the error since some odd integers are greater than 12.

If we make an analysis using both domains, before the line 6, we know that $\mathtt{v} \in [\![11,12]\!] \wedge \mathtt{v} \equiv 1\,[2]$. Without cooperation, we would make the same mistake as before, and we would get $\mathtt{v} \in [\![12,12]\!] \wedge \mathtt{v} \equiv 1\,[2]$ inside the body of the if. To improve the precision and avoid the error, we need to be able to use each abstract property to refine the other. In this case, we can see that $\mathtt{v} \in [\![11,11]\!] \wedge \mathtt{v} \equiv 1\,[2]$ has exactly the same concretization as $\mathtt{v} \in [\![11,12]\!] \wedge \mathtt{v} \equiv 1\,[2]$. But this improved version will not enter the if-statement. Similarly, we could refine $\mathtt{v} \in [\![12,12]\!] \wedge \mathtt{v} \equiv 1\,[2]$ into $v \in \varnothing \wedge v \in \bot_{\text{parity}}$ since both have an empty concretization.

In this case, we could argue that a first parity analysis would be enough to know that `v` is odd, and it could feed the interval analysis with this property to get a precise analysis. But this method does not work if the parity domain needs information from the parity domain. For instance, if we replace the constant 2 by the expression `rand(10)/1000 + 2` (which is equal to 2), the parity domain will not be able to understand by its own that `v` is odd: it needs the interval domain to simplify `rand(10)/1000` into `0`. The ill-willer could argue that this problem can be solved by performing alternating interval and parity analysis, but that would just give extra work to communicate information between domains.

12.2 Reduced Product

Before improving abstract states, we have to make composite domains from two individual domains.

Proposition 12.1 – Product of complete lattices

Let $(E_1, \sqsubseteq_1)$ and $(E_2, \sqsubseteq_2)$ be complete lattices. Let $E := E_1 \times E_2$ and $\sqsubseteq$ be the relation on E defined by

$$\forall((x_1, x_2), (y_1, y_2)) \in E^2, (x_1, x_2) \sqsubseteq (y_1, y_2) :\Leftrightarrow x_1 \sqsubseteq_1 y_1 \wedge x_2 \sqsubseteq_2 y_2$$

$(E, \sqsubseteq)$ is a complete lattice.

Proof. We can indeed check the minimum in E is $(\bot_1, \bot_2)$ and the maximum is $(\top_1, \top_2)$. Similarly, least upper bound and greatest lower bounds are applied pointwise: let $A \subseteq E$

$$\sqcup A := (\sqcup_1\{x \in E_1 \mid (x, \_) \in A\}, \sqcup_2\{y \in E_2 \mid (\_, y) \in A\})$$
$$\sqcap A := (\sqcap_1\{x \in E_1 \mid (x, \_) \in A\}, \sqcap_2\{y \in E_2 \mid (\_, y) \in A\})$$

□

A similar result holds for Galois connections.

Proposition 12.2 – Product of Galois connections

Let $(C, \leqslant) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} (A_1, \sqsubseteq_1)$ and $(C, \leqslant) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} (A_2, \sqsubseteq_2)$.
Let $A := A_1 \times A_2$, $\gamma := (x, y) \in A \mapsto \gamma_1(x) \wedge \gamma_2(y)$, $\alpha := c \in C \mapsto (\alpha_1(c), \alpha_2(c))$ and $\sqsubseteq$ be the relation on A defined by

$$\forall((x_1, x_2), (y_1, y_2)) \in A^2, (x_1, x_2) \sqsubseteq (y_1, y_2) :\Leftrightarrow x_1 \sqsubseteq_1 y_1 \wedge x_2 \sqsubseteq_2 y_2$$

We have $(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$.

Proof. Obviously, α and γ are monotone.

Since α is monotone, and $\alpha_1 \circ \gamma$ and $\alpha_1 \circ \gamma$ are contractive, $\alpha \circ \gamma$ is contractive as well.

Similarly, $\gamma \circ \alpha$ is extensive.

Using the alternate definition of Galois connections (proposition 11.11 "Alternative definition of Galois connections" (page 146)), we have the result. □

It is interesting to remark that the concretization is the intersection of both concretization maps. Indeed, since both domains are sound, the concrete state is lower than the concretization of both abstract states, thus, it is lower than the intersection. We can also understand it in terms of properties: if two properties hold, the conjunction is true as well.

This gives a way to improve the abstract states using each other. In the context of the previous proposition, let us define

$$\begin{aligned}&\rho : A \to A\\&(x, y) \mapsto (\alpha_1(\gamma_1(x) \wedge \gamma_2(y)), \alpha_2(\gamma_1(x) \wedge \gamma_2(y)))\end{aligned}$$

that is, $\rho = \alpha \circ \gamma$. The concretization allows each domain to contribute to the overall precision, and by abstracting again, we gain a more precise abstract state. We indeed know that $\alpha \circ \gamma$ is contractive.

This is the best possible improvement. Moreover, we can easily check that it is still sound, i.e.:

$$\forall a \in A, \gamma(a) \leqslant \gamma \circ \rho(a)$$

We could also say that ρ is a sound approximation of the identity. Indeed, since $(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, we have

$$\begin{array}{rcll}\gamma \circ \rho(a) & = & \gamma \circ \alpha \circ \gamma(a) & \wr \text{ by definition of } \rho \text{ } \wr \\ & = & \gamma(a) & \left\{\begin{array}{l}\text{by proposition 11.12 “Itera-}\\ \text{tion of adjoints” (page 147)}\end{array}\right\}\end{array}$$

Which is even stronger, but it is not the point here.

However, this reduction operator is not usable in practice. First, we need to use the Galois connection framework. We explained in section 11.2 “Galois Connections Framework” (page 145) why this framework is usually too strong to be usable in practice, in particular, the abstraction map does not always exist. Moreover, even with an abstraction map, the concretization is often very expensive to compute in general, if it terminates. As a consequence, this direct definition is not usable in an implementation. An alternative approach is not to go by the concrete world but compute, by hand, the reduction operator, thanks to this definition, and implement the resulting operator. This method may seem appealing but fails in general: the reduction operator may still be very expensive to compute, and we need to compute it for all combinations of abstract domains. Moreover, we still need the theoretical framework of Galois connections.

We need another way to perform reduction. It might be an approximate reduction but it has to be efficiently computable and implementable modularly.

12.3 Partially Reduced product

Usually, fully reduced product is not necessary, but we need an effective reduction method and, to be able to implement it in practice, we need it to be modular: adding a domain should not require to update all existing domains.

Definition 12.1 – Sound partial reduction

Let D be the concrete domain and $D_1^\sharp$ and $D_2^\sharp$ be abstract lattices with respective concretization $\gamma_1 : D_1^\sharp \to D$ and $\gamma_2 : D_2^\sharp \to D$.
$\rho : D_1^\sharp \times D_2^\sharp \to D_1^\sharp \times D_2^\sharp$ is a sound reduction operator if

$$\forall (a_1, a_2) \in D_1^\sharp \times D_2^\sharp, \textbf{let } (b_1, b_2) := \rho(a_1, a_2) \textbf{ in}$$
$$\gamma_1(a_1) \cap \gamma_2(a_2) \subseteq \gamma_1(b_1) \cap \gamma_2(b_2)$$

Morally, we want to improve a_1 and a_2, so in most cases, $b_1 \sqsubseteq_1 a_1$ and $b_2 \sqsubseteq_2 a_2$. But this is not a requirement and, indeed, it is not always verified in real-world domains.

The solution to make a real-world reduction operator we will describe here is explained in detail in [Cou+06b]. The point is to use an abstract lattice $IO^\sharp$, called communication channel, common to all domains, that allows them to communicate. We are also given a concretization $\gamma_{IO^\sharp} : IO^\sharp \to D$ where D is the concrete domain.

Each abstract domain $D^\sharp$ (with concretization γ) features two additional functions:

$$\textsc{Extract} : D^\sharp \times IO^\sharp \to IO^\sharp$$
$$\textsc{Refine} : D^\sharp \times IO^\sharp \to D^\sharp$$

Of course, we need soundness requirements:

- $\forall a \in D^\sharp, \forall io \in IO^\sharp, \gamma(a) \cap \gamma_{IO^\sharp}(io) \subseteq \gamma_{IO^\sharp}(\textsc{Extract}(a, io))$
- $\forall a \in D^\sharp, \forall io \in IO^\sharp, \gamma(a) \cap \gamma_{IO^\sharp}(io) \subseteq \gamma(\textsc{Refine}(a, io))$

EXTRACT function enriches the content of a communication channel given an abstract element. REFINE uses the content of a communication channel to improve an abstract element.

The communication channel must be able to represent all properties that are inferred in a domain but used in another one. It is a conjunction of a lot of properties, but there is no transfer function in the communication channel. Adding a new kind of property does not imply more work, especially, all EXTRACT and REFINE functions are unchanged: they respectively never emit such property and ignore it. We just need to update the EXTRACT function of the domain that deduces the interesting property and the REFINE function of the domain that uses it.

The way we achieve that is merely technical and very dependent on the language we are using. We will simply detail how it is done in ASTRÉE as an example to demonstrate that this is perfectly feasible.

In ASTRÉE, we use in fact two communication channels according to the kind of communication we need. This implies that each domain implements two versions of EXTRACT and REFINE.

ASTRÉE uses a product of a lot of domains; the product orders them, and their abstract transfer functions are evaluated in this order. The first channel follows this evaluation order, it is called input channel. Each domain receives the state of the communication channel corresponding to the precondition and the postcondition, and can refine the postcondition. The constraints on the precondition come from the postcondition of the previous statement and the postcondition comes from $\top_{IO^\sharp}$ that has been refined by domains that have been run already. With this communication channel, information only flows from left to right, as domains are used. Once all domains have been run, the postcondition is refined by all domains and used as the precondition for the next statement.

Input channel is implemented as a product type, each field storing some kind of property. Domains read only fields they can use and update fields they can refine. If we need to add a new field, we just have to update the type definition and the initial value $\top_{IO^\sharp}$. Domains, by default, will simply ignore this field and never refine it, which is safe. To implement a reduction, we need to make the sender domain to refine this new field, and the receiver domain to read it and refine its own internal state accordingly. In this kind of communication, each domain communicates information without knowing if it will be used. It may seem uselessly costly, but we will see in subsection 13.2.2 “The Tree of Relational Numerical Domains” (page 167) that memoization and laziness make unused communication almost free.

The other kind of channel is more active: domains are allowed to emit some constraints they judge to be especially interesting. This channel, called output channel, allows two kinds of communication: to all domains (broadcast channel) or only to underlying domains (domains on the left, those that have been run before). Each concerned domain refines its internal state using the constraint. This communication channel allows information to be sent back to first (leftmost) domains.

Output channel is implemented using a sum type, each constructor representing some kind of property. Domains emit only constructors corresponding to properties they can deduce, which may be none: domains are not required to emit anything in the output channel. However, domains must be able to refine their internal state using such constraint. Unknown (or irrelevant) properties are ignored, which is safe. To enrich the output channel, we just need to update the type definition: new constructors are ignored by default by a catch-all pattern. To add a new reduction, once again, we just need to update the sender and the receiver.

This way of building reduction operators is easily implementable in practice, can be made efficient and precise enough. This is a very parametric framework. It has another benefit, but theoretical: just as implementation is distributed across domains, soundness proofs are also distributed. Indeed, we simply need to prove the soundness of EXTRACT and REFINE functions of each domain to get the soundness of reduction operator.

Yet, we must take care of termination: enforcing a constraint may generate new constraints, causing new reductions and so on. We cannot simply iterate reduction naively. Simple strategies solve this issue: like setting a maximum number of iterations or performing reduction only in the directions allowed by a directed acyclic graph.

Chapter 13

Astrée

MOST OF THE WORK presented in following parts has been implemented in a development version of ASTRÉE. ASTRÉE [Cou+01; Bla+03] is a static analyzer designed to analyze critical C source code coming from automotive, avionics, astronautics, etc.

ASTRÉE stands for “Analyseur statique de logiciels temps-réel embarqués” (real-time embedded software static analyzer). The development of ASTRÉE started in November 2001 and is still going. Due to its applications, ASTRÉE has some limitations like a limited support of dynamic allocations or the absence of support of variable-length arrays (VLAs). ASTRÉE is based on abstract interpretation and is sound: it finds all possible errors. It has some industrial successes like the proof of absence of RTEs in the primary flight control software of Airbus A340 and A380 fly-by-wire system and in the automatic docking software of the ATV[*], the cargo spacecraft developed by the ESA[†] to resupply the ISS[‡] (see [Cou+20]).

ASTRÉE has been extended in ASTRÉEA [Cou+06a] (which stands for “Analyseur statique de logiciels temps-réel asynchrones embarqués” (real-time asynchronous embedded software static analyzer)). The main improvement of ASTRÉEA is the capacity to analyze multi-threaded software. The name ASTRÉE is both used for the historic version and the family, in particular, nowadays ASTRÉE usually stands for ASTRÉEA, which is the last released major version. Marketed by AbsInt[§], ASTRÉE has now many industrial applications.

The changes explained in the following parts are implemented in a development version of ASTRÉE, called ASTRÉES (ASTRÉE for Security). ASTRÉES is developed on top of ASTRÉEA and may become the next major version of ASTRÉE family.

In the following, we will detail some implementation aspects of the work presented hereafter. To introduce the context required for these remarks, in this chapter, we present the architecture and some implementation details of ASTRÉE.

[*]Automated Transfer Vehicle
[†]European Space Agency
[‡]International Space Station
[§]AbsInt Angewandte Informatik GmbH, `https://www.absint.com/`

13.1 Overview of the Architecture

ASTRÉE is written mostly in OCAML [Ler+20], with a few parts in C. ASTRÉE makes an intensive usage of OCAML modules and functors. First, let us describe the global architecture of ASTRÉE.

ASTRÉE includes its own C parser that is able to parse custom analysis directives that are included in source code thanks to specific syntax extensions. Parser represents C programs in an abstract syntax tree (AST) of a format called `C`. This format is very syntactic: typing is not checked yet, redundant constructions are neither simplified nor unified, even parenthesis in arithmetic expressions are still mentioned in this AST.

The `C` tree is not directly usable. It is translated to another AST called `Cc` (or `Cc_code`). This tree is much more elaborated: it contains types at all expression nodes and operators. Variables and functions are identified by integer unique identifiers (UID) to solve explicitly scoping. Some syntactic sugar is removed, but we keep as much syntactic information as possible[¶]. For instance, `Cc` have only one node for loops, but with optional fields to represent while-loops as well as for-loops without transforming them into a while-loop. This is because syntax carries some good hints about the semantics that can be very useful for heuristics. For instance, in a for-loop, it is usually very easy to find the variable that acts as loop index and adapt the abstraction accordingly. It would be clumsy to compile for-loops to while-loops and trying to decompile while-loops to get information that was explicit in the first place.

The `Cc` AST is not processed enough to be easily used. It is translated a last time to another format called `Graph`. In contrast to what the name may indicate, it is not a CFG, but still an AST with horizontal links, for instance between goto-statements and labels, because they are not syntactically linked; yet, control flow nodes that are more local (like loops or conditional branching) are still represented as subtrees. This format contains almost all the information that may be computed statically and that is useful during the analysis. For instance, goto-statements are annotated with the set of variables to remove and to add. Another interesting point is that expressions are function call-free. Function calls are represented by a separate node, just as we did in ASCLEPIUS (see chapter 4 "A Minimal C-like Language" (page 43), and especially definition 4.3 "Statements" (page 44)). Indeed, when function calls are ordinary expressions, the semantics is more difficult to define since function calls imply running statements. `Graph` AST also includes synthesized code, in particular "glue" code added before and after function calls to initialize parameters variables (to the expressions in arguments) and to copy the returned value in the variable from the function call. `Graph` tree is the format on which we run the analysis.

Other things need to be done initially. A very important one is to instantiate abstract domains according to configuration (coming from files and command line). ASTRÉE uses a composite abstract domain made of a stack of parametric domains (including a product of domains). This is implemented with OCAML functors and will be detailed in

[¶]Frama-C does the exact opposite in CIL (C Intermediate Language) (see [Nec+02]): it removes all redundant syntax features.

section 13.2 “Composite Abstract Domain” (page 165).

Astrée instantiates also a central component: the iterator. This is the module that iterates over the `Graph` tree and calls appropriate transfer functions of the abstract domains. It will be detailed in section 13.3 “The Iterator” (page 169). Morally, this is the only part that is language-dependent. Abstract domains should be seamlessly pluggable into another interpreter designed for another language. Of course, the separation is not that strict, especially because the transfer functions we need in domains depend on the features of the language. We can compile any language to use a minimal set of transfer functions, but this would mean to give up information that is obvious in the first place. This is the same kind of trade-off that we discussed when examining the question of loops. We can have a lot of transfer functions, some of them may be complex, while keeping most original structure and information. The other strategy is to compile to a reduced set of transfer functions, and we can expect them to be simpler. In this last case, to get syntactic information, we need to reverse this compilation, which is not robust. In most cases, in Astrée, the chosen strategy is to keep the analyzed program as it is, even if it requires more complex transfer functions.

13.2 Composite Abstract Domain

Astrée uses a composite abstract domain made of a big tree of parametric abstract domains. A parametric abstract domain is a domain that is defined with respect to other domains. It can be seen as a function whose parameters and images are abstract domains. Of course, there are non-parametric domains as leaves of the tree of domains.

They are several uses to parametric domains. They can simply be reduced product, as presented in section 12.3 “Partially Reduced product” (page 159) (and [Cou+06b]), or the improved product explained in Part IV “A Reduced Product with Ghost Variables” (page 229). Indeed, given two domains the reduced product builder produces a new domain that have the expressive power of both domains. By iterating, we can generalize this product to take an arbitrary number of domains to combine.

Another application of parametric domains is to change the type of abstraction. For instance, from a domain that abstract sets of integers, we can build a domain that abstracts memory states by mapping each variable to an abstract state of the underlying domain. This abstraction is non-relational: each variable is abstracted independently of each other.

Astrée uses these two kinds of parametric domains. On the top (i.e. closer to the interpreter, the outermost domains), there is a stack of domains that abstract away several kinds of language features. On the very top, they abstract executions traces with concurrency, while the domain at the bottom only abstracts numbers. Consequently, each domain has its own signature (module type) adapted to the type of the concrete domain. This stack of domain is thus very rigid: they cannot commute since we need to simplify the type of the concrete domain from the outside to the inside.

Under this stack of domains, there is a tree of reduced product of relational numerical domains. At this level, pointers have already been simplified, domains only see physical memory cells. These domains abstract set of maps from memory cells to integers of floating-point numbers. The leftmost domain of this tree is particular: it is a non-relational domain build from a domain that abstracts set of integers or floats, as explained previously. In fact, this underlying domain is also composite: it is a product of domains that abstract numbers, like the interval or the parity domains used in the example of section 12.1 "Example: When Reduction is Needed" (page 156).

Let detail each part from the bottom.

13.2.1 The Tree of Non-relational Numerical Domains

At the very bottom of the tree, there are two products of non-relational domains: one abstracts integers and the other abstracts floating-point values.

These domains are very simple and their interface is as well. For instance, they cannot evaluate directly any arbitrary expression, but they provide transfer functions for elementary operations only. They are like abstract pocket calculators.

Their reduction possibilities are also limited. Transfer functions do not directly take communication channel on the pre- and postcondition. Here, communication is made separately.

Both trees (integers and floats) are lifted to a domain that abstract functions from memory cells to values. Memory cells are typed, thus, we know which should map to integer abstract value and which should map to floats. The lifter domain fulfill other tasks. Especially, it manages iteration on underlying domains and communication between them.

From a theoretical point of view, the non-relational composite domain abstracts an element of $\mathcal{P}(C \to V)$ (where C are memory cells and V the set of possible values) into an element of $C \to \mathcal{P}(V)$. For instance

$$\left\{ \begin{pmatrix} a \mapsto 1 \\ b \mapsto 1 \end{pmatrix}, \begin{pmatrix} a \mapsto 2 \\ b \mapsto 2 \end{pmatrix} \right\}$$

is abstracted into

$$\begin{cases} a \mapsto \{1, 2\} \\ b \mapsto \{1, 2\} \end{cases}$$

losing the relation $a = b$. Indeed, its concretization is

$$\left\{ \begin{pmatrix} a \mapsto 1 \\ b \mapsto 1 \end{pmatrix}, \begin{pmatrix} a \mapsto 2 \\ b \mapsto 2 \end{pmatrix}, \begin{pmatrix} a \mapsto 1 \\ b \mapsto 2 \end{pmatrix}, \begin{pmatrix} a \mapsto 2 \\ b \mapsto 1 \end{pmatrix} \right\}$$

Then, the abstraction of $\mathcal{P}(V)$ is delegated to underlying domains. Likewise, the implementation of such a domain usually works with an associative container from C to elements of the underlying abstract domain.

This composite domain is very imprecise, as it is unable to establish relations between variables, but on the other hand, it is very efficient both in time and memory consumption. Moreover, this domain is able to run, even imprecisely, any kind of expression: it has an opinion on everything, even if it is not an accurate one.

13.2.2 The Tree of Relational Numerical Domains

A non-relational domain is a special case of relational domain. From the non-relational composite domain, and other (actually) relational numerical domains, we build a product of domains that abstracts elements of $\mathcal{P}(C \to V)$. This product implements more finely communication between domains, and the signature of these domains is much richer than non-relational domains. Non-relational domains define functions only for elementary arithmetic and logical operators. On the opposite, relational domains directly run statements, like assignments. This allows domains to evaluate expressions by induction according to their own abstract semantics, or to pattern-match expressions to detect expressions of a certain form (e.g. linear combinations) for specific treatments.

Input channels are implemented as a product type of functions. Each field answers a single kind of question, for instance, one of them tells if two variables are equal: it takes two parameters (variable identifiers) and returns a Boolean value. Of course, precomputing all possibilities for each field may be very expensive, and most of them are never used, thus fields implement lazy computation with transparent memoization, so only used properties are computed, and only once. This way, unused properties have a negligible cost.

The downside of using functional values is that they are hard to debug. Indeed, when there is a bug in such a function, it should be understood in the context where this function was created, and not in the context of execution (when the bug appears). But when the bug appears, the context of creation of the function does not exist anymore. This observation will explain several choices in the following.

13.2.3 The Stack of Domains

The relational numerical domain is not sufficient to analyze C programs. This domain is thus gradually adapted to a domain that is able to take all aspects of C into account by a stack of parametric domains. The resulting composite domain is drawn on Figure 13.1, and domains of the stacks are detailed below, from bottom to top.

1. *Pointer domain.* It builds a domain that is able to abstract pointers and numerical variables using a domain that abstract numerical values. It takes the composite relational numerical domains in parameter, and uses it to represent directly numerical values or the offset part of pointers. This domain is explained in detail in [Min13; Min06]. This domain is not able to handle dereference: it knows what pointers point to, but at its abstraction level, variables are memory cells and expressions do not longer contain indirections: they are solved by the immediately upper domain: struct domain.
2. *Struct domain.* This domain abstracts C aggregate types (structures, union and array). Also, it resolves pointers: above this domain expressions can use C variables and dereferences; under it (like in pointer domain), expressions contain only physical cells, and indirections are solved and thus absent from expressions. To resolve dereferences, it uses points-to information given by the pointer domain. This domain is also exposed in [Min13; Min06].

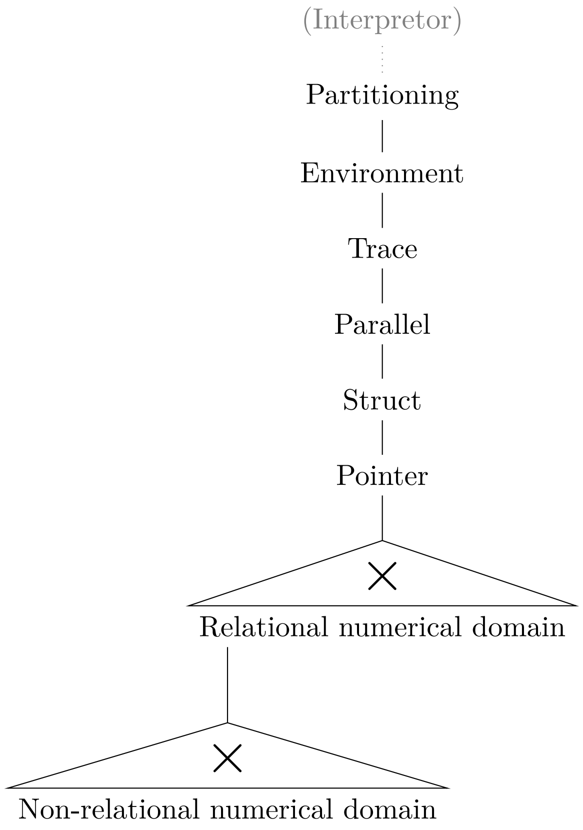


Figure 13.1: Structure of the composite domain of Astrée

3. *Parallel domain.* Real-world programs are often multi-threaded. The parallel domain allows analysis of multi-threaded programs using a rely-guarantee method. Underlying domains have the responsibility to track the effect of assignments and to apply the effect of assignments performed by other threads. Nevertheless, except these operations, that are explicit, underlying domains run the program as if it was in a single thread, so without worrying about multi-threading. One can find details about this domain in [Min13; Min14].
4. *Trace domain.* Trace domain allows state partitioning according to an automaton. We will not be interested anymore in this one.
5. *Environment domain.* Under this domain, execution traces seem uninterrupted. To handle discontinuity in control flow (like goto, continue, break or return statements), environment domain implements unsynchronized iterations (see [Cou78]). This domain receives commands that describe the control flow. For instance when it receives the command corresponding to a goto, it suspends the current execution, and restores it only when it gets the command signaling the corresponding label. The way this domain works is explained more precisely in section 17.2 "Intra-function Near Jumps" (page 212), as we extend it to handle another kind

of jumps using a method very similar to what is already implemented.

6. *Partitioning domain.* This domain performs partitioning according to the trace prefix: whether it went through an if, or through the else, for instance. This part is not very interesting to us. Nevertheless, as it is just under the iterator, it has special duties. In particular, the iterator prepares the function that performs the analysis of a C function and gives it to the partitioning domain to stabilize backward gotos and handle return statements.

13.3 The Iterator

The composite domain that we have described defines the type of abstract values and transfer functions to act on abstract values. The iterator is in charge of reading the program and calling transfer functions accordingly.

As input, the interpreter takes a program in the `Graph` format already mentioned. We remind that this format is mainly an AST, and not a CFG. Iterating on a CFG is simpler: one just has to identify loops, choose at least one widening node on each loop (though this decision can be made dynamically) and use a worklist algorithm. This algorithm keeps a list of nodes to update. At each step, it takes a node, updates it according to its predecessors and if the condition is not stable, it adds the successors in the list. This algorithm makes the abstract execution (the analysis) very different from the concrete execution, and thus bugs of the iterator and domains may be difficult to find and understand.

Moreover, transforming the program into a CFG drops most syntactic information, that could help heuristics. There is another drawback with CFGs: we need to be able to build it, making languages with dynamic jumps more difficult to handle. Even if pure C does not have such dynamic jumps[**], x86 has (and consequently, mixed C and x86 also). It is relatively easy to dynamically compute successors of a node, but we need to dynamically chose widening-nodes as well. This is a uselessly complicated design.

In contrast, the AST approach has the main drawback to force the iterator to handle a tree with a large variety of nodes, one for each language feature. It may also require more transfer functions in the domains. Once again, it is a reasonable trade-off since it allows domains to be more precise than they would be by decomposing complex operations into elementary ones.

The AST approach is very similar to the inductive definition of a denotational semantics. There is one tricky feature of C that is not obvious to handle: function pointers. When a function is called through a pointer, we need to get the functions this pointer can designate. For this purpose, the abstract value of the pointer must be concretized. Usually, we avoid this operation as it can be costly and the concretization of an abstract value can be a very large set, even infinite. Here, this is more reasonable: the concretization cannot be bigger than the set of all functions of the program, which is usually not

[**]A GNU extension, called “Labels as Values”, allows pointers to labels. Function pointers are also dynamic but it is more a problem of function calls, since they adapt poorly in CFGs.

excessive. Then, we check there is no recursive calls. ASTRÉE does not handle recursivity: it is both very difficult to do precisely and useless since ASTRÉE is mainly dedicated to embedded programs, which do not allow recursivity. To detect recursive calls, the iterator relies on the partitioning domain which keeps information about past events, such as function calls; if the same function is already present in the trace, the call is recursive and an alarm is raised. Yet, the analysis continues normally for functions in the concretization of the pointer that are not recursive calls. At the end of the analysis of each possible function, postconditions are joined.

This strategy is very precise, as it inlines all function calls, but might be costly in computation time. However, a weaker context-sensitivity is not sufficient for industrial applications. Moreover, it has a serious consequence: the body of a function is analyzed only if it is called. As it seems natural, it hides the fact that it forbids C `longjmp`. As this function is close to an assembly jump between C functions, and this case actually exists in industrial code, we will need to adapt the iterator accordingly.

Part III

Abstraction of Mixed C and x86 Assembly

Chapter 14

Abstraction of Arithmetic and Logic Statements

Abstracting assembly statements requires an enriched model compared to C. Yet, additions and modifications are not trivial and motivated by the semantics of assembly instructions. For these reasons, to make the explanations clearer, we will not detail the model, then the abstract semantics etc. Rather, we will show how to abstract statements by category, introducing elements of the model progressively.

We start by showing the abstraction of arithmetic and logic statements, as they are the simplest. Especially, this class of statements is simple enough so that we do not really need to detail the abstract transfer functions, but we show the consequence they have on the modeling.

14.1 Model

Assembly instructions, unlike C statements, can access registers. We need to abstract them in such a way that their values are persistent between instructions. Otherwise, any write to a register would fall into oblivion and reading it would give the most general value $\top$. Abstracting general-purpose registers as C `int` (or `unsigned int`) variables is, of course, not wrong, but it is not sufficient either. This question will be explored in the next section.

We will in particular emphasize which parts of the central processing unit (CPU) we are ignoring.

A very standard feature of processors is the cache. The need for caches stands in two remarks: the main memory is slow to access from the CPU and programs I/O usually are concentrated on small regions of memory. A cache is a piece of memory of low capacity[*], but very fast that acts as a proxy for the memory: when a zone in memory is read, it is loaded in cache. Following reads and writes will just query and update the cache.

[*]For instance, recent processors such as the Intel Core i9–10980HK have a cache of 16 MB while it allows up to 128 GB of memory.

When asked explicitly (e.g. with WBINVD instruction), or when the cache is needed for something else, modified cells in cache are written in memory.

Nowadays, processors have multiple cores and multiple levels of caches: fastest caches are smallest. Some caches (biggest and slowest ones) are usually shared between cores, but fastest caches are specific to each core. This is a serious matter since writes to a non-shared cache will not be visible from other cores. Writebacks will occur later and might be overwritten by another core that is not aware of modifications done by all other cores. This kind of memory model is called weakly consistent (see [DSB86]). The study of such memory models is not the purpose of our work. Nevertheless, we trust that taking such considerations into account will not invalidate the following. We think it might be solved modularly by wrapping memory domains with a parametric domain (implemented as a functor) that lifts an abstraction of consistent memory to an abstraction of weakly consistent memory, just like the domain described in [Min14; Min13] lifts an abstraction of sequential programs to an abstraction handling multithreading.

When memory ranges used by each core are disjoint, caches are transparent, i.e. memory accesses behave like there were no cache. An even more special case is when there is only one core (or at least, we use only one). Our target software indeed uses only one core, thus it is safe to ignore caches and related instructions.

Caches are managed according to cache policy, but this is not documented: we do not know how the processor uses the cache. For example, when it needs some space, it may invalidate the oldest, the less recently accessed or the less often used data. All are legitimate strategies and have very different behaviors. The CPU might also use a hybrid strategy, so that, trying to guess is hopeless.

In addition to caches, processors have other internal hidden memories. We already mentioned TLBs (see subsubsection 2.3.3.4 “Translation Lookaside Buffers” (page 28)), that are a special kind of cache to speedup address translation for memory accesses. TLBs have a mixed management: they are mainly managed by the processor, but the user can influence the behavior by marking some entries as persistent, or, conversely, to invalidate them precociously. Since they may be manually managed, TLBs may have a semantic impact if they are poorly used. Indeed, if a page is marked as persistent though it belongs to a simple user process, it will not be purged when a task switch occurs, and address translation will be faulty. Nevertheless, TLBs are ignored for now since our study case does not use them explicitly. Moreover, they are a part of paging, which is not handled yet.

There are other caches that allow the processor to make smart guesses for jumps. Indeed, instructions are processed in several stages: when an instruction enters the second stage, the first one is already free for the next instruction. When we run a conditional jump statement, we need to guess whether the condition will be true, to choose the next instruction we start to execute, before actually being able to decide. If the guess is correct, the execution is pursued, otherwise, the current instruction is dropped, and the right instruction is run instead. This technique is called speculative execution. To help this guess, the processor uses a memory of lastly evaluated conditional jumps. This kind of additional memory is ignored in our abstraction as it does not impact the semantics of instructions.

A very important kind of hidden memory is the hidden parts of registers (see subsubsubsection 2.3.2.2.4 “Segment Selectors” (page 23) and subsection 2.5.2 “Task Register” (page 35)). They are not accessible by the software, but they are documented and crucial for the semantics. For instance, the hidden part of a segment register is loaded from the GDT at the same time as the segment register itself, then only the hidden part is used during address translation, in order to minimize accesses to the GDT that is in (slow) memory. A side effect is that if the GDT is modified but the segment register is not reloaded, it has no effect on segmentation. For that reason, we must take hidden part of segment registers into account. This will be detailed in section 16.1 “System Statements” (page 195).

There are probably other undocumented caches, INTEL processors are famous for that. They are not documented because they do not impact the semantics with respect to the model described in [Int20]. They have an impact that is critical for other kinds of studies, especially, when we are concerned about execution time, as well as when we are seeking a bound or when we are interested in determinism. Since we do not care about timing, we can ignore these kinds of undocumented features.

To sum up, the main addition with respect to C is the presence of registers. Another important difference is the point of view over the stack. While C and assembly both work with the same stack they consider it with a different level of abstraction. This will be treated in a following chapter (see chapter 15 “Stack Abstraction” (page 183)).

14.2 Register Abstraction

General-purpose registers are type-agnostic sequences of 32 bits: they can contain a signed integer, an unsigned integer, a binary coded decimal number, a pointer or simply a sequence of flags. Unlike C variables, that are considered as numbers independently of their representations, the meaning of assembly register depends on the context. The problematic point is not about storing arbitrary values into a variable, but that C operators behaviors change according to the type.

```
1  int x, y;
2  y = &x;
3  y++;
```

Listing 14.1: Applying integer arithmetic to a pointer

For instance, in the snippet of Listing 14.1, at line 2, `y` stores a pointer. The C standard allows this conversion up to some implementation defined behaviors that are known to be favorable in our case. ASTRÉE is already able to analyze such code, thanks to the pointer domain (see [Min06; Min13]). But the interesting point is that the increment of line 3 must be understood in integer arithmetic, and not in pointer arithmetic.

Due to type-agnosticism of registers, the meaning of each operation do not depend on the type, but on the opcode. Some operations are meaningless with some types. For instance, applying a binary coded decimal instruction to a register storing a pointer will give an unpredictable result. This kind of situation is easy to detect, and we can raise an alarm and default to $\top$ in such cases. A problem arises when a single opcode is valid for multiple datatypes that are compatible. For instance, `ADD` performs the sum as well for signed or unsigned integers, thanks to two's complement. It is not *a priori* known whether the result will be used as a signed or unsigned integer. We shall assume both are possible. Moreover, to set accurately the status flags in EFLAGS register (see subsubsection 2.2.1.2 "EFLAGS Register" (page 11)), we indeed need both signedness.

In our application, this is the main ambiguous case. Thus, we modeled each register by a pair of variables: a signed integer and an unsigned integer. This abstraction relies entirely on the underlying abstraction of variables, and its concretization is the intersection of two's complement representation in each concretization.

Formally, given an abstract domain $\mathcal{D}^\sharp$, if we let $r_s \in \mathcal{D}^\sharp$ (resp. $r_u \in \mathcal{D}^\sharp$) be the abstract value representing a signed (resp. unsigned) 32-bit integer that represents register r, $\gamma_s : \mathcal{D}^\sharp \to \mathcal{P}(\mathbb{Z})$ (resp. $\gamma_u : \mathcal{D}^\sharp \to \mathcal{P}(\mathbb{Z})$) be the concretization of a signed (resp. unsigned) variable and $\beta : \mathbb{Z} \to \{0,1\}^{32}$ the two's complement representation of an integer over 32-bit. The concretization of the register r is

$$\begin{aligned} \gamma : \mathcal{D}^\sharp &\to \mathcal{P}\left(\{0,1\}^{32}\right) \\ (r_s, r_u) &\mapsto \{\beta(n) \mid n \in \gamma_s(r_s)\} \cap \{\beta(n) \mid n \in \gamma_u(r_u)\} \end{aligned}$$

```
int a = 4;
int b;
void f() {
    asm {
        mov EAX, a
        mov EBX, EAX
        mov b, EAX
    }
}
```

Listing 14.2: Copying through registers

For instance, the code on Listing 14.2 copies `a` to `b` through registers EAX and EBX. Roughly, it is executed as code on Listing 14.3. The assignment from `a` to EAX is executed as an assignment from `a` to signed and unsigned values of EAX. We can notice that the unsigned version receives `*((unsigned int*)(&a))`, this is the binary sequence of `a` reinterpreted as an `unsigned int`. Then, when EAX is used, we select the version to use according to the type of the destination: a register destination requires two copies (like the copy from EAX to EBX), but a C variable destination allows a more precise decision based on the signedness of the destination.

```c
int a = 4;
int b;
void f() {
    EAX_s = a;
    EAX_u = *((unsigned int*)(&a));
    EBX_s = EAX_s;
    EBX_u = EAX_u;
    b = EAX_s;
}
```

Listing 14.3: Copying through registers in pseudo-C

When the operation using a register is not signedness-agnostic, like multiplication, we select only the right version of the register to perform the operation. Both variables associated to the register are updated according to the result: one is a mere copy, the other requires to be reinterpreted.

It is interesting to remark that we cannot directly rely on the corresponding C operation. For instance, overflow during C addition of signed integers is an undefined behavior, but it nicely wraps around in assembly. In C, we perform the integer addition, followed by a bound check, while in assembly, we use the integer addition followed by a modulo. The difference is quite simple, but it shows once again that a C stub for assembly instructions is insufficient.

14.3 Partitioning According to EFLAGS

This section has been co-thought with Yves-Stan Le Cornec, who had a research engineer position in Antique team at that time, and entirely implemented by him in the development version of Astrée.

Arithmetic operations set status flags in EFLAGS register. These flags are typically used by conditional jumps.

```asm
cmp EAX, EBX
je label
; ...
label:
```

Listing 14.4: A conditional jump

Let us illustrate with Listing 14.4. The `CMP` instruction performs a subtraction and sets status flags accordingly but does not save the result. The conditional jump mnemonic `JE` (jump if equal) is an alias for the mnemonic `JZ` (jump if zero), indeed $\text{EAX} - \text{EBX} = 0 \Leftrightarrow \text{EAX} = \text{EBX}$. In detail, it jumps if ZF flag of EFLAGS register is set. In this example, the control jumps to `label:` if and only if EAX and EBX are equal. A way to assign ZF is to compute the intersection of abstract values of EAX and EBX. If the intersection is not empty, then ZF may be set. Conversely, ZF may be cleared if EAX and EBX may have different values. Without further precaution, this is too naive, because the constraint ZF = 1 does not propagate backward to EAX = EBX. We need to remember that

$$(\mathrm{ZF} = 0 \Rightarrow \mathrm{EAX} \neq \mathrm{EBX}) \wedge (\mathrm{ZF} = 1 \Rightarrow \mathrm{EAX} = \mathrm{EBX})$$

This kind of relational information is called partitioning. One can use any kind of condition to partition the abstract state, including a Boolean formula defined over status flags. Instructions that update status flags, e.g. arithmetic or logic instructions, perform such partitioning so that a possibly following conditional jump would behave correctly. This is implemented using binary trees [Mau99] where nodes are annotated with conditions and leaves are abstract states of the underlying domain.

14.4 Implementation

Internally, registers are declared as global variables. They are added into the set of global declarations, just before being passed to the interpreter for allocation.

The translation from source code instructions to instructions on signed and unsigned representation of registers must be done above the struct domain (as explained in subsection 13.2.3 “The Stack of Domains” (page 167)), since variables are translated into physical cells in this domain. There is no intrinsic upper bound for the position in the stack of domains, but it is better to put is as low as possible: the lowest it is, the most features are already abstracted and the simplest the domain is to implement. The new structure of the composite domain is shown on Figure 14.1: there is a new CPU domain that is in charge of translating instructions as they are in the source code to the instructions required for underlying domains. This domain is not limited to this task, as we will see later (see chapter 15 “Stack Abstraction” (page 183) and chapter 16 “Other Non-control Statements” (page 195)).

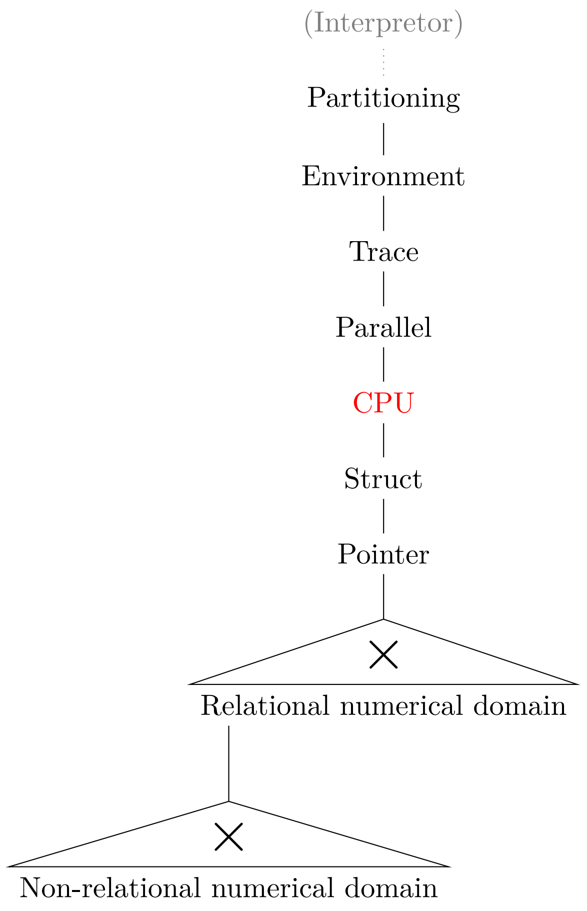


Figure 14.1: New structure of the composite domain of ASTRÉE

14.5 Envisaged Improvements

Several improvements have been considered; this section will detail some of them. Most of them have not been implemented since it was not felt useful, neither in terms of performance nor precision. Moreover, while some of these optimizations are very lightweight, others implies heavy modifications of the implementation.

14.5.1 Precision Improvements

The first kind of improvements aims to increase the precision of abstractions, so as to avoid false alarms.

14.5.1.1 Reduction of Both Register Representations

Signed and unsigned versions of registers are updated independently as much as possible. When only one signedness is available, it is reinterpreted so as to fill the other version.

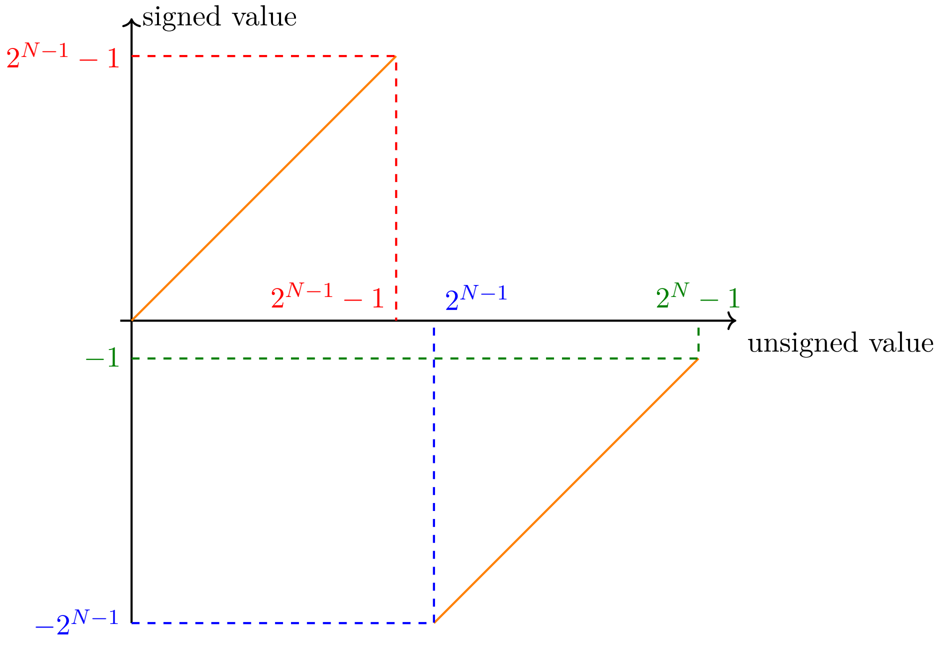


Figure 14.2: Relation between signed and unsigned integer with same representation

But after this point, when we perform operations independently on signed and unsigned values, one of the two version may lose precision. Since both variables are linked by a well-known relation, we may hope to refine them. The relation is the equality of binary representation, which implies a relation on the integer values. This relation is shown on Figure 14.2.

This reduction was not implemented as it was not necessary for our applications. The next subsubsection describes another precision improvement that was enough by its own.

14.5.1.2 Wrapped Interval Domain

Another approach to handle the signedness-agnosticism of registers is to directly use abstract property that are themselves signedness-agnostic. Some properties used on integer arithmetic can be quite nicely adapted to modular arithmetic. In particular, this is the case of the interval domain: we can think of modular intervals. Such a domain has been implemented as described in [Nav+12]. Yves-Stan Le Cornec implemented this domain in our development version of Astrée.

The idea of such domain is to allow intervals to wrap around, as illustrated on Figure 14.3. This is useful when the interval representing a variable is pushed (by an addition) on each side of the maximum value of the type. Using a simple (integer) interval domain, the convex hull of such a wrapped interval would give all representable values, which is the most imprecise abstract value. With a wrapped interval domain, the addition of a constant value never increases the size of the interval.

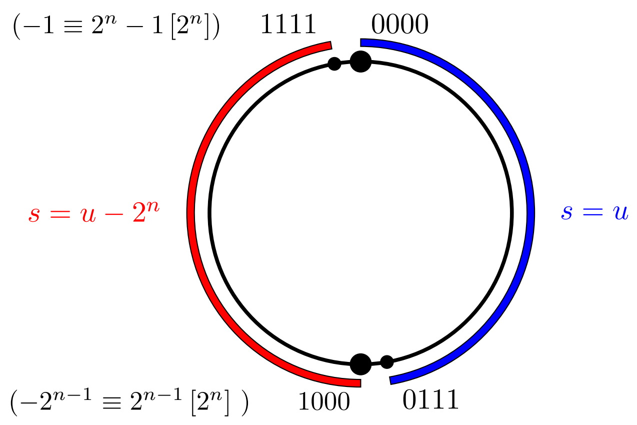

$(-1 \equiv 2^n - 1\,[2^n])$
1111
0000
$s = u - 2^n$
$s = u$
$(-2^{n-1} \equiv 2^{n-1}\,[2^n]\,)$
1000
0111

Figure 14.3: Representation of wrapped intervals with $n = 4$ bits

14.5.2 Performance Optimization

In addition to making the analysis precise enough to prove that the code is correct, we are also interested into analysis time. Since we are targeting industrial applications, one must be able to run the analysis in reasonable time. This is the second kind of optimization: without losing precision, we want the analysis to run faster. In fact, it may be acceptable to have a less precise analysis as long as we can still prove desired properties.

Optimizations mentioned here should not impact precision, but only execution time. They have not been implemented since the performance of this part of the code is considered to be good enough, as far as we know.

14.5.2.1 Minimizing Partitioning

When there is a sequence of arithmetic instructions, status flags of each instruction are overwritten by the next one. One can thus avoid partitioning according to flags that are known to be ignored. For instance, in Listing 14.5, flags set by `SUB` of first line are always overwritten by `ADD` on the second line. Thus, it is useless to partition at the first line. The third line checks the value of OF flag, thus, we need to partition at least according to this flag at the second line. But is would be foolish to think that it is enough. Indeed, at the destination of `JO`, we may need the other flags, for instance, if the next instruction is another conditional jump. When destinations of jumps are statically known, we may build the dependency graph to find which flags of which instructions may be used. When a jump has a dynamic destination, this deduction is impossible, and we must consider that all flags may be used.

This is morally simple, and in fact a classical method in compilation to find unused values. Implementation is also quite straightforward but a bit extensive as it would

```
sub EAX, EBX
add ECX, EDX
jo label
```

Listing 14.5: A situation where some flags may be ignored

require deducing more information in the front end, and to forward it to the domain that handles partitioning.

14.5.2.2 Lazy Evaluation of Flags

The optimization described in the previous subsubsection can prevent partitioning with respect to flags that will always be ignored. But we can go even further and never compute any partitioning and the value of any flag *a priori*. The point is to remember which instruction is the last to assign each flag. When the value of a flag is requested, the adequate instruction is replayed to deduce the useful partitioning.

The previous strategy was clearly an optimization since it has a linear cost at initialization time (which is considered to be reasonable) and is only an improvement afterward. On the other hand, evaluating lazily flags as some cost: if status flags are needed rarely enough, asymptotically, this strategy is profitable; conversely, if we use all flags, it implies an overhead. Consequently, this strategy would need to be tested before being able to know how efficient it is. Moreover, from a practical point of view, it is quite complex to implement, thus, we deemed not to try it for now.

Chapter 15

Stack Abstraction

Both C and assembly use a stack but, as it was explained in chapter 8 "Semantics of Mixed C and Assembly" (page 99), they have a very different point of view of it.

This chapter proposes an abstraction of the stack that is compatible with both languages, and is expressive and precise enough for our industrial application. Globally, the stack is segmented into two kinds of blocks: C call-frames and explicit stack segments manipulated by assembly. The point is to make this abstraction as sparse as possible in order to minimize the set of additional hypothesis, and to make the abstract state as lightweight as possible.

It is good to keep in mind that we are allowed to reject behaviors that are correct according to the concrete semantics, but for which the abstraction is not good enough or too complicated. We are mainly interested into accepting behaviors that actually exist in our applications. In practice, the abstraction described in this chapter accept most legal behaviors requiring only crucial assumptions.

15.1 Abstracting Stack in a Single Assembly Block

15.1.1 Informal Explanation

When entering an assembly block from C, the stack is not empty: it contains the call frame of all currently active functions. We know that the local variables are contained in the stack, but we cannot tell where. For the C code after the assembly block to behave correctly, we need the stack to be the same at the end of the assembly as it was at the beginning, up to local variables that may be safely changed.

Since we do not know the content of the stack, it is difficult to guarantee that after some modifications, it is restored to its initial state. A safe criterion is not to touch the cells of the stack that were existing previously, except local variables that may be accessed by the literal `x[EBP]` that is an alias for the C local variable `x`. For that access to be legal, we add another constraint to check: EBP has not been modified before and it is restored by the end of the assembly block. We simply use a copy of EBP to remember its original value, and guarantee that it is unchanged when used or exiting the assembly block.

Since we do not know what is initially in the stack when we enter the assembly block, it is pointless to read it, and we said that we want to forbid modifications in this region. Overall, we exclude any kind of access in this region of the stack: we want to ignore it completely. A natural abstraction is to represent only the suffix of the stack, initially empty and that must be empty at the end of the block.

Let us illustrate this abstraction over examples before giving a formalization. The example of Listing 15.1 pops the top of the stack and stores it in EAX, then this value is pushed on the stack. Obviously, the stack is restored to its initial state* (and only EAX is changed, which is allowed), but this code is rejected as it reads outside the suffix of the stack, especially, the `POP` will fail with a definite error, since it reads the unknown prefix. Such code does not exist in practice since we do not know what the `POP` will return, and thus, there is no point to get this value. The Listing 15.2 shows an example of a correct code that is provable by this abstraction. At the beginning of the assembly block, the stack is totally unknown, with an empty suffix. We push the integer 1 on the stack, so after the line 4, the suffix is 4 bytes long (when operating in 32-bit mode). Line 5 pops 4 bytes and stores them in EAX, The suffix is empty when we exit the assembly block, thus the stack is restored to its initial state.

```
1  int f()
2  {
3      asm {
4          pop EAX
5          push EAX
6      }
7  }
```

Listing 15.1: A rejected legal code

```
1  int f()            //+----- unknown prefix of the stack
2  {                  //↓  +-- suffix
3      asm {          ; ...|
4          push 1     ; ...| 1 |
5          pop EAX    ; ...|
6      }              // Empty suffix: OK!
7  }
```

Listing 15.2: A provable legal code

The suffix cannot be stored as independent variables since they are contiguous in memory, as illustrated by Listing 15.3: popping the stack is replaced by an addition

*In fact, it is not if the stack is initially empty: popping is an error. This is not possible in practice, but not forbidden by our semantics.

```
int f()              //+----- unknown prefix of the stack
{                    //↓   +-- suffix
    asm {            ; .../
        push 1       ; .../ 1 /
        add ESP, 4   ; .../
    }                // Empty suffix: OK!
}
```

Listing 15.3: A slightly more complicated provable legal code

into ESP. The suffix is stored in an array `a` of constant size `N`. When we enter an assembly block, ESP is set to `a` + `N`, that is, it points after the end of the array. If we perform too many `PUSH` on the stack, we will entirely fill the array. At this point, the abstraction defaults to $\top$, and we shall abandon all hope to restore the original value of the stack by the end of the assembly block. We are doomed to use a statically allocated array since dynamically allocated array are much more difficult to abstract precisely. Moreover, a good choice of the size of this array prevents overflow in the abstract. Indeed, in critical embedded software, memory is a rare resource, we have to know exactly how much memory we use. This implies the choice of guidelines that promote some programming styles, and avoid others. For instance, most variables (especially arrays) are global, so that they are statically allocated at the beginning of the program, and there is no subsequent `malloc` that can fail. We also avoid recursive functions, as we mentioned before. Likewise, `PUSH` statements do not appear in loops, as it could lead to a stack overflow; in practice, they only appear outside loops (but possibly in conditional branching), and thus are executed at most once. This is the foundation of our heuristic to choose the size of the array that store stack suffixes (the constant `N`): the array must at least be large enough to store as many 4-byte variables as the maximum number of `PUSH` statements in a single assembly block[†]. In the examples of Listing 15.2 and Listing 15.3, we would choose a size of 4 B (1 `PUSH` $\times$ 4 B).

Of course, it is very easy to trick this heuristic, as shown on Listing 15.4. Here, we see only one `PUSH`, thus, the heuristic deduces that an array of size 4 is enough. But when we extend the stack for the second time, at line 5, the array is not long enough, thus the abstract value defaults to $\top$, for want of something better. All following statements raise an alarm since we cannot guarantee that they are error-free: $\top$ includes the possibility that the suffix is empty, in which cases, lines 6 to 8 would all be faulty.

Such tricky code is rarely encountered in practice, and an ad hoc fix to the previous heuristic is to add a big enough static number of cells in the array. The choice of 10 4-byte cells emerged to be satisfactory for all examined cases. It could be changed by the user if needed.

[†]If we assume that the stack suffix must be empty at the beginning of each assembly block. If we implement the bridging described in subsubsection 8.3.3.3 "Bridging Blocks" (page 109), we should adapt slightly this heuristic.

```
int f()
{                          // concrete suffix ! abstract suffix
    asm {                  ; ... |              ! ... |
        push 1             ; ... |  1 |         ! ... |  1 |
        sub ESP, 4         ; ... |  1 | ?? |    ! ... |  ⊤
        mov [ESP], 42      ; ... |  1 | 42 |    ! ... |  ⊤ (error ?)
        pop EAX            ; ... |  1 |         ! ... |  ⊤ (error ?)
        pop EAX            ; ... |              ! ... |  ⊤ (error ?)
    }
}
```

Listing 15.4: A provable legal code that fools the heuristic

15.1.2 Formal Details

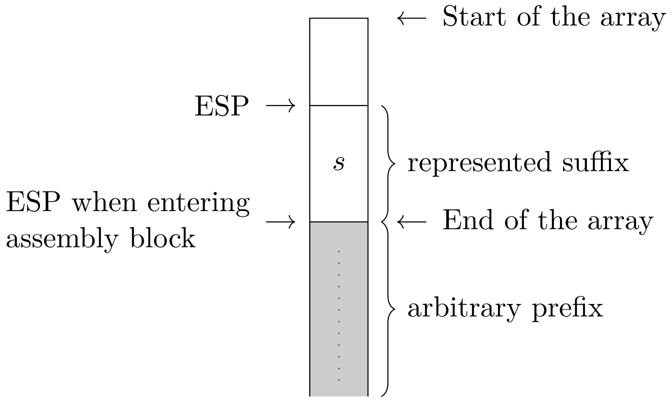


Figure 15.1: Stack suffix abstraction layout

The stack abstraction that we have presented is a two-stage abstraction: we abstract the stack by a stack suffix, and the stack suffix by a fixed-sized array. Let Σ be the alphabet of the stack. In the case of x86 assembly, $\Sigma = \llbracket 0, 2^8 - 1 \rrbracket$. But this method can be used for any kind of stack, though it may not be well-fitted. A stack is a word in $\Sigma^\star$, as we exclude infinite stacks.

For the first stage, we need an abstract domain $\mathcal{D}^\sharp_\mathcal{S}$ that abstracts stack suffixes with $\gamma_\mathcal{S} : \mathcal{D}^\sharp_\mathcal{S} \to \mathcal{P}(\Sigma^\star)$ its concretization. An abstract state $s^\sharp \in \mathcal{D}^\sharp_\mathcal{S}$ is an abstraction of the set of the potential contents for the blocks in the represented suffix area. This set is defined as a parameter of our abstraction by the means of a concretization function $\gamma_\mathcal{S}$. Since we keep no information about the prefix, $s^\sharp$ may also be interpreted as the set of the potential values for the entire stack (by completing a potential suffix with an arbitrary prefix). This interpretation is formalized by the following concretization function:

$$\begin{aligned}\gamma : \mathcal{D}^{\sharp}_{\mathcal{S}} &\to \mathcal{P}\left(\Sigma^{\star}\right)\\ s^{\sharp} &\mapsto \left\{ p \cdot s \,\middle|\, (p, s) \in {\Sigma^{\star}}^{2}, s \in \gamma_{\mathcal{S}}\left(s^{\sharp}\right) \right\}\end{aligned}$$

where $a \cdot b$ is the concatenation of a and b.

The second step is the abstraction of the suffix: how we build $\mathcal{D}^{\sharp}_{\mathcal{S}}$ and $\gamma_{\mathcal{S}}$. ESP is abstracted, as explained in section 14.2 “Register Abstraction” (page 175), by a variable in the underlying domain. Likewise, the array is abstracted as a single array whose size is fixed and computed before the analysis. The abstract domain is simply the underlying domain, but the concretization has to be slightly tweaked, to transform the concrete array into a word such that the last cell is the first letter. This is a simple right composition by $i \in [\![0, N-1]\!] \mapsto N - 1 - i$ to revert the order of letters.

15.1.3 Discussion

The preeminent qualities we expect from an abstract domain is to be precise enough to prove the desired properties on the target code and fast enough to make the analysis run in reasonable time. This abstraction meets both expectations. Moreover, it is really easy to implement since most abstract transfer functions rely directly on underlying transfer functions. For instance, `PUSH` and `POP` instructions are converted into reads and writes in the suffix array (and updates of ESP), and delegated to the underlying domain. Non-stack operations are simply forwarded to underlying domains.

The main draw back is that it does not behave well when performing a union between abstract states with suffixes of unequal sizes. In this case, the abstract value of ESP allows it to point to several cells of the array, which make difficult to prove that the suffix is empty at the end of the stack. A possible solution is to partition according to the value of ESP to keep a precise abstraction of ESP in each case.

We could relax slightly the constraints using a stronger hypothesis. If all local variables are allocated at the beginning of the function, and are all accessed relatively to EBP, ESP does not really matter during function execution. We just need to restore it correctly before executing the postlude that cleans the current call frame. This would allow more convoluted stack usage by simply saving ESP in a local variable, and restoring it at the end. Of course, while it is easy to check the stack is cleaned up, stack instructions will still suffer the lack of precision.

15.1.4 Evolution Possibilities

This abstraction allows only a fixed maximum length for the suffix. As explained, for our application, this is perfectly satisfactory, but we may be interested into possible evolution for other kinds of software.

The main difficulty is the representation of unbounded stack suffix, typically when `PUSH` statements appear in a loop (or a recursive function). In fact, there is very little difference with array abstraction, and thus, that would fall within the purview of conundrums of shape analysis. Classical abstractions that could help are modulo smashing and segment smashing.

Modulo smashing may smash all the iterations of a loop into the variables pushed by the first iteration. For instance, in the case of Listing 15.5, it is likely that `push EAX` always push a value that has morally the same meaning at each iteration. The same remark stands for `push EBX`. Consequently, mixing the value of EAX (resp. EBX) at each iteration may be an acceptable abstraction. Modulo smashing allows a segment of stack to be represented as only two 4-byte cells (one for each syntactic `PUSH`) and an abstract integer to keep the length of this segment of stack.

```
head:
  ; ...
  push EAX
  push EBX
  ; ...
  je after
jmp head
after:
```

Listing 15.5: Two `PUSH` statements in a loop

As the code may contains several loops, one can represent the whole stack as a sequence of modulo-smashed segments, thanks to segment smashing.

Of course, these are only simple strategies that have been selected as they are satisfactory on simple examples. Better-fitted abstractions must be designed according to the code one would like to analyze, with the help of the extensive literature on array abstraction, e.g. [Čer03; HP08; CCL11].

15.2 Abstracting Successions of Call Frames

15.2.1 Recall on Mixed Calls and Calling Conventions

Functions can either be implemented in C or in assembly and called from C or assembly: there are 4 combinations. There are two situations relatively simple, when the calling language is the same as the language of the callee. When a C statement calls a C function, the stack is no matter to be concerned about. Likewise, when an assembly function is called by an assembly statement, the stack is explicit in both sides, there is nothing special to do: assembly calls are not handled differently as jumps.

But when languages are mixed, we need supplementary assumptions to make them work together. The set of these assumptions is a calling convention (see section 6.5 "Calling Convention" (page 87)). A calling convention specifies how arguments are passed to functions, how result values are returned, whether the caller or the callee saves each register, who allocates the call-frame etc. We use a calling convention called cdecl. We recall quickly the characteristics:

– parameters are passed on the stack in right-to-left order;
– result (if any) is returned via EAX;
– EBX, EBP, ESP, ESI and EDI are callee-saved, other are caller-saved.

```c
extern int h(int);

void h_wrapper() {
    asm {
        h:  ; We do not need to save callee-saved
            ; registers as we do not modify them.
        mov EAX, [ESP + 4]   ; copy the parameter in EAX, thus EAX <- 1
        add EAX, 1337
        ret                  ; Return EAX (1338)
    }
}

int g(int a) {  // here a == 42
    return a + h(1);  // return 42 + 1338 (== 1380)
}

void f() {
    int result;
    asm {
        push 42                ; Pushes the argument of g on the stack
        call g                 ; (We do not need to save
                               ;  caller-saved registers)
        add ESP, 4             ; Cleans the stack
        mov result[EBP], EAX   ; copy the returned value in result
    }  // result == 1380
}
```

Listing 15.6: An alternation of C and assembly call frames

Let us illustrate on an example how we end with an alternation of C and assembly call frames thanks to the snippet of Listing 15.6. The execution starts in `void f()`. We are in a C function, thus, the stack contains a C call frame, but we do not care about the stack in pure C. When entering the assembly block, the stack gets an explicit suffix. In this assembly block, we call the C function `int g(int)`, which adds a new C (thus opaque) call frame above the explicit segment of the stack. Finally, `g` calls assembly function `int h(int)`, which appends a new explicit suffix atop of `g`'s C call frame. At this point, the stack has a quite complex structure, as depicted on Figure 15.2.

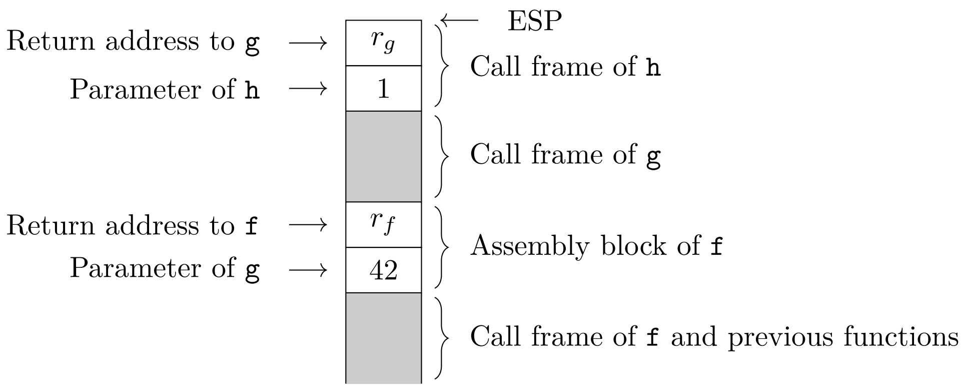


Figure 15.2: State of the stack in `h` in the example of Listing 15.6

Then, `h` returns using a `RET` statement, which pops r_g. Since the return type of `h` is `int`, EAX contains the returned value. According to cdecl, it is `g`'s job to clean the stack from the parameters. As `g` is a C function, it returns using a C `return` statement, and since it was called from assembly, the returned value of `g` is in EAX. The caller is in charge of cleaning the stack.

In this example, we implicitly used a callee-saved register: EBP in `f`. When we use EBP to access the local variable `int result`, we need EBP to be restored to the value it had before `call g`. Since EBP is callee-saved, it is the responsibility of `g` to restore it. Nevertheless, EBP is modified in between, it may even been changed explicitly in `h`. Callee-saved registers are backed up in the call frame of `g`. Even if we do not know how and where they are located, we need to keep this information in the abstraction. One may remark that `g` may not save all registers, but only the ones that `g` writes. It could also save the content of such a register in a caller-saved register, like ECX. However, if `g` calls another function, it will be its responsibility to save ECX, just like it is its responsibility, at the end, to copy back ECX to the appropriate callee-saved register. What matters is that from the caller point of view, callee-saved registers are not restored to their original value.

Conversely, when `h` is called, `g` expects it to restore all callee-saved registers. In this case, we do not save anything since we do not change such register. But, for the execution of `g` to be correct, we need to check that assembly functions let callee-saved registers in the same state they have at the beginning.

15.2.2 The Idea

We assume we start with a C function. When we enter an assembly block, as explained previously, we represent the suffix in an array. If we need to add a new C call frame, we freeze the suffix, and we add another structure to represent callee-saved registers. These are just copies of the 5 callee-saved registers. If we call an assembly function, or we enter an assembly block, we use another array to represent the new stack suffix, and so on. We shall remark that contiguous C call frames are not separated: the only thing that matters is to restore the original state of callee-saved registers, thus subsequent C functions does not change anything.

We see that this abstraction does not make a special place for callee-saved registers in assembly functions, as there is for callee-saved registers in C functions. The moral reason is that there are many ways to save callee-saved registers, and if there are some backups, there are in the explicit segment of the stack. Some callee-saved registers may not be saved at all if there are unused, and thus, unchanged. The technical reason is that we only need to check that the value of callee-saved registers is the same when we enter the assembly function, and when we exit it, and the development version of Astrée provides a way to check that a variable has the same value at two given program points. This feature allows easy checking that calling convention requirements are fulfilled by assembly functions.

Because of technical constraints, we have to allocate statically all arrays used for explicit stack segments and all structures used to remember callee-saved registers in C call frames. This issue was already mentioned previously. We already discussed how to choose an appropriate size for explicit stack segments. Here, there is another difficulty: choosing the right number of available arrays and structure to back up registers, that is, the number of alternation between C and assembly call frames.

For that question, we found no satisfactory heuristic. We make this quantity a manually parameterizable static integer. The point is that there are complicated control flow, alternating C and assembly, but they are relatively exceptional: most of the control flow is pure C. For instance, to perform a syscall, an application program in C calls a C function that is part of system library. Ultimately, the syscall needs assembly to be triggered. From the other side, the handler of the syscall must be in assembly as well to handle precisely the content of the stack. But of course, we do not want to perform the requested operation in pure assembly, it would be painful. The control goes back to C as soon as possible. According to syscall code, the correct handler is called, and we are again in a C world. At some point, the syscall may require some assembly blocks to perform some low-level operations, but hardly more. Overall, we need two call frames of each kind, in this example.

Most execution paths require not even that many call frames. Overall, we find 3 to be a good number. For development and debugging purpose, we used only 1 call frame of each kind: fewer variables implies shorter logs, easier to read. It turns out that 1 of each kind is enough in most cases. The reason is that often, assembly functions are leaves in the call graph, or they call only other assembly functions.

15.2.3 Formalization

Once again, this is a multistage abstraction: we abstract a stack into a succession of C and assembly call frames, and each call frame is abstracted using an underlying domain.

Concretization is not direct though: this abstraction is not about directly abstracting the shape of the stack, but being able to restore registers correctly. But there is a myriad of possibilities to restore the registers: they can be untouched, they can be saved on the stack, they can be saved in a structure on the heap with a pointer in the stack to this structure, or to another pointer pointing to this structure. . . .

For this abstraction, the concrete domain is not the simple memory state from the naive standard semantics. We need to use an enriched semantics. An enriched non-standard semantics is an extension of the standard semantics that carries some additional pieces of information that do not exist as is. But since these pieces of information come from the standard semantics, enriched semantics are not more precise, but they may be easier to abstract precisely as they can distill relevant data that would be dissolved otherwise. The Part IV "A Reduced Product with Ghost Variables" (page 229) is about abstracting such semantics modularly with an extensive generality that we do not entirely need for now.

We can give a simple example of enriched semantics. From the standard semantics of C (or Asclepius (see chapter 4 "A Minimal C-like Language" (page 43)), or any similar language), we track the number of currently nested functions: we add a single integer to states, which is incremented on calls and decremented on returns. This integer does not carry additional information, but it makes it explicit and easier to abstract. If we are interested of the number of nested functions (for instance, for precise stack usage analysis, to prevent stack overflows), this semantics may be very useful.

In our case, the information we need to make explicit is the state of callee-saved registers along with each C call frame that follows an assembly call frame. The concrete domain is thus not only a stack, but a stack divided into segments, with half of them being annotated with 5 integers (for the 5 callee-saved registers). Formally, the concrete domain has type

$$\mathcal{D} := \left(\Sigma^\star \cdot \left(\Sigma^\star \times \mathbb{Z}^5\right)\right)^\star \cdot \Sigma^\star$$

The first component includes the unknown prefix and the first explicit segment. We can remark that the semantic of Panacea already took the registers to save into account: in section 8.2 "Memory Model" (page 103), the stack contains copies of the enriched heap exactly for this purpose[‡]. Thus, from the point of view of the semantics introduced in chapter 8 "Semantics of Mixed C and Assembly" (page 99), this model of stack is not enriched. This is because the concrete semantics is itself enriched compared to the actual behavior of the computer, in order to be able to detect errors like calling convention violation. In a real program, we do not have copies of register to check that they are properly managed: such a violation would not be detected, and the program might behave erratically[§].

Yet, even if both stack models store copies of registers, the type of stacks here and in the definition of the semantics of Panacea are not the same: in chapter 8 "Semantics of Mixed C and Assembly" (page 99) we were interested in a stack that contains values we can indeed access (explicit segments of the stack), data useful for control flow (e.g. return addresses) and the stack of local environments. In this chapter, we only want

[‡]We do not really need the whole enriched heap, but it makes the formalization lighter and mathematically, copying huge structure is free. Here, since we are closer to the implementation, we are interested into saving exactly what is necessary.

[§]But, probably, deterministically from a lower level point of view.

to abstract the data part. The question of return addresses is treated as a part of the problem of control flow (and is explained further in chapter 17 "Control Statements and Modification of the Iterator" (page 207)), and local environments are useless since we forbid recursive functions.

Let $\mathcal{D}^\sharp_\mathcal{S}$ a domain that abstracts m stack segments and n integers for any $(m, n) \in \mathbb{N}^2$, possibly in a relational way, with concretization

$$\gamma_\mathcal{S} : \mathcal{D}^\sharp_\mathcal{S} \to \bigcup_{m\in\mathbb{N}} \bigcup_{n\in\mathbb{N}} \mathcal{P}\left(\Sigma^{\star m} \times \mathbb{Z}^n\right)$$

We define the abstract domain $\mathcal{D}^\sharp$ the subset of $\mathcal{D}^\sharp_\mathcal{S}$ whose concretization contains $n+1$ stack segments and $5n$ integers, for any $n \in \mathbb{N}$, that is

$$\mathcal{D}^\sharp := \left\{ s^\sharp \in \mathcal{D}^\sharp_\mathcal{S} \,\middle|\, \exists n \in \mathbb{N} : s^\sharp \in \mathcal{P}\left(\Sigma^{\star n+1} \times \mathbb{Z}^{5n}\right) \right\}$$

and the concretization

$$\begin{aligned} \gamma : \mathcal{D}^\sharp &\to \mathcal{P}(\mathcal{D}) \\ s^\sharp &\mapsto \left\{ p \cdot \left(\prod_{i=1}^{n} s_i \cdot r_i \right) \cdot s_{n+1} \,\middle|\, \begin{array}{l} n \in \mathbb{N}, p \in \Sigma^\star, \\ (s_i)_{i\in[\![1,n+1]\!]} \in \Sigma^{\star n+1}, \\ (r_i)_{i\in[\![1,n]\!]} \in \left(\Sigma^\star \times \mathbb{Z}^5\right)^n : \\ \exists (q_i)_{i\in[\![1,n]\!]} \in \Sigma^{\star n} : \\ \exists (w_i)_{i\in[\![1,5n]\!]} \in \mathbb{Z}^{5n} : \\ \left((s_i)_{i\in[\![1,n+1]\!]}, (w_i)_{i\in[\![1,5n]\!]} \right) \in \gamma_\mathcal{S}\left(s^\sharp\right), \\ \forall i \in [\![1,n]\!], r_i = \left(q_i, \left(w_{5(i-1)+j} \right)_{j\in[\![1,5]\!]} \right) \end{array} \right\} \end{aligned}$$

This is a glorified way to express that the stack is made of an unknown prefix followed by an alternation of explicit and unknown segments. The latter are annotated with 5 integers (for the values of each callee-saved register).

The underlying abstraction ($\mathcal{D}^\sharp_\mathcal{S}$) is left to an appropriate domain. It must be able to abstract arrays and integer values. It is interesting to see that we do not change the type of abstract values, but we interpret them differently.

15.2.4 Discussion

This abstraction matches the master requirements (precision and performance). The main remaining question is whether it is future-proof: does it work on other kind of software or can it be easily improved to work?

Assembly is mixed with C code to do things that cannot be expressed in C, either because it is platform-dependent, like what OSes do, or for performance reasons. When assembly is used for performance, it seems to be rarely in the middle of C control flow, they rather are leaf-functions or just blocks in C functions.

In OSes, system developers try to use assembly in these simple ways: this is the cleanest, but it is not always possible. Due to legacy code or weird architecture, there are always some weird twisted control flows, especially at startup or when we need to interact with low-level features, like exceptions or call gates. Such codes are often particular enough to be alone in their category, thus, we cannot tell if our abstraction will work without being confronted to such problem. Nevertheless, we can try to comfort ourselves with the same rationale as in subsection 15.2.2 "The Idea" (page 190): assembly is needed at some point but there are probably not that many alternations. Indeed, we have not found code that need unbounded alternation of C and assembly, so far. Moreover, such a code probably uses recursive functions, that would imply very heavy modifications in ASTRÉE, anyway.

Another good point of this abstraction is that it relies on the underlying domain for most abstract transfer functions, thus it has all the power of the underlying domain and it is easy to implement.

15.3 Implementation

For abstracting a single assembly suffix, or alternation of call frames, we need an underlying domain to represent array and integer variables. In the hierarchy presented in section 14.4 "Implementation" (page 178), and especially Figure 14.1 "New structure of the composite domain of ASTRÉE" (page 179), we can see that we need to be above struct domain since, as explained before, under this domain, variables are replaced by physical cells. Just like for register abstraction, we want to be as low in the hierarchy as possible, to make the domain as simple as possible. So, just like registers, stack is abstracted by the new CPU domain.

This position has another advantage: it is under parallel domain which receives from struct domain information about modified variables so as to determine the effect of each thread to others (see [Min13; Min14]). As the CPU domain forward these pieces of information, it can read it to guarantee that deep assembly call frames (explicit segments that are not the current suffix) are not modified.

Chapter 16

Other Non-control Statements

So far, we have treated most kinds of statements: arithmetic, logic and stack operations. There are other kinds of instructions that are quite rare overall, but that are important nevertheless. We are still not concerned about control flow statements: this is the matter of chapter 17 “Control Statements and Modification of the Iterator” (page 207).

The most important category of remaining statements (excluding control-flow) are system instructions. They are a small group of very heterogeneous instructions. We present how we manage to implement their semantics with minimum pain.

Following sections are about other kinds of statements. Also, we will see how to handle dynamic code. This is not a distinguished type of statements, but they require a special preprocessing.

16.1 System Statements

16.1.1 Scope

System instructions are central in the control of memory protection features of x86 processors. These features are used to guarantee memory isolation of each process, especially, to isolate the privileged world from the user world.

Because of time constraints, we are not able to handle all kinds of system instructions. Especially, paging requires memory abstractions that are not designed yet (see section 26.1 “Paging” (page 327)). Moreover, instructions that are supported are only in the settings of our industrial application.

More precisely, we are handling `LGDT` (load GDT, see subsubsubsection 2.3.2.2.3 “Global Descriptor Table” (page 23)), `LIDT` (load IDT, see subsection 2.4.3 “Interrupts and Exceptions” (page 32)), `LTR` (load task register, see subsection 2.5.2 “Task Register” (page 35)), task switches (from control transfer statements, see section 2.5 “Tasks” (page 34)) and loading segment registers (see subsubsection 2.2.1.3 “Segment Registers” (page 12)). Globally, we are mainly concerned with segmentation, and especially a particular kind of segmentation: flat memory model. We want to prove that this model is correctly implemented, with right access flags, and that it is fully set up before using it.

We explicitly do not support other segmentation model as we did not need them and most systems use a flat memory model nowadays (see subsubsubsection 2.3.2.2.2 “Segmenting the Linear Address Space” (page 21)).

16.1.2 System Registers and CPU State

Most system instructions modify or use system registers: we had to model them. Of course, we add control registers (CR0, CR2, CR3 and CR4 as 32-bit variables) and segment registers (CS, SS, DS, ES, FS and GS as 16-bit variables). We also need some more specifics registers: GDTR, IDTR and TR. GDTR and IDTR are split into two parts: their limits (a 16-bit variable) and their base (a 32-bit variable) (see subsubsubsection 2.3.2.2.3 “Global Descriptor Table” (page 23)). They are all added into the CPU domain (see Figure 14.1 “New structure of the composite domain of ASTRÉE” (page 179)), and represented using underlying domains.

But this is not enough. As explained in subsubsubsection 2.3.2.2.4 “Segment Selectors” (page 23) and subsection 2.5.2 “Task Register” (page 35), registers may have a hidden part to store the GDT entry they point to, so as to improve performances: GDT lies in the memory, which is slow to read; by storing appropriate pieces of information into hidden registers, the processor speeds up access to these GDT entries without allowing software to modify this cached value. Yet, we are not modeling hidden parts of segment registers; since we are limited to a single kind of segmentation model, we know what hidden parts shall contain. When a segment register is loaded, we check that the corresponding GDT entry matches the conditions and if it does, it has the expected shape, thus we do not need to copy it.

There is no such restrictions about TR, thus, we need to represent the hidden part of TR. Since a GDT entry is 8-byte long, the hidden part of TR is modeled as two 32-bit variables.

The OS that we are analyzing does not start with the processor, but it is launched by a boot loader, as well as most Linux installations use the boot loader GRUB. The boot protocol is even very close; when the boot loader launches the OS, segmentation is already enabled but paging is not. We assume a flat memory model with access rights that allow execution, otherwise, the OS would not start. Even if the model given by the boot loader is the same as the one we want to set up, we cannot just rely on initial segments. First, the boot loader is not known to provide user-level segments. Moreover, initial segments allow execution in ring 0, but we are not sure about the access rights, for instance, we do not have the guarantee that these segments do not allow execution of system code from ring 3; this is not a problem as long as there is no user-world task, but it should be fixed before creating them. Finally, GDT does not only store segments, but also other kinds of system descriptors that have to be set by the OS, like the IDT (see subsection 2.4.3 “Interrupts and Exceptions” (page 32)). Since we do not know *a priori* the shape of the GDT the boot loader built, we shall rewrite everything.

In our initialization scenario, we need to set the GDT (using a `LGDT` instruction) before loading segment registers. Since the original GDT is unknown, loading a segment register before setting up the new GDT would give an unknown result, and possibly an error (if outside GDT limit). By doing so, we reject concrete traces that are legal but that cannot be abstracted meaningfully, since each subsequent memory access could do anything. These unwanted traces do not exist in the analyzed OS, and are not sensible.

To remember whether the new GDT has been set, the new CPU domain stores a Boolean value that tells whether a `LGDT` statement has been executed. This is an important, synchronizing event. We do not allow binary operations (like join or meet) between states with unequal values for this Boolean variable. Once again, this restriction rejects legal traces that cannot appear in any reasonable code.

16.1.3 Microcode

Until now, we only talked about system data structures and registers, but not the semantics of system instructions. Their behavior is described with pseudo code in [Int20], their components are quite simple, but the complexity comes from the length of this pseudo code. Even if the expressions and elementary statements are simple, they are very long to write in the internal format of statements in ASTRÉE: we need to specify explicitly the full type at each node of expressions, fill some fields that are not all relevant. . . . Moreover, in the abstract, testing a condition is much more complicated: we need to guard by the negation of the condition and test if the result is $\bot$. Translating the pseudo code of simple instructions ($\sim$ 20 lines of pseudo code) results in more than 150 very long lines ($\sim$ 300 columns) in OCAML code. Writing complex system instruction whose pseudo-code is more than 4-page long seems unfeasible without a great risk of errors.

In practice, statements in system instruction pseudo-codes are a very particular subset of internal statements. From this remark, we defined a language, as OCAML datatype, that is merely a shorthand for writing internal statements.

Let us describe a simplified version of this language, starting by the structure of expressions and lvalues. We ignore aspects that are only shortcuts or debugging features, though the weary developer will not miss appreciating them.

```
⟨size⟩        ::= 'BYTE' | 'WORD' | 'DWORD'

⟨signedness⟩  ::= 'SIGNED' | 'UNSIGNED'

⟨arith2⟩      ::= '+' | '-' | '*' | '/' | '%'
                |  '&' | '|' | '>>' | '<<'

⟨expr⟩        ::= 'Name' '('⟨expr_name⟩')'
                |  'Deref' '(' ⟨expr⟩ ',' ⟨int⟩ ',' ⟨size⟩ ',' ⟨signedness⟩ ')'
                |  'Bits' '(' ⟨expr⟩ ',' ⟨int⟩ ',' ⟨int⟩ ')'
                |  'Expr' '(' ⟨expr⟩ ',' ⟨expr⟩ ')'
                |  'Const' '(' ⟨int⟩ ')'
                |  'Arith2' '(' ⟨expr⟩ ',' ⟨arith2⟩ ',' ⟨expr⟩ ')'
                |  'Expr' '(' ⟨internal_expr⟩ ')'
```

⟨*lvalue*⟩ ::= ‘`LName`’ ‘`(`’ ⟨*lvalue_name*⟩ ‘`)`’
| ‘`LPtr`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’
| ‘`LValue`’ ‘`(`’ ⟨*internal_lvalue*⟩ ‘`)`’

Leaves in expression trees are either `Name(_)` or `Expr(_)`. The latter is used to introduce an expression of the internal format of ASTRÉE. We use it mainly to introduce variables. The former type of leaves contains a name that is an alias for another ⟨*expr*⟩: this language is executed with an environment that stores map of aliases to share complex expressions. These maps also contain the operands of assembly instructions, usually aliased by names `dest` and `src`. `Deref(e, o, s, u)` is used to dereference a number of bytes specified by `s` ($\texttt{BYTE} = 1$, $\texttt{WORD} = 2$ and $\texttt{DWORD} = 4$), at address $\texttt{e} + \texttt{o}$ (`e` is the base pointer and `o` an integer offset in bytes), interpreted as an integer with signedness given by `u`. `Bits(e, f, n)` is used to select `n` bits from `e`, starting at bit number `f`. Other nodes are quite clear.

⟨*cmp_op*⟩ ::= ‘`==`’ | ‘`!=`’ | ‘`<`’ | ‘`<=`’ | ‘`>`’ | ‘`>=`’

⟨*cond*⟩ ::= ‘`Cmp`’ ‘`(`’ ⟨*expr*⟩ ‘`,`’ ⟨*cmp_op*⟩ ‘`,`’ ⟨*expr*⟩ ‘`)`’
| ‘`And`’ ‘`(`’ { ⟨*cond*⟩ } ‘`)`’
| ‘`Or`’ ‘`(`’ { ⟨*cond*⟩ } ‘`)`’

The only surprising point in the syntax of conditions is that conjunctions and disjunctions are not limited to 2 operands. We can also note there is no “`Not`” node, since it is a source of mistakes: a correct approximation of “`Not(a)`” shows whether `a` may be false, but we are often more interested in the case where `a` is definitely false*.

⟨*static_cond*⟩ ::= ‘`Static_bool`’ ‘`(`’ ⟨*bool_name*⟩ ‘`)`’
| ‘`Static_and`’ ‘`(`’ { ⟨*static_cond*⟩ } ‘`)`’
| ‘`Static_or`’ ‘`(`’ { ⟨*static_cond*⟩ } ‘`)`’
| ‘`Static_not`’ ‘`(`’ ⟨*static_cond*⟩ ‘`)`’

Static conditions are conditions whose leaves are names, that are aliases in the environment for Boolean values. They are very useful to parameterize the semantics of instructions: when two instructions have very close semantics, they can share the same code with a static condition to branch between two variants, thanks to a Boolean value that is set or cleared depending on the instruction we want to run. For instance, codes for task switches have a lot in common, but differ depending on whether the task switch comes from a call, a jump or a return. The microcode is common to all these flavors, but it contains static conditions to enable statements only when appropriate. Here, we allow a “not” node since these Boolean values are not part of an abstract state.

⟨*exception*⟩ ::= Custom
| ⟨*intel_exception*⟩

*Typically, when `a` is an error condition: we do not want to know whether the execution may be error-free, but when it is definitely error-free.

```
⟨stat⟩          ::= 'Sequence' '(' { ⟨stat⟩ }')'
                  | 'Assert' '(' ⟨cond⟩ ',' ⟨exception⟩ ')'
                  | 'If' '(' ⟨cond⟩ ',' ⟨stat⟩ ',' ⟨stat⟩ ')'
                  | 'StaticIf' '(' ⟨static_cond⟩ ',' ⟨stat⟩ ',' ⟨stat⟩ ')'
                  | 'Switch' '(' { '('⟨cond⟩ ',' ⟨stat⟩ ')' ',' } ⟨stat⟩ ')'
                  | 'StaticSwitch' '(' { '(' ⟨static_cond⟩ ',' ⟨stat⟩ ')' ',' } ⟨stat⟩ ')'
                  | 'AddAlias' '(' ⟨expr_name⟩ ',' ⟨expr⟩ ')'
                  | 'Call' '(' ⟨fun_name⟩ ')'
                  | 'Assign' '(' ⟨lvalue⟩ ',' ⟨expr⟩ ')'
                  | 'Fail' '(' ⟨exception⟩ ')'
                  | 'Skip' | 'Ignore' | 'Absurd'
```

We define two kinds of exceptions: standard exceptions planned by the semantics, and a custom exception for additional checks that are relevant only for our analysis. `Fail(e)` raises exception `e`. `Assert(c, e)` evaluates condition `c` and raises the exception `e` if it can be false. It can be rewritten as `If(c, Skip, Fail(e))`, where `Skip` is a statement that does nothing, like `Sequence()`. `Call(f)` runs the statement aliased by `f` in the current environment and then comes back to the next statement. `AddAlias(name, e)` adds `name` as an alias for `e` in the current environment. `Switch` and `StaticSwitch` take a list of pair of conditions and statement, with an additional statement for the default case. The executed statement is the first whose condition is true. `Ignore` is used to replace the current abstract state by $\bot$. It is useful as the default case of an exhaustive `Switch`[†]. `Absurd` asserts that the current abstract state should be $\bot$, otherwise, there is an error in the microcode[‡]. It is useful as default case of a `StaticSwitch`.

Overall, in addition to the abstract state that we are processing, a memory state in the microcode is made of 3 maps: names to expressions, names to lvalues, and names to (static) Boolean. The microcode of an instruction is a statement with a map that associates a statement to each $\langle fun\_name \rangle$. Thanks to this language, we can write formally the semantics of instruction is a form very close to the pseudo-code we find in INTEL's manuals. Unfortunately, since the pseudo-code is not very formal (in particular, it contains many natural language descriptions), we cannot automatize the translation from pseudo-code to our microcode.

16.2 Dynamic Code

Dynamic code is not a recommended programming style, but we found some in real world program. Sometimes because it is a light fix avoiding a huge restructuring, or because technical constraints make it difficult (or inefficient) to do otherwise. It would be foolish to believe that it is an old habit that does not exist anymore. Dynamic code is introduced because it was necessary, not because it was fashionable. We should not

[†]Indeed, because of imprecision, the default case of an exhaustive switch may not be $\bot$.

[‡]It should not be used to raise an alarm if the abstract state does not satisfy a property. We use `Fail` or `Assert` for that purpose. `Absurd` is used when a program point in the microcode is not reachable.

say that modern OSes do not have dynamic code, but that they do not have dynamic code *yet*.

Since it exists in real-life code, we shall support it in our analyses, at least partially.

16.2.1 A Simple Strategy

Typically, dynamic code is written into an array, then a jump statement transfer the control at the beginning of the array. A first approach is to concretize the content of the array, disassemble the result to get the instructions. For efficiency concerns, we should take some precautions in this process: concretization may return a very large set. We want to detect this early enough to raise an alarm. Even if the concretization of each byte gives a reasonably small set, Cartesian product may still be huge. If both these operations succeed by giving a reasonably small set of possibilities, we can forward the result to a disassembler.

Disassembling is performed by AMICAL (see chapter 9 "AMICAL" (page 121)). Each sequence of byte is translated into an instruction and a sequence of remaining bytes, by iterating, we get all the instructions to execute. Possible sequences of instructions should be run in parallel, and abstract states joined at the end. We shall also remark that such dynamic code should contain a jump statement otherwise, we would reach the end of the array, at which point, fetching the next instruction is an illegal memory access, at least, in our restricted semantics.

This strategy seemed to be sufficient for analyzed sections of the code. But it has a lot of limitations, especially about the size of concretization.

16.2.2 Improving Amical

Some instructions may be very simple, but leading to concretizations with very poor precision. For instance, we dynamically write byte E9H in an array, at address a. E9H is the opcode for a relative near jump with a 32-bit offset, which is interpreted relatively to the next instruction, i.e. the end of the current instruction. The opcode shall be followed by 4 bytes. The end of the instruction will be at address $a + 5$. This instruction will transfer the control to address $a + 5 + o$ where o is the specified offset. To jump to a given label l, we need that $a + 5 + o = l$, i.e. $o = l - a - 5$. Let us write o on 32 bits after the opcode. We get a 5-byte statement that jumps unconditionally to l. An example of such code is shown on Listing 16.1.

Address of labels are often unknown, and concretizing a byte sequence such as the example would give a perfectly unknown offset, while a symbolic domain can have a very precise abstract value. The reason is that the real byte sequence does not matter; we only need to know a more symbolic property on the value defined by the byte sequence.

To handle correctly such cases, we need a less eager disassembling. We disassemble just enough bytes to get the prefixes and the opcode. With this information, we can directly use parts of the array as immediate operands or displacements, without having to concretize them. For memory and register operands, we still need to concretize the Mod-R/M and SIB bytes. In Listing 16.1, we would start by concretizing and disassembling

```c
char a[5];
extern void l(void);  // Necessary to take the address of l in C code.
void g() {
    asm {
        l:
        ; ...
    }
}
void f() {
    int offset = &l - &a - 5;
    a[0] = 0xe9;
    *((int*)(a + 1)) = offset;
    asm {
        jmp a   ; Runs the instruction in a that jumps to l
    }
}
```

Listing 16.1: Simple dynamic code with problematic handling

one byte: we get a precise value, from which we can tell that the instruction is a simple relative jump, whose offset is contained in the 4 following bytes: `*((int*)(a + 1))`. A domain such as ASTRÉE's struct is able to remember that these 4 bytes make a single cell whose content can be represented using any underlying domain (like linear combinations domain).

This solution is not implemented yet but it will probably be as it is required to analyze code such as the snippet of Listing 16.1, that exists in not-yet-analyzed parts of our industrial application.

There is another point on which AMICAL can be improved. When we need the instructions dynamically written in an array, we said that we can concretize each cell to get all possible sequences, which may lead to a huge set of sequence (the Cartesian product of each cell). A solution is to make AMICAL to accept sequences of set of potential bytes (rather than a single precise sequence): by consuming each cell consecutively, AMICAL can reject huge sets of illegal sequences. For instance, if a prefix appears twice, the instruction is not valid, whatever the following. There is a limitation: this method does not allow relations between bytes. A satisfying approach may be to apply this method with a coarser granularity, by providing independent blocks of bytes[§].

[§]Typically, an offset and an immediate operand are likely not to be related, and they can be seen as two independent 4-byte block. Of course, there are relations between the bytes of each of these values.

16.3 Floating Point Instructions

Early processors of the x86 family can use a coprocessor for floating-point operations. The first was designed to be used with the 8086 and was called 8087. Consequently, this family of coprocessors took the name of x87. Following x86 processors had their x87 coprocessors. The 80387, math coprocessor of the 80386, was the first fully compliant with the IEEE 754 standard. The 80487 was already like modern processor: it contains a complete processor and, when installed, it deactivates the main 80486 to run all instructions by itself. Nevertheless, it still needed the 80486 to be present. Math coprocessors did not last very longer as the 80587 was the last x87. Since then, the floating-point unit (FPU) is always integrated in the main CPU.

FPU instructions are mostly straightforward, and not very different from C's math library. The main difficulty in the FPU is the memory model: it uses a cyclic stack of 8 registers. Registers are not named absolutely, but relatively to the current top. This very particular structure is not as straightforward to abstract as other registers.

Since FPU instructions are not very relevant in the context of an OS, this problem has been decided to have low priority and is not treated yet. With the next section and paging, they form the main categories of statements that are not handled and would probably require heavy work.

16.4 Input/Output

In addition to access to the RAM by the memory bus, processors of x86 family can transfer data through input/output (I/O) ports. Ports are created by the hardware circuitry from processor pins and then configured to access external devices. By I/O ports, the processor can control these external pieces of hardware, receive data from, or send data to external storage. . . .

I/O ports are controlled by their own mechanisms of privilege levels that we will not explain in detail. Receiving and sending data is done respectively by instructions `IN` and `OUT`, and their string instruction counterparts `INS` and `OUTS`. These instructions have two operands: the port number (always stored in general-purpose register DX), and the register (always AL/AX/EAX for simple versions) or memory (always `[ES:(E)DI]` for string versions).

Modeling correctly these instructions requires knowing hardware circuitry around the processor and specifications of devices with which we are communicating.

In our application, we do not have all these hypotheses. But, in analyzed parts of the target OS, I/O instructions are used in a very restricted way: to set up an external timer to trigger the scheduler (see section 2.7 “Boot” (page 37)). For want of a precise specification of this timer, we could not make a good abstraction. Instead, we choose to compare I/O port accesses with an expected scenario.

A communication scenario is a (usually finite) sequence of tuples made of a value of a two-valued type (whether it is an input or an output), a port number and an integer. This is scenario is an *a priori* knowledge. The semantics of I/O instructions with respect to a scenario are quite direct. For `IN`, we check that the next step in the scenario is indeed an input on the appropriate port, the result is set to the integer value planned by the scenario, and we pop this step from the scenario. For `OUT`, we check that the scenario expects an output on the given port, then we check that the expected output is indeed the value of the operand. Likewise, we remove the executed step. For our application, at the end of initialization, we expect that the remaining scenario is empty. For instance, Listing 16.2 shows a code that reads 4 bytes from I/O port 42, adds 1, and outputs the result to port 43. This code matches any communication scenario that expects an input from port 42 of a value n and an output to port 43 of a value $n+1$, just like the scenario shown in Table 16.1. Indeed, when we run the first line of the code, we check that we indeed read a value from port 42, and EAX is assigned the value 5. We remove the first line of the scenario. The second line of the code has no effect on the communication scenario. The third line is an I/O instruction: we check that it matches the requirement of the first line of remaining scenario. It should be an output to port 43 of the value 6, which is correct. At the end of the code, the remaining scenario is empty, thus the initial scenario is satisfied.

```
in EAX, 42    ; Input doubleword from I/O port 42 into EAX
add EAX, 1
out 43, EAX   ; Output EAX to I/O port 43
```

Listing 16.2: Simple communicating program

Input/Output	Port	Value
Input	42	5
Output	43	6

Table 16.1: Simple communication scenario

It is easy to design an abstract domain to check a communication scenario: it simply holds an integer that is the progress in the scenario. We reject the union of two states with unequal progress: like setting the GDT, these communications are important synchronizing events. If such a union occurs, the abstract state becomes $\top$ and is removed with an alarm. During the abstract execution of an `IN` or `OUT` instruction, if a check does not hold, we raise an alarm and remove these traces. If such an instruction is executed when the communication scenario reached the end, we also raise an alarm.

Such a basic check is useful to verify that all execution traces go indeed through the same communication process. In addition, specifying the communication process can be useful to check that is match the specification of the external device, either manually, or by another analysis, centered on the device. Such a modular analysis requires to write the communication scenario by hand.

A very simplistic scenario has been hard coded in the CPU domain to study the feasibility of this method. Results were encouraging, and we trust this method can indeed be used for the initialization sequence of our application. This proof of concept was then removed as it is in fact useless, and tried to check the toy scenario on every program. A proper implementation requires to allow the scenario to be specified as external configuration or annotations in the source code. Due to time constraints, this is not implemented for now: making the scenario parameterizable is a much bigger work than implementing the transfer functions for I/O instructions.

To check the initialization sequence of our study case, we will need to support I/O instructions, but we need to know the expected scenario. Since it depends on external hardware, which was not communicated precisely enough by our industrial partner, we cannot design the scenario. We may still craft it from what the study case does, but building the specification from the implementation we want to check has very little interest. It can only check that communications are indeed synchronizing events (communications cannot be skipped or reordered). Considering the estimated amount of work for the implementation, we deemed that this weak property was not interesting enough to spend more time on it for now.

We shall still note that the communication scenario does not model entirely the hardware. Indeed, it does not take timing into account: each time a line of the scenario is consumed, the next is ready. This is not actually true: external devices may need some time to provide the next value. Some communications are indeed timing based (and thus, we need a model with a notion of clock), but some are more asynchronous: an interrupt is triggered when there is something to read. For this kind of communication, we just need to remember that data are available as soon as the interrupt is triggered. This communication method seems safer, but unfortunately, this is not what our study case does during initialization. Indeed, such interrupt allows suspending most computations, which is dangerous during system set up; for that reason, these interrupts are often disabled during initialization.

16.5 String Instructions

Some instructions can have a prefix to repeat them automatically; they have a mnemonic that ends with a `S` (`MOVS`, `CMPS`, `SCAS`, `LODS`, `STOS`, `INS` and `OUTS`). They are called string instructions since they treat contiguous blocks of data. They all use `DS:ESI` as pointer to the source operand and `ES:EDI` as pointer to the destination operand (if applicable).

There are 3 repetition prefixes: `REP`, `REPZ` (and the alias `REPE`) and `REPNZ` (and the alias `REPNE`). `REP` repeats the string instruction as long as ECX $\neq 0$, and ECX is decremented at each iteration. `REPZ` (repeat if zero) (or `REPE`: repeat if equal) also decrements ECX, but the condition is ECX $\neq 0 \wedge$ZF $= 1$. This condition allows repeating the operation at most ECX times, but stopping as soon as the ZF flag is cleared, that is, the result of the string operation is not zero (or if both operands are not equal, depending on the instruction). `REPZ` (repeat if non-zero) (or `REPE`: repeat if not equal) does the same, but the condition is ECX $\neq 0 \wedge$ ZF $= 0$: it repeats the given instruction as long as the result is not zero (or operands are not equal).

The main benefit, other than readability, of using a string instruction rather than a classical loop is performance. Otherwise, especially from a semantic point of view, it is equivalent to classical loop statement. Translating string instructions into a for-style loop whose body is an elementary instruction is probably good enough, both in performance and precision.

For now, string instructions are not implemented yet in ASTRÉE, due to time constraints and, as they are not present in analyzed sections of the target OS, they have not been judged to have high priority.

Chapter 17

Control Statements and Modification of the Iterator

UNTIL NOW, we were interested in most categories of statements expect control transfer statements. Yet, without them, control flow is totally linear and programs are mostly uninteresting. Such statements allow programming languages to be TURING-complete, as they hold the power to select and repeat operations. The main drawback is that they often make analyses difficult.

Assembly control transfer statements may not seem particularly difficult to handle. The complexity comes from the interaction with C, since they have a very different point of view on control flow. In C, control flow structures exist even in the syntax. The only kind of transfer that is not well parenthesized are goto-statements. Yet, their destination is static (in standard C) and they only can reach labels in the current function. Contrariwise, assembly has no syntactic construct to reflect control flow, and control flow is subjected to very few rules, in particular, destination of control transfer statements are not always statically known.

As most control flow is done in C code, we need to keep all the precision we have. An even better property is to make it so that pure C code analysis is unchanged. While assembly control flow is rare, it has a significant importance and it is not limited to simple cases. On the contrary, regular control flow is rather implemented in C as much as possible. The remaining cases are mostly difficult ones.

17.1 Computing the Destination

We already mentioned the problem of the translation between code pointers and program points. A code pointer is an address in memory that designates a point into the code (and not a variable). The processor works with code pointers, since code, like data, is just a sequence of bytes, without structure. There is no *a priori* separation between consecutive statements*. Program points are logical points in a program, especially visible from the source code. They are positions between two statements. This is what semantics and analyzers work with. A destination is designed by a code pointer, but to continue the analysis, we need to translate it into program points. In x86, from the binary encoding and without context, we cannot tell where program points are. We need at least to know where the first instruction starts. Worse, not respecting these program points is allowed: the processor has no knowledge about the boundary of instructions. Reading instruction with such a shift is a hacker technique, or used in very advanced compression. In mainstream code, it is not a desirable behavior. When we have the correspondence between the binary sequence and the assembly source code, usually, we do not allow jumps whose destination is not between two statement. This is the case here: even if the processor would not complain, we forbid such jumps; when they happen, an alarm should be emitted.

Yet, this translation is not trivial. In section 5.5 “A More Symbolic Semantics” (page 77), we needed the start address of the code in memory, and we used an oracle giving the length of each instruction to be able to resolve code pointers into program points. This is the first difficulty: we do not know precisely the length of each assembly instruction. The second imprecision source is more intrinsic to abstract interpretation: a code pointer can be imprecise.

Possible sizes of assembly statements are computed statically using Amical (see chapter 9 “Amical” (page 121)). From these individual sizes we compute a function that, given a program point and an offset relative to the program point, computes possible program points corresponding to the code pointer, and whether it can be an illegal pointer. A code pointer is illegal if it is before the beginning of the block, after the end, or if it may be in the middle of an instruction. Just like data in C, assembly blocks are supposed to live in separate universes: as we cannot craft a pointer to a variable from the address of another variable, we cannot designate a program point in an assembly block from a program point in another block and an integer offset. For obvious performance reasons, this function memoizes even intermediate computations. Formally, let $b \in \mathbb{N}$ be the number of assembly blocks in the program and for all $i \in [\![1, b]\!]$, let $l_i \in \mathbb{N}$ be the number of assembly statements in the i^{th} block. For all $i \in [\![1, b]\!]$ and $j \in [\![1, l_i]\!]$, let $S_{i,j} \subseteq \mathbb{N}^*$ be the set of possible lengths of the j^{th} instruction of the i^{th} block. The sets $S_{i,j}$ are computed by Amical. A program point is identified by a pair $(i, j) \in \mathbb{N}^2$ where i is the number of the block and j the number of instructions before the program point. The first program point of a block i is thus $(i, 0)$ and the last is

*At least in x86. In some architectures, especially RISC (reduced instruction set computer), instructions have a constant size and are aligned.

(i, l_i). We define the function

$$\begin{aligned}
\mathrm{pp} : \mathbb{N}^* \times \mathbb{N} \times \mathbb{Z} &\to \mathcal{P}\left(\mathbb{N}^* \times \mathbb{N}\right) \\
(i, j, o) &\mapsto \left\{ (i, j') \in \mathbb{N}^* \times \mathbb{N} \;\middle|\; \begin{array}{l} \exists (s_k)_{k \in [\![j+1, j']\!]} \in \mathbb{N}^{j'-j} : \\ o \geqslant 0, j' \geqslant j, \\ \forall k \in \left[\!\left[j+1, j' \right]\!\right], s_k \in S_{i,k}, \\ \displaystyle\sum_{k=j+1}^{j'} s_k = o \end{array} \right\} \\
&\cup \left\{ (i, j') \in \mathbb{N}^* \times \mathbb{N} \;\middle|\; \begin{array}{l} \exists (s_k)_{k \in [\![j'+1, j]\!]} \in \mathbb{N}^{j-j'} : \\ o \leqslant 0, j' \leqslant j, \\ \forall k \in \left[\!\left[j'+1, j \right]\!\right], s_k \in S_{i,k}, \\ \displaystyle\sum_{k=j'+1}^{j} s_k = -o \end{array} \right\}
\end{aligned}$$

$\mathrm{pp}(i, j, o)$ is the set of program points that can be at byte offset o of program point (i, j). If we had access to the assembled version of assembly blocks, we would precisely know the length of each instruction, that is, for all $i \in [\![1, b]\!]$ and $j \in [\![1, l_i]\!]$, $S_{i,j}$ would be a singleton. In this case, we would be able to know exactly the program point corresponding to a (precise) code pointer. More formally, images of pp are only singletons or the empty set.

We also need the predicate

$$
\begin{aligned}
&\mathrm{invd} : \mathbb{N}^* \times \mathbb{N} \times \mathbb{Z} \rightarrow \{ff, tt\} \\
&(i,j,o) \mapsto \left(\begin{array}{l} o \geqslant 0 \wedge \exists j' \in \mathbb{N} : \exists (s_k)_{k \in [\![j+1, j']\!]} \in \mathbb{N}^{j'-j} : \\ \qquad \begin{cases} \forall k \in [\![j+1, j']\!], s_k \in S_{i,k} \wedge \\ \displaystyle\sum_{k=j+1}^{j'-1} s_k < o < \sum_{k=j+1}^{j'} s_k \end{cases} \end{array} \right) \\
&\qquad \bigvee \left(\begin{array}{l} o \geqslant 0 \wedge \exists (s_k)_{k \in [\![j+1, l_i]\!]} \in \mathbb{N}^{l_i - j} : \\ \qquad \begin{cases} \forall k \in [\![j+1, l_i]\!], s_k \in S_{i,k} \wedge \\ \displaystyle\sum_{k=j+1}^{l_i} s_k < o \end{cases} \end{array} \right) \\
&\qquad \bigvee \left(\begin{array}{l} o \leqslant 0 \wedge \exists j' \in \mathbb{N} : \exists (s_k)_{k \in [\![j'+1, j]\!]} \in \mathbb{N}^{j-j'} : \\ \qquad \begin{cases} \forall k \in [\![j'+1, j]\!], s_k \in S_{i,k} \wedge \\ \displaystyle\sum_{k=j'+2}^{j} s_k < -o < \sum_{k=j'+1}^{j} s_k \end{cases} \end{array} \right) \\
&\qquad \bigvee \left(\begin{array}{l} o \leqslant 0 \wedge \exists (s_k)_{k \in [\![1, j]\!]} \in \mathbb{N}^{j} : \\ \qquad \begin{cases} \forall k \in [\![1, j]\!], s_k \in S_{i,k} \wedge \\ \displaystyle\sum_{k=0}^{j} s_k < -o \end{cases} \end{array} \right)
\end{aligned}
$$

invd(i, j, o) is true if and only if the code pointer can be outside the assembly block, or in the middle of an instruction. The first and third cases correspond to offsets that may be in the middle of an instruction. Second and fourth cases test whether the offset may be out of the block.

We are interested into resolving program points expressed as the sum of another program point and an offset because we cannot build a code pointer *ex nihilo*, as well as pointers in C (to variables) or functions must be based on the "address of" (&) operator. The assembly equivalent is to use a label: they are the only way to get a valid code pointer. Moreover, each label corresponds to a single program point.

Because of this similitude, code pointers are abstracted in the same way as C pointer, as described in [Min06; Min13]. C pointers are made of a base (a variable, an array, a structure) and an offset inside the block it points to. The abstract counterpart of such a pointer is a set of bases and an abstraction of a set of integers. The same holds for code pointers. Bases are reference points: they can be any assembly program point. In practice, we cannot directly address any program point: it has to be labeled, or pushed on the stack by a call instruction.

```
1 mov EBX, 0
2 mov EAX, label
3 add EAX, 3
4 jmp EAX
5 label:
6 add EBX, 1  ; This instruction has 3 bytes: 83 C3 01
7 add EBX, 2
8 ; Here EBX == 2
```

Listing 17.1: Computing a program point as a pointer

We can illustrate such abstraction with Listing 17.1. EBX is used as a counter, and EAX stores a code pointer. After line 2, EAX has a value relative to `label` with offset 0. After the line 3, we added an offset of 3, then EAX is the pointer `label` + 3. In the abstract, it is represented by a set of program points containing `label` and an abstract set of integers such that its concretization includes 3 (and only 3, if the abstraction is precise). Thanks to pp, we can translate `label` + 3 (and other code pointers in the concretization of EAX) to program point so that we can translate this dynamic destination into a set of possible destinations. Thanks to invd, we can check whether `label` + 3 might not designate a program point.

Like C function pointers that must be concretized to get pointed functions, we need to concretize code pointers. Function pointers are quite safe to concretize: there are no more elements in the concretization than the number of function in the code. For code pointer are made from a label (that is easy to concretize), and an offset. These labels are no more than the number of program points in assembly blocks, which is small enough not to be worried about. But, we should take care, when concretizing the offset part, to avoid too large (or infinite) concretizations. But we can bound the interval of interesting offsets from statement (i, j): the minimal offset is

$$-\sum_{k=0}^{j} \max S_{i,k}$$

and the maximal is

$$\sum_{k=j+1}^{l_n} \max S_{i,k}$$

Offsets out of these bounds are out of the assembly block and necessarily invalid. Proper implementation should only consider offsets within these extrema to avoid catastrophic concretization.

In ASTRÉE, since code pointers are very similar to C pointers, they are implemented in the same way, and especially in the same domain: they are both handled by the pointer domain (see Figure 14.1 “New structure of the composite domain of ASTRÉE” (page 179)). The main modification is to add assembly program points to the set of allowed bases and to adapt handling of other kind of pointers: concretizing an abstract pointer as function pointer may be invalid when it contains assembly bases; dereferencing a data pointer may also be invalid if there are assembly bases.

This is just the first step to execute control transfer statements. Once we have the possible destination program points, we still need to actually perform the jump. Depending on the destination, the jump is not handled in the same way. We distinguish several kinds of jumps. Among near jumps (see subsection 2.4.1 “Near Calls” (page 29)) we make two cases, depending on whether the destination is in the same C function as the jump statement. Far jumps (see subsection 2.4.2 “Far Calls” (page 30)) are preceded by other assignments and checks.

Since destinations are imprecise, a single statement can have destinations in the same function and in others functions. This is not a problem: the jump statement for each destination is performed separately.

17.2 Intra-function Near Jumps

The simplest case is when the destination is in the same C function as the jump statement. Once we know the destination program point, this is not very different from a C goto-statement. They are thus handled in the same way. We will here explain how it works for C gotos, as it is slightly simpler, before explaining the differences to take into account.

17.2.1 The Case of Gotos

The method is to store the current property aside when we execute a goto, and to restore it when we reach the corresponding label. We also need an additional accumulator to save the context on return-statements. For the sake of the formalization, we assume that all return-statements of a function return the same variable. This can be ensured using a simple rewriting.

Formally, let $\mathcal{D}_\mathcal{S}$ be the concrete domain of states (typically, sets of pairs made of a stack of environments and a heap). We define

$$\mathcal{D} := \mathcal{D}_\mathcal{S} \times (\mathbb{L} \to \mathcal{D}_\mathcal{S}) \times (\mathbb{L} \to \mathcal{D}_\mathcal{S}) \times \mathcal{D}_\mathcal{S}$$

where $\mathbb{L}$ is the set of labels. This is the domain that we use to resolve gotos. The first component is the current state (also called direct flow). The second component is the map of states to restore for each label. The third component is where we store the current state while encountering a goto. The last component is the union of states where a return happens.

When we run a goto-statement, we save the current state. The current state become $\bot_\mathcal{S}$ as the program point following a goto is unreachable.

$$\begin{aligned}&[\![\texttt{goto l;}]\!] : \mathcal{D} \to \mathcal{D}\\ &\quad (d, m, g, r) \mapsto (\bot_\mathcal{S}, m, g\left[\texttt{l} \mapsto d \cup_\mathcal{S} g(\texttt{l})\right], r)\end{aligned}$$

When we reached the corresponding label, the provided property for this label is restored into the current one.

$$\begin{gathered}\llbracket \texttt{l:} \rrbracket : \mathcal{D} \to \mathcal{D} \\ (d, m, g, r) \mapsto (d \cup_{\mathcal{S}} m(\texttt{l}), m, g, r)\end{gathered}$$

And finally, to run a return-statement we simply save the current flow, while the current flow is killed.

$$\begin{gathered}\llbracket \texttt{return x;} \rrbracket : \mathcal{D} \to \mathcal{D} \\ (d, m, g, r) \mapsto (\bot_{\mathcal{S}}, m, g, r \cup_{\mathcal{S}} d)\end{gathered}$$

Other unary operations, like assignments and guards, are performed only on the current property. Any function $f_{\mathcal{S}} : \mathcal{D}_{\mathcal{S}} \to \mathcal{D}_{\mathcal{S}}$ can be lifted to $\mathcal{D} \to \mathcal{D}$:

$$\begin{gathered}f : \mathcal{D} \to \mathcal{D} \\ (d, m, g, r) \mapsto (f_{\mathcal{S}}(d), m, g, r)\end{gathered}$$

```
1  int f() {
2      int i = 0;
3      goto l;
4      if(0) {
5          l:
6          i++;
7      }
8      return i;
9  }
```

Listing 17.2: Executing code reachable only through a label

This allows reaching labels in otherwise unreachable blocks, like in Listing 17.2. Initially, the state is totally empty:

$$(\varnothing_{\mathcal{S}}, \texttt{l}:\bot_{\mathcal{S}}, \texttt{l}:\bot_{\mathcal{S}}, \bot_{\mathcal{S}})$$

where $\varnothing_{\mathcal{S}}$ is the state with not living variable. This is not $\bot_{\mathcal{S}}$ since it does not mean that the program point is unreachable. After line 2, the state is

$$((\texttt{i} \mapsto 0), \texttt{l}:\bot_{\mathcal{S}}, \texttt{l}:\bot_{\mathcal{S}}, \bot_{\mathcal{S}})$$

At line 3, we save the current property in the appropriate map.

$$(\bot_{\mathcal{S}}, \texttt{l}:\bot_{\mathcal{S}}, \texttt{l}:(\texttt{i} \mapsto 0), \bot_{\mathcal{S}})$$

We can see that the program point after the goto-statement is unreachable, since the direct flow is $\bot_{\mathcal{S}}$. The condition of line 4 requires applying a guard. This means that it is performed only on the current property. Thus, the state after line 4 is unchanged. At line 5, we restore the saved property for the label `l:`. But the map of states to restore (the second component) is empty. Thus, we keep

$$(\bot_{\mathcal{S}}, \texttt{l:}\bot_{\mathcal{S}}, \texttt{l:}(\texttt{i} \mapsto 0), \bot_{\mathcal{S}})$$

Likewise, at line 6, we increment `i` in a bottom state: it stays the same. At line 7, we have to compute the union of the state that did not enter the if and the state at the end of the body of the if, that is:

$$(\bot_{\mathcal{S}}, \texttt{l:}\bot_{\mathcal{S}}, \texttt{l:}(\texttt{i} \mapsto 0), \bot_{\mathcal{S}}) \cup (\bot_{\mathcal{S}}, \texttt{l:}\bot_{\mathcal{S}}, \texttt{l:}(\texttt{i} \mapsto 0), \bot_{\mathcal{S}})$$

Binary operations are performed pointwise, the property after line 7 is:

$$(\bot_{\mathcal{S}}, \texttt{l:}\bot_{\mathcal{S}}, \texttt{l:}(\texttt{i} \mapsto 0), \bot_{\mathcal{S}})$$

Then, after the return

$$(\bot_{\mathcal{S}}, \texttt{l:}\bot_{\mathcal{S}}, \texttt{l:}(\texttt{i} \mapsto 0), \bot_{\mathcal{S}})$$

We can see that the first iteration returns nothing. This is because nothing was restored at the label `l:`. Indeed, the map of states to restore is initially empty. But we can see that the map of saved states is not empty. This means we need to perform a new iteration of the body of the function with newly collected states. But this time the initial property is

$$(\bot_{\mathcal{S}}, \texttt{l:}(\texttt{i} \mapsto 0), \texttt{l:}\bot_{\mathcal{S}}, \bot_{\mathcal{S}})$$

The second iteration looks like the first one until line 5: this time, there is a property to restore. After line 5, we get

$$((\texttt{i} \mapsto 0), \texttt{l:}(\texttt{i} \mapsto 0), \texttt{l:}(\texttt{i} \mapsto 0), \bot_{\mathcal{S}})$$

This time, the incrementation has an effect; after line 6, we have:

$$((\texttt{i} \mapsto 1), \texttt{l:}(\texttt{i} \mapsto 0), \texttt{l:}(\texttt{i} \mapsto 0), \bot_{\mathcal{S}})$$

After line 7, we have to compute:

$$(\bot_{\mathcal{S}}, \texttt{l:}(\texttt{i} \mapsto 0), \texttt{l:}(\texttt{i} \mapsto 0), \bot_{\mathcal{S}}) \cup ((\texttt{i} \mapsto 1), \texttt{l:}(\texttt{i} \mapsto 0), \texttt{l:}(\texttt{i} \mapsto 0), \bot_{\mathcal{S}})$$

which is

$$((\texttt{i} \mapsto 1), \texttt{l:}(\texttt{i} \mapsto 0), \texttt{l:}(\texttt{i} \mapsto 0), \bot_{\mathcal{S}})$$

This time, there is a direct flow when we reach the return statement, thus, at the end of the function, we get

$$(\bot_{\mathcal{S}}, \texttt{l:}(\texttt{i} \mapsto 0), \texttt{l:}(\texttt{i} \mapsto 0), (\texttt{i} \mapsto 1))$$

Since we are always returning the variable `i`, we know which value to extract in the returned property. This second iteration is stable since we can observe that the initially

```
1 int f() {
2     int i = 1;
3     goto l;
4     m:
5     return i;
6     l:
7     i++;
8     goto m;
9 }
```

Listing 17.3: Backward goto

provided map of states to restore is equal to the map of saved states: we have $(\mathtt{i} \mapsto 0)$ in both maps.

This method work perfectly well for backward gotos, too. Let us illustrate on Listing 17.3. After line 3, the state is

$$\left(\bot_{\mathcal{S}}, \begin{pmatrix} \mathtt{l}{:}\bot_{\mathcal{S}} \\ \mathtt{m}{:}\bot_{\mathcal{S}} \end{pmatrix}, \begin{pmatrix} \mathtt{l}{:}(\mathtt{i} \mapsto 1) \\ \mathtt{m}{:}\bot_{\mathcal{S}} \end{pmatrix}, \bot_{\mathcal{S}}\right)$$

Line 4 restores the property for label `m`, i.e. nothing, so the current property stays $\bot_{\mathcal{S}}$. Line 5 just adds $\bot_{\mathcal{S}}$ to returned state, once again, nothing change. At line 6, we restore the property saved for label `l` (so, still $\bot_{\mathcal{S}}$), and we save the current property for label `m`. At the end of the function, the state is still

$$\left(\bot_{\mathcal{S}}, \begin{pmatrix} \mathtt{l}{:}\bot_{\mathcal{S}} \\ \mathtt{m}{:}\bot_{\mathcal{S}} \end{pmatrix}, \begin{pmatrix} \mathtt{l}{:}(\mathtt{i} \mapsto 1) \\ \mathtt{m}{:}\bot_{\mathcal{S}} \end{pmatrix}, \bot_{\mathcal{S}}\right)$$

Since the map of saved jumps is not equal to the provided map of states to restore, we need to iterate further. In particular, we can see there is a jump to label `l` that was not known before.

For the second iteration, we start with this state

$$\left(\varnothing_{\mathcal{S}}, \begin{pmatrix} \mathtt{l}{:}(\mathtt{i} \mapsto 1) \\ \mathtt{m}{:}\bot_{\mathcal{S}} \end{pmatrix}, \begin{pmatrix} \mathtt{l}{:}\bot_{\mathcal{S}} \\ \mathtt{m}{:}\bot_{\mathcal{S}} \end{pmatrix}, \bot_{\mathcal{S}}\right)$$

At line 6, we restore the state for label `l`. We get the state

$$\left((\mathtt{i} \mapsto 1), \begin{pmatrix} \mathtt{l}{:}(\mathtt{i} \mapsto 1) \\ \mathtt{m}{:}\bot_{\mathcal{S}} \end{pmatrix}, \begin{pmatrix} \mathtt{l}{:}(\mathtt{i} \mapsto 1) \\ \mathtt{m}{:}\bot_{\mathcal{S}} \end{pmatrix}, \bot_{\mathcal{S}}\right)$$

Then, `i` is incremented, and, at the end of the function, we get

$$\left(\bot_{\mathcal{S}}, \begin{pmatrix} \mathtt{l}{:}(\mathtt{i} \mapsto 1) \\ \mathtt{m}{:}\bot_{\mathcal{S}} \end{pmatrix}, \begin{pmatrix} \mathtt{l}{:}(\mathtt{i} \mapsto 1) \\ \mathtt{m}{:}(\mathtt{i} \mapsto 2) \end{pmatrix}, \bot_{\mathcal{S}}\right)$$

Once again, there new possible jumps. In particular, there a jump to label `m` that was unknown before. We iterate the analysis again with the initial state

$$\left(\varnothing_{\mathcal{S}}, \begin{pmatrix}\mathtt{l}\colon(\mathtt{i}\mapsto 1)\\ \mathtt{m}\colon(\mathtt{i}\mapsto 2)\end{pmatrix}, \begin{pmatrix}\mathtt{l}\colon\bot_{\mathcal{S}}\\ \mathtt{m}\colon\bot_{\mathcal{S}}\end{pmatrix}, \bot_{\mathcal{S}}\right)$$

At this iteration, we have something to restore at label `m`. Thus, after line 4, we have the state

$$\left((\mathtt{i}\mapsto 2), \begin{pmatrix}\mathtt{l}\colon(\mathtt{i}\mapsto 1)\\ \mathtt{m}\colon(\mathtt{i}\mapsto 2)\end{pmatrix}, \begin{pmatrix}\mathtt{l}\colon(\mathtt{i}\mapsto 1)\\ \mathtt{m}\colon\bot_{\mathcal{S}}\end{pmatrix}, \bot_{\mathcal{S}}\right)$$

This time, the direct flow is not empty at line 5: we have something to return.

$$\left(\bot_{\mathcal{S}}, \begin{pmatrix}\mathtt{l}\colon(\mathtt{i}\mapsto 1)\\ \mathtt{m}\colon(\mathtt{i}\mapsto 2)\end{pmatrix}, \begin{pmatrix}\mathtt{l}\colon(\mathtt{i}\mapsto 1)\\ \mathtt{m}\colon\bot_{\mathcal{S}}\end{pmatrix}, (\mathtt{i}\mapsto 2)\right)$$

We finish this iteration, the state for label `l` is restored, `i` is incremented, and the state is saved for label `m`. We obtain:

$$\left(\bot_{\mathcal{S}}, \begin{pmatrix}\mathtt{l}\colon(\mathtt{i}\mapsto 1)\\ \mathtt{m}\colon(\mathtt{i}\mapsto 2)\end{pmatrix}, \begin{pmatrix}\mathtt{l}\colon(\mathtt{i}\mapsto 1)\\ \mathtt{m}\colon(\mathtt{i}\mapsto 2)\end{pmatrix}, (\mathtt{i}\mapsto 2)\right)$$

This time, we have reached a fixpoint since the map of saved states is the same as the provided map of jumps. We can also see that the end of the function is definitely unreachable and that the function returns the integer 2.

Formally, the semantics of one iteration of the body of a function is a map

$$f : \mathcal{D} \to \mathcal{D}$$

Let $d \in \mathcal{D}_{\mathcal{S}}$ the property at the top of the function. This is $\varnothing_{\mathcal{S}}$ in the examples, but it can be non-empty if the function has parameters or if there are global variables. We are interested in the least fixpoint of

$$\begin{aligned} f' : \mathcal{D}_{\mathcal{S}}^{\mathbb{L}} \times \mathcal{D}_{\mathcal{S}} &\to \mathcal{D}_{\mathcal{S}}^{\mathbb{L}} \times \mathcal{D}_{\mathcal{S}} \\ (g, r) &\mapsto \left(\pi_3\left(f(d, g, g, r)\right), \pi_4\left(f(d, g, g, r)\right)\right) \end{aligned}$$

The least fixpoint of f' allow to get the actual returned value, where all gotos have been stabilized.

We can see that this semantics looks different to what we formalized in Part I "Semantics of Mixed C and x86 Assembly" (page 41). In PANACEA, when we encounter a jump, we simply update the current program point. This is a very intuitive interpretation of gotos, and it works well in an operational semantics. To formalize an abstraction, it is easier to start from a denotational semantics, which is closer to the formalism of this section.

The equivalence between both formalisms for gotos is quite direct: rather than updating the program point to the destination (that is forwarding the current state directly elsewhere), we save the current property, and we enumerate program points cyclically until we reach the destination, where we restore the property.

It is easy to get an abstraction from this formalism: we need an abstract domain $\mathcal{D}^{\sharp}_{\mathcal{S}}$, that abstracts $\mathcal{D}_{\mathcal{S}}$, and we change the transfer functions and union to their abstract counterparts, and we replace the least fixpoint by a correct approximation of it.

For implementation purpose, we may propose a few improvements. The major one is to use a single map to save and restore states, without waiting for the next iteration. With this optimization, the domain has type

$$\mathcal{D} := \mathcal{D}_{\mathcal{S}} \times (\mathbb{L} \to \mathcal{D}_{\mathcal{S}}) \times \mathcal{D}_{\mathcal{S}}$$

In addition to the memory gain, this optimization allows forward jumps to be resolved in a single iteration. In other words, we perform computations as soon as possible, rather than systematically waiting for the next iteration. Soundness of this improved method can be proved by seeing it as an instance of chaotic iterations (see [Cou78]). The point of chaotic iterations is to arbitrarily reorder computations in the resolution of a system of equations, but to do them anyway. Eventually, we reach the same solution.

17.2.2 Handling Assembly Jumps

We use a very similar technique to handle assembly local jumps, but with few differences.

On the jump side, the destination may be imprecise. The current property must be saved for each of possible destination. Using pointers (explicitly or not) in the data structure that saves properties allows sharing the same structure for each possible destination, allowing efficient storage.

In assembly, destinations are not only labels: they may be arbitrary program point. We must check at each computation step whether there is a program point for the current program point, and restore it if applicable. Assembly labels are only a special case. Otherwise, everything is the same, especially stabilization of non-forward jumps[†], except that we need to stabilize both maps of properties saved by C gotos and by assembly jumps.

From an implementation point of view, this is implemented in the environment domain (see Figure 14.1 "New structure of the composite domain of ASTRÉE" (page 179)). The underlying domain (i.e. trace domain and below) forms the domain $\mathcal{D}^{\sharp}_{\mathcal{S}}$ that abstract a single state. For practical reasons, the maps of saved states for C gotos and assembly jumps are disjoint. This is a precaution not to accidentally change the behavior of the analysis of pure C programs. Moreover, it is easier to make OCAML's typer happy this way, as the domain (the type of keys) of both maps are different.

17.3 Inter-function Near Jumps

We have treated the case of jumps whose destination is in the same function as the jump statement. But they can also be in another function. There are implementation

[†]Either backward or horizontal jumps.

constraints: Astrée does not support recursive functions, thus we cannot treat all jumps like a C call. Moreover, flavors of jumps (`JMP`, `CALL` and `RET`) are not reliable hints on the control flow. A simple jump can be done with a `RET`, as well as what are morally a call to and a return from a procedure can be done using `JMP` only. In the following, we will only talk about undiscerned jumps, since they can all be reduced to `JMP` with some stack operations; the control flow part stays the same. Another difficulty is that, sometimes, there is not such strict interpretation of calls and returns: an assembly function `f` can jump to a function `g`, then `g` jumps to `h` and `h` returns to `f`. The first jump can be understood as a call whose return would be the last jump, but the jump from `g` to `h` has no clear meaning. Such cases are not only feasible, they appear in real-world code. Indeed, more regular control flows are done in C, as much as possible. Overall, it is useless to try to impose a C conception of control flow to assembly code. This problem has been explored in chapter 7 "Pathological and Didactic Examples" (page 91). We also remember that we should preserve pure C analyses.

The method used for gotos and local jumps allows mixing well-structured C with poorly-structured jumps. Unfortunately, this is limited to a single function. The idea is to adapt this method to inter-function jumps. The first step is to recall how we detect recursive calls.

As Astrée uses a single variable for each local variable, we should not have two instances of the same syntactical local variable. That's why Astrée does not support recursive functions. This is not a problem since they are not allowed in embedded critical software as they easily lead to a memory usage difficult to predict: recursive calls are like dynamic allocations on the stack. Astrée cannot only assume that analyzed programs do not have recursive calls: it has to warn when they happen, after which Astrée can kill such traces[‡]. To detect recursive calls, we need to have some kind of stack abstraction that keeps, at least, functions in the current call stack. Partitioning domain (see Figure 14.1 "New structure of the composite domain of Astrée" (page 179)) is in charge to build a trace abstraction from a state abstraction. It abstracts traces in a very light way: they are a list of events on the trace (like active calls[§], whether conditions of if-statement are successful, the number of unrolled iterations in loops...) with the last state of the trace. Older states are totally forgotten. When a call is performed, the iterator asks the composite underlying domain to filter traces that already have an active call to the same function in their history. If there are some, an alarm is raised, and these traces are ignored: the analysis continue with traces without recursive calls.

This is the criterion according to which we classify jumps: whether the destination is in the call stack or not. If it is not, we should add it; these are "forward distant jumps"[¶]. If the destination function is already in the call stack, adding it would be like a recursive function, thus we need to wait to return until we reach this destination function again

[‡]More generally, we can reject any unwanted behavior, as long as we detect when they happen, and warn the user about it. The only constraint is that the analysis should still be useful for our applications.

[§]That is, calls have not yet returned.

[¶]This terminology is not standard. It was chosen specifically these notions that have little meaning outside the context of this work.

to handle it; these are “return distant jumps”. We can easily classify whether a distant jump is forward or return using the partitioning domain; this classification must be done dynamically[**].

A forward distant jump is not that different from a regular C call, but it does not always include information needed for return (if it is performed by a `JMP` or `RET`) and the state at call-site is not injected at the beginning of the function (it is $\perp$ here), but on the destination of the call. This is very similar to a local jump: the current state becomes $\perp$, but we saved it to be able to restore it when appropriate. But, in the case of forward distant jumps, the saved state does not come from the previous iterations of the function, but on the analysis of another function. Before being able to give an example, we shall see how to handle return distant jumps.

Return distant jumps are dual of forward distant jumps, and they differ in the same way from regular C returns, as forward distant jumps differ from calls: return distant jumps may not use return information, and the property at jump-site is not returned at a hypothetical call-site (which may not exist). The current property is saved to be restored at the suitable destination, but this backup has no use in the current function: it is given to the previous function on the stack, until it reaches the function that contains the destination.

We shall emphasize that C-style returns are not compatible with assembly jumps: a C return statement would execute a postlude whose corresponding prelude have not been executed. We have to check that a function that is accessed with an assembly jump does not end with a C return. Likewise, the end of such functions should not be reachable as it would imply the execution of an implicit return, including the execution of the postlude. The control can only escape by a distant jump from a function accessed by a distant jump.

Let us illustrate with example of Listing 17.4. We assume that the execution starts at `f` with a call stack that does not contain any of the functions of this snippet. We start by analyzing `f`. After line 18, the state is $\mathrm{EAX} \mapsto 0$, other variables are not relevant for our example. Line 19 jumps to label `g` in function `g_wrapper`. Current state is saved and replaced by $\perp$. This jump is detected to be a forward distant jump and thus we create the map g: $(\mathrm{EAX} \mapsto 0)$ of forward distant jumps. Then, the analysis of the current function proceeds with only $\perp$ until the end. There is no map of local jumps, thus, there is no need to iterate the analysis of this function for now. But, there is a map for forward jump, thus we need to analyze new functions. Since the destination is in `g_wrapper`, we analyze this function. The current state at the beginning is $\perp$, but it also gets the map g: $(\mathrm{EAX} \mapsto 0)$, which is a map for local jumps in this context. At this point, the call stack contains both `f` and `g_wrapper`.

During the analysis of `g_wrapper`, the current property stays $\perp$ until we reach label `g`. When we do, the saved property is restored. After line 11, it is $\mathrm{EAX} \mapsto 1$, and at line 12, we execute a new forward distant jump, thus we use the same strategy as for

[**]Indeed, we need such a lenient handling of control flow because it is hopeless to understand it statically. If we could know in advance which jumps are forward or return, we could infer interesting structures.

```
1  void h_wrapper() {
2      asm {
3          h:
4          add EAX, 2
5          jmp r
6      }
7  }
8  void g_wrapper() {
9      asm {
10         g:
11         add EAX, 1
12         jmp h
13     }
14 }
15 void f() {
16     int x;
17     asm {
18         mov EAX, 0
19         jmp g
20         r:
21         mov x[EBP], EAX
22     }
23 }  // Here x == 3
```

Listing 17.4: Forward and return distant jumps

`f`. At the end of the function, the current property is $\perp$, and we have saved the map `h:`(EAX $\mapsto$ 1).

Analysis of `h_wrapper` starts as the analysis of `g_wrapper`: $\perp$ as current property and saved map `h:`(EAX $\mapsto$ 1). The stack contains `f`, `g_wrapper` and `h_wrapper`. After line 3, the property is restored and the current state becomes EAX $\mapsto$ 1, and EAX $\mapsto$ 3 after line 4. Here, the destination of the jump is found in `f`, which is already in the stack. This jump is thus a return distant jump. The current property becomes and stays $\perp$ until the end of the function. As there are no local jump maps, we reached the fixpoint, and since there are no forward distant jump maps, there is no new function to analyze. But the map `r:`(EAX $\mapsto$ 3) is returned to the "calling" function (according to the stack): `g_wrapper`. Here, we detect that there is no new property as the return distant jump goes even further. No new analysis is required, and the map is forwarded even deeper.

Function `f` gets in input both the initial environment and the return jump `r:`(EAX $\mapsto$ 3) that is local in `f`'s context. Its analysis gives the same result until line 20. Before, current property is $\perp$, like it was at the first iteration, but now, there is a state to restore for label `r:`. Current property after line 19 becomes EAX $\mapsto 3$, and we reach the end of the function this way. As there is no local jump map, this function is stabilized, and since the forward jump map is the same, all called functions are also stabilized.

```
1  // Current property: ⊥
2  // Local jump map: h ↦ (EAX ↦ 1)
3  void h_wrapper() {
4      asm {
5          h:
6          add EAX, 2
7          jmp r
8      }
9  }
10 // Current property: ⊥
11 // Return distant jump map: r ↦ (EAX ↦ 3)
12
13 // Current property: ⊥
14 // Local jump map: g ↦ (EAX ↦ 0)
15 void g_wrapper() {
16     asm {
17         g:
18         add EAX, 1
19         jmp h
20     }
21 }
22 // Current property: ⊥
23 // Forward distant jump map: h ↦ (EAX ↦ 1)
24
25 void f() {
26     int x;
27     asm {
28         mov EAX, 0
29         jmp g
30         r:
31         mov x[EBP], EAX
32     }
33 }
34 // Current property: ⊥
35 // Forward distant jump map: g ↦ (EAX ↦ 0)
```

Listing 17.5: Propagation of forward jumps iterations

This analysis is illustrated on Listing 17.5 and Listing 17.6, where input and output

```
// Current property: ⊥
// Local jump map: h ↦ (EAX ↦ 1)
void h_wrapper() {
    asm {
        h:
        add EAX, 2
        jmp r
    }
}
// Current property: ⊥
// Return distant jump map: r ↦ (EAX ↦ 3)

// Current property: ⊥
// Return distant jump map: r ↦ (EAX ↦ 3)
// Local jump map: g ↦ (EAX ↦ 0)
void g_wrapper() {
    asm {
        g:
        add EAX, 1
        jmp h
    }
}
// Current property: ⊥
// Return distant jump map: r ↦ (EAX ↦ 3)
// Forward distant jump map: h ↦ (EAX ↦ 1)

// Local distant jump map: r ↦ (EAX ↦ 3)
void f() {
    int x;
    asm {
        mov EAX, 0
        jmp g
        r:
        mov x[EBP], EAX
    }
}
// Current property: x ↦ 3
// Forward distant jump map: g ↦ (EAX ↦ 0)
```

Listing 17.6: Propagation of return jumps iterations

states and maps are annotated in comments. Listing 17.5 shows the resolution of forward jumps, until we get a return distant jump. Listing 17.6 shows the remaining of this analysis, propagating backward the return distant jump.

We can easily see that this method works perfectly with C analysis as it does not interfere.

To formalize this method, we need a stack to distinguish forward and return distant jumps. We analyze a function, and we iterate it until we reach a fixpoint for local jumps. Then, we analyze function pointed by forward distant jumps (and recursively), and we collect return jumps. The function is then reanalyzed with these new jumps, until fixpoint of forward jumps. Each iteration to reach the fixpoint of forward jumps imply to find the fixpoint of local jumps and to analyze functions containing the destinations.

We need to adapt slightly the type $\mathcal{D}$ to allow maps for distant jumps:

$$\mathcal{D} := \mathcal{D}_{\mathcal{S}} \times (\mathbb{L} \to \mathcal{D}_{\mathcal{S}}) \times (\mathbb{L} \to \mathcal{D}_{\mathcal{S}}) \times (\mathbb{L} \to \mathcal{D}_{\mathcal{S}}) \times \mathcal{D}_{\mathcal{S}}$$

Such objects contains respectively the current state, the map of local jumps, the map of forward distant jumps, the map of return distant jump and returned states.

We need to adapt slightly the semantics of jumps so that it saves the current state to the right map. For this step, we need to be able to translate a code pointer to program points (this issue has already been discussed), to know in which function stands each program point, and to know whether each of these functions is in the call stack.

After stabilization of local jumps, the semantics of a function is a function of type

$$f : \mathcal{D}_{\mathcal{S}}^{\mathbb{L}} \to \mathcal{D}_{\mathcal{S}}^{\mathbb{L}} \times \mathcal{D}_{\mathcal{S}}^{\mathbb{L}} \times \mathcal{D}_{\mathcal{S}}$$

where $\mathbb{L}$ is the set of assembly program points. The parameter is the map of new local jumps: distant jumps that are local in the context of the function. Returned values are respectively the map of forward distant jumps, the map of return distant jumps and the returned property.

From f, we build $\overleftarrow{f}$ the function of type

$$\overleftarrow{f} : \mathcal{D}_{\mathcal{S}}^{\mathbb{L}} \to \mathcal{D}_{\mathcal{S}}^{\mathbb{L}} \times \mathcal{D}_{\mathcal{S}}$$

that stabilized forward distant jumps and returns the return distant jump and C-returned property. For that, we get all bindings $l \mapsto s$ (with s non-empty) in the forward distant jump map returned by f, and for each such binding, we use the function $\overleftarrow{g}$ where g is the function containing program point l, with the input map $l \mapsto s$. This recursive process terminates since the depth cannot be greater than the number of functions in the program. Indeed, when there is no forward distant jumps, we have

$$\overleftarrow{f}(m) = (b, c) \text{ where } (\bot, b, c) = f(m)$$

For functions accessed by distant jumps, such as g, we have to check that $c = \bot$, and raise an alarm otherwise.

In $\overleftarrow{f}$, there are still some returned flows. There is no relevant notion such as the semantics of a function with such arbitrary jumps. We are instead interested in the semantics of the program. For that, we consider the function $\overleftarrow{\texttt{main}}$ where `main` is the entry point of the program. As there is no caller for `main`, it cannot produce return distant jumps. Then, $\overleftarrow{\texttt{main}}$ stabilized all jumps and, consequently, it is the semantics of the program.

In this method, we analyzed independently each jump $l \mapsto s$ and $m \mapsto t$, even if program points l and m are located in the same C function, which allows a better precision. We could also analyze it once, by injecting the map $l \mapsto s, m \mapsto t$. This could allow better performances. In practice, the precise strategy was good enough in terms of execution time.

17.4 Mixed Calls

Mixed calls from assembly to C do not require more care than respecting the calling convention. Indeed, from the point of view of the control flow, there are just like regular C calls: the callee is a standard C function and should end with a normal C statement. Of course, during the execution of the callee, the control flow can be complex, but this is outside the scope of the mixed call.

But calls from C to assembly are more complicated. The assembly callee should return properly so that the C caller can continue correctly its execution. Calling convention cdecl specifies that the callee gets a stack containing (from top to bottom[††]) the return address and parameters, as shown on Figure 17.1. Return address R is a code pointer in compiled C code, just after the generated `CALL` instruction. According to the calling convention, return is expected to jump to address R with a stack that still contains parameters.

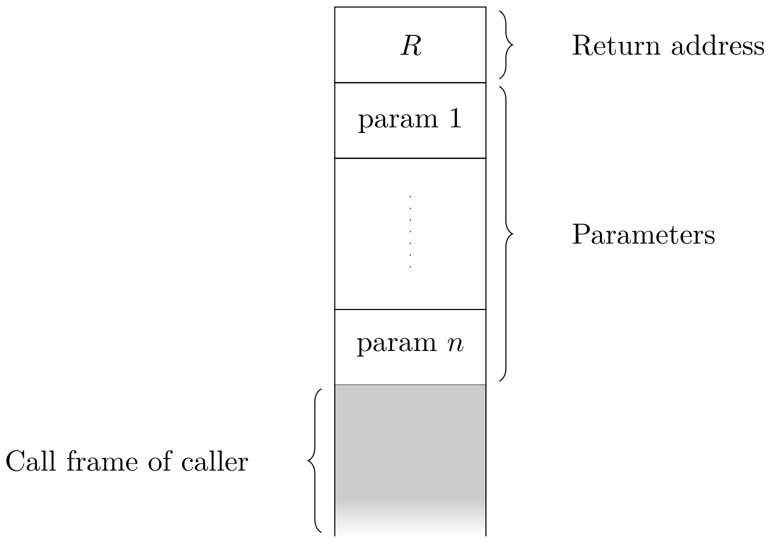


Figure 17.1: State of the stack just after a call from C to assembly

[††]That is, in increasing addresses. We recall that x86 stacks grow down in memory.

This case is not different from the other ones: `RET` instruction is not special here either. One could perform the same jump by popping the address in a register before using a simple `JMP`. But the return address is particular. We cannot tell its value, thus, we handle it purely symbolically. When jumping to R, the analyzer checks that the stack still contains n variables, and performs the jump just like a C return: EAX holds the returned value and the execution of the caller can resume. We cannot merge C returns with "jump to R" returns. Indeed, in this context, C returns are forbidden since the caller called directly an assembly label: we need to distinguish these two kinds of returns.

To handle symbolically R in ASTRÉE, we push a special value on the stack that can be moved, but is never the result of any operation, which guarantee that if the top of the stack is this special value, it has not been modified. It fails to analyze a program that pops the return address, applies some computations that does not modify the final value (like an increment and a decrement), and pushes it back before returning. There is no such code in our target software and it was judged to be dubious, for the least. If we want to be able to analyze such code, we need to make R and ordinary unknown value, and keep a copy. Return is successful if we manage to prove that the current value is still the original value. Power of this method depend directly on the precision of underlying domains. The technique explained in Part IV "A Reduced Product with Ghost Variables" (page 229) might help to prove that the return address is correctly restored.

Return distant jumps are still possible and does not end the call.

17.5 Far Jumps and Task Switches

For now, we have explained how to make any near jump work with C code. This kind of jumps only updates EIP, the instruction pointer, and thus cannot switch between tasks, use a new segment register or, a fortiori, perform inter-privilege calls.

These calls are very important during boot to set up segments, tasks.... They also happen when using a call gate or handling an exception (for a syscall, typically).

Fortunately, these jumps are not very different from near jumps. Pseudo-code given in [Int20] shows that far jumps can be decomposed into multiple checks and assignments, and an update of EIP. This local view drops completely the intuitive notion of tasks or temporary privilege elevation. These are not intrinsic to instructions, but are emergent properties resulting of a good use of these features, just like the notion of procedure comes from a nice use of calls and returns. But as near jumps are arbitrary in assembly, and not well parenthesized, far jumps are sometimes used in surprising ways. It would be vain to try to give a high-level interpretation of far jumps, thus, we consider it as a sequence of assignments, checks and a simple jump.

In ASTRÉE, far jumps have been quite easy to implement thanks to microcode (see subsection 16.1.3 "Microcode" (page 197)), by translating directly microcode in INTEL's manual.

Part IV

A Reduced Product with Ghost Variables

Chapter 18

Introduction

Some properties are easier to prove using quantities that do not appear directly in the semantics of the program. A semantics where such quantities are made explicit is called non-standard. It does not have more information, it is just presented differently, but it makes programs easier to analyze.

Using such implicit value is a common method in physics, since they allow seeing the problem with a different point of view, which may make it easier to solve. This is especially interesting to exhibit some invariant. For instance, in thermodynamics, Laplace's law states that, for an ideal gas, during an isentropic process, PV^γ is constant (where P is the pressure, V is the volume and γ the adiabatic index). Even if this value does not naturally appear as a parameter of the system, it may help to find actually interesting parameters, in this case, P and V. There are many other conservation laws; among very fundamental such laws, we can mention the conservation of energy-mass, the conservation of electric charge or the conservation of color charge [GSS07]. They all introduce a non-elementary value (resp. the total quantity of energy, the sum of electric charges and the total color charge) that is constant and may help to find the value of elementary parameters.

In abstract interpretation, the mission of an abstract domain is to prove some kinds of properties, and it may need to make explicit some hidden quantities, like physicists do. These are called ghost variables. There are as many kinds of ghost variables than there are kinds of properties. Unlike the examples we gave previously, ghost variables are not necessarily constant: they may change or be refined during the analysis. For modularity reasons we need a way to make several domains to cooperate with a decentralized handling of ghost variables. This is the goal of this part.

In the following chapters, we will see how to make a product of abstract domains with ghost variables, examples of such domains and implementation details. This part is a detailed version of [CF20]. In this chapter, we will see how ghost variables may be useful and why existing solution lacks generality.

18.1 Motivation

Firstly, let us illustrate a few examples where ghost variables are useful. They will help to point out difficulties and constraints that one should expect from a reduced product with shared ghost variables.

18.1.1 Pointers

This first example is not particularly linked to the support of mixed C and assembly: it is the abstraction of pointers. As said before, ASTRÉE uses the abstraction described in [Min06; Min13]. This method describes concrete pointers as an unevaluated sum of a base (a function, the beginning of an aggregate-typed variable (array or structure) or a scalar-typed variable) and an offset relative to this base. Abstract pointers are represented by a set of possible bases and an abstract integer. None of these quantities appear as it in the standard semantics, but we can make them explicit in an enriched semantics. The set of bases is represented directly, since it is not bigger than the set of all declared variables and functions, thus reasonably small. But the abstract integer is not as nice, we depute its representation to an abstract domain that is able to handle integer variables. In ASTRÉE, this domain is the product of relational numerical domains.

The abstract domain that handles pointers is a parametric domain that accepts an underlying numerical domain (see Figure 14.1 "New structure of the composite domain of ASTRÉE" (page 179)). All the commands that come to the numerical domain are generated by the pointer domain. When it needs to represent a pointer's offset, it commands the numerical domain to add a new variable to represent it. Then, all operations on pointers are translated into operations on offsets. For instance, in Listing 18.1, `p` initially points to `x[0]`, its base is then `x` and the offset, denoted $\mathcal{O}\text{ffset}\,(\texttt{p})$ is 0. At line 3, `p` is incremented, according to pointer arithmetic, which is translated into $\mathcal{O}\text{ffset}\,(\texttt{p}) \leftarrow \mathcal{O}\text{ffset}\,(\texttt{p}) + 1 \times$ `sizeof(int)`, which a purely numerical assignment.

```
1  int x[4];
2  int* p = &x[0];
3  p++;
```

Listing 18.1: Simple operations on pointers

From the numerical domain's point of view, the offset is a regular variable, it has no idea that it is a ghost variable. This is very easy since the pointer domain is stacked over the numerical domain, it can rewrite all commands coming from above (struct domain) so that the numerical domain sees only the world built by the pointer domain. This stack structure is very typical, as it can be also found in VERASCO [Jou+15] and is the cornerstone of parametric domains such as power set completion or partitioning. Yet it implies a hierarchical ordering of domains: each domain control the underlying (composite) domain. This strategy does not allow a fully decentralized sharing of ghost variables.

This abstraction already exists in ASTRÉE, but we want the new framework to be able to include an adaptation of this domain. By doing so, we want to get rid of the limitations coming from this hierarchy: for instance we would like the integer domain to be usable by any other domain.

18.1.2 GDT Entries

In subsubsection 2.3.2.2 “Protected Mode Segmentation” (page 19), we have explained segmentation in x86 architecture. It is a memory protection feature that allows cutting the memory into contiguous pieces called segments. Each segment is described by a base address, a length and some access flags. These pieces of information are stored in a memory structure called segment descriptor (see subsubsubsection 2.3.2.2.1 “Segment Descriptor” (page 19)), whose fields are arranged as shown in Figure 2.7 “Layout of a segment descriptor” (page 20).

Call gates are protected pointers to a more privileged procedure (see subsubsection 2.4.2.3 “Inter-Privilege Calls” (page 31)). They form, with similar features (trap gates, interrupt gates and task gates), a very useful way to allow controlled privilege transfer. These gates are described by structures very similar to segment descriptors. For instance, call gate structure is shown on Figure 2.13 “Structure of a 32-bit call gate” (page 31).

We can see that fields in segment and trap descriptors are not contiguous. For instance, in call gate descriptors, the pointer to the procedure is cut in 3 parts. They are usually built with bitwise computations, as shown on Listing 2.1 “Building a segment descriptor” (page 20). When these descriptors are used by the processor, it reconstructs the fields transparently. To analyze successfully such code, we need to be able to handle precisely bitwise operations on pointers. Especially, we want to prove that a pointer that is split into multiple parts and then rebuilt gets its original value back.

```
1 int f(int x) {
2     int low = x & 0xffff;
3     int high = x >> 16;
4     int rebuilt = low | (high << 16)
5     return rebuilt
6 }
```

Listing 18.2: A complicated way to do nothing

For instance, in Listing 18.2, function `f` returns the same value it was given in argument, no matter if the value was an integer or a pointer. Lines 2 and 3 cut the argument using bitwise expressions, this is a simplified version of the construction of a segment/gate descriptor. Line 4 rebuilds the original value, this is what the processor does silently. In the case of a gate descriptor, `int rebuilt` is the reconstructed pointer to the desired procedure.

The problem posed by this kind of code can be easily solved using ghost variables. We need ghost variables to remember from which expression each bit-slice comes from. After line 2, `int low` is made of two parts: 16 lower bits are the 16 lower bits of `int x`, while 16 upper bits are zeroes. Zeroes can be stores as it, but we need a ghost variable to store the lower part. Indeed, we cannot just keep the relation "16 lower bits of `x` are the 16 lower bits of `low`" since it would not hold if `x` is modified and cannot be established if `x` is replaced by an rvalue, like `x + 1`*. Instead, we use a ghost variable $\texttt{low}'$ which is assigned the value of `x`, and we remember the relation that "16 lower bits of $\texttt{low}'$ are the 16 lower bits of `low`". By doing so, we rely on another domain to remember than $\texttt{low}' = \texttt{x}$, at least until `x` is modified, and then to keep a relation between them. The relation can be drawn as on Figure 18.1, using the notation of bit slice extraction as explained in subsection 2.1.3 "Conventions" (page 9) and section A.1 "Architecture" (page 369).

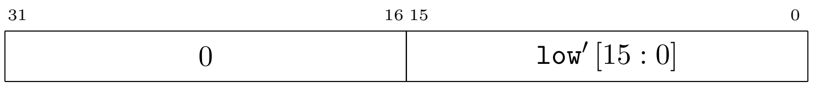


Figure 18.1: A variable defined by bit slices

For the sake of clarity, we need a most systematic nomenclature of these ghost variables. We name $\mathscr{S}\text{lice}^{[a:b]}_{[c:d]}(v)$ the ghost variable whose bits a to b are used to represent bits c to d of variable v, with the constraint $b - a = d - c$. In this nomenclature $\texttt{low}'$ is called $\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low})$.

Of course, there are several legal values for $\mathscr{S}\text{lice}^{[a:b]}_{[c:d]}(v)$: bits outside the segment $[\![a, b]\!]$ are not constrained. But the value of v at the moment where we created $\mathscr{S}\text{lice}^{[a:b]}_{[c:d]}(v)$ is a good candidate as it allows easier deduction later. Once again, using another value is still correct, but subsequent computations and refinements may fail to give something interesting.

After line 3 of Listing 18.2, the 16 higher bits of `high` are zeroes and the 16 lower bits are the 16 higher bits of `x`. We introduce a variable $\mathscr{S}\text{lice}^{[16:31]}_{[0:15]}(\texttt{high})$ whose bits 16 to 31 are bits 0 to 15 of `high`. This is illustrated by Figure 18.2. Ghost variable $\mathscr{S}\text{lice}^{[16:31]}_{[0:15]}(\texttt{high})$ gets the value of `x`.

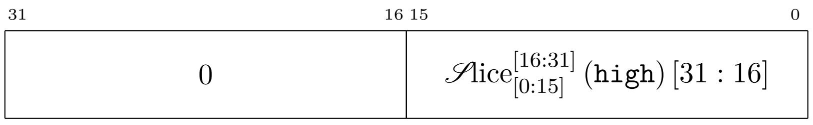

31
16 15
0
0
$\mathscr{S}\text{lice}^{[16:31]}_{[0:15]}(\texttt{high})[31:16]$

Figure 18.2: A variable defined by bit slices with a more systematic terminology

At line 4, the right-hand side expression is a 32-bit value whose 16 higher bits are the 16 higher bits of $\mathscr{S}\text{lice}^{[16:31]}_{[0:15]}(\texttt{high})$ and 16 lower bits are the 16 lower bits of $\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low})$, as shown on Figure 18.3. Thanks to another domain, like a simple

*Or by anything that has a short lifetime.

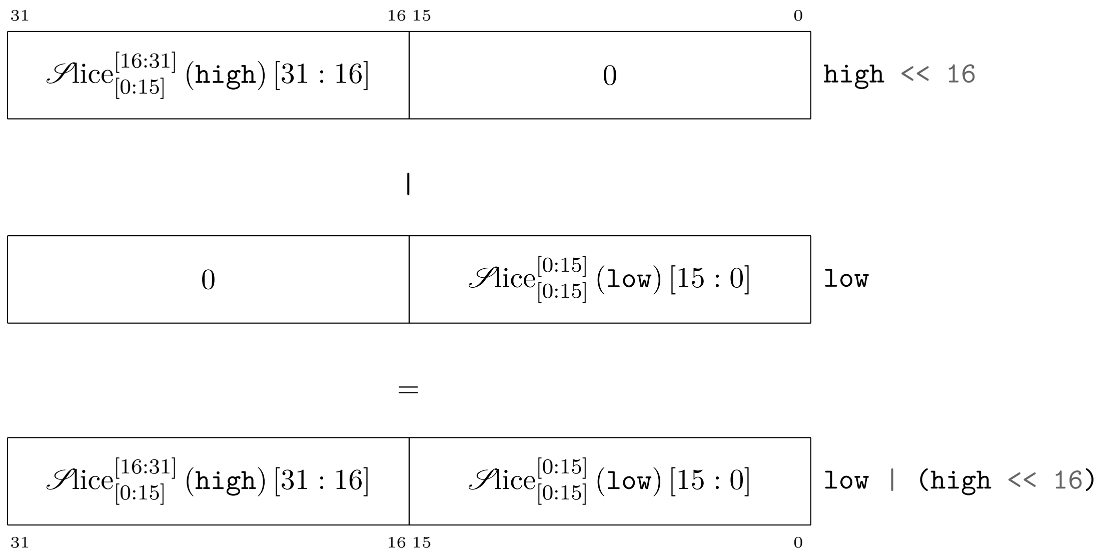

31
16 15
0
$\mathscr{S}\text{lice}^{[16:31]}_{[0:15]}(\texttt{high})\,[31:16]$
0
high << 16
|
0
$\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low})\,[15:0]$
low
=
$\mathscr{S}\text{lice}^{[16:31]}_{[0:15]}(\texttt{high})\,[31:16]$
$\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low})\,[15:0]$
low | (high << 16)
31
16 15
0

Figure 18.3: Rebuilding a sliced variable

equality domain, we know that $\mathscr{S}\text{lice}^{[16:31]}_{[0:15]}(\texttt{high}) = \mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low})$, so we can simplify this expression as on Figure 18.4. Then `int rebuilt` gets the value of $\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low})$, which is known (in the equality domain), to be equal to `x`.

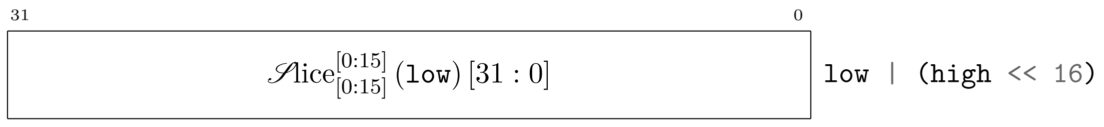

31
0
$\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low})\,[31:0]$
low | (high << 16)

Figure 18.4: Rebuilt value

This method works whatever the value of `x`. But let us detail what happens if `x` is a pointer, and thus needs itself a ghost variable $\mathscr{O}\text{ffset}(\texttt{x})$. When `x` is copied into $\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low})$, the latter gets a pointer value, and thus, requires a ghost variable for its offset which is $\mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low})\right)$. Similarly, $\mathscr{S}\text{lice}^{[16:31]}_{[0:15]}(\texttt{high})$ gets a ghost variable for its offset: $\mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}^{[16:31]}_{[0:15]}(\texttt{high})\right)$. Both these new offsets are equal to $\mathscr{O}\text{ffset}(\texttt{x})$, which is needed to prove that $\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low}) = \mathscr{S}\text{lice}^{[16:31]}_{[0:15]}(\texttt{high})$ and then, to prove that $\texttt{rebuilt} = \texttt{x}$.

Using this method, we can solve this kind of problems for any kind of variables that are representable in ghost variables. If we use a stack architecture, this domain should lie above any other domain that are used to handle the value of variables we slice. Here, we would need to put this domain above stack domain. For this example, it would work, but we will see later how this solution lacks generality.

18.1.3 Linear Combinations of Pointers

When writing low-level software, bit slicing is not the only kind of strange trick that we use. Here, we will see that linear combinations of pointers can be useful. In Listing 16.1 “Simple dynamic code with problematic handling” (page 201), we have shown a use of the difference of two pointers that does not point to the same block. This is really easy to solve by representing this value as an unevaluated linear combination. Each term is made of an integer coefficient and a ghost variable. This could be the job of a dedicated domain that handles linear combinations and which lies above the pointer domain, as term of the linear combination are also pointers.

We can think of an even more problematic case. We already talked about call gates. They allow unprivileged programs to access a privileged procedure. They contain a segment selector in which the procedure is stored and the offset inside the segment. In descriptors, most fields are not contiguous, in particular for call gates, the offset is indeed cut into two parts (see Figure 2.13 “Structure of a 32-bit call gate” (page 31)). Two problems are mixed here. First, the offset inside the segment is the difference between the procedure address and the base address of the segment: the offset is a linear combination of pointers. Then, this linear combination is split into two pieces, as explained in subsection 18.1.2 “GDT Entries” (page 231). To get a precise abstraction of this problem, we need a bit slice domain, that requires a linear combinations domain to represent sliced values, linear combinations domain needs a pointer domain to represent each term, and the pointer domain requires a numerical domain to handle offsets.

If we assume a stack architecture, domains should be ordered as cited, from top to bottom. But this is very constraining: we can only represent nesting of properties that follows this order. We can handle slices of integers, slices of pointers, linear combinations of pointers, slices of linear combinations of pointers…; but we cannot represent linear combinations of slices of pointers, as it is not the order enforced by the stack.

18.2 Related Work

A very standard and generic way to improve the precision of an analysis is to use partitioning. The point is to represent the state as a disjunction of states that are easier to abstract precisely. In these works, ghost variables are states we use in the disjunction. States can be partitioned depending on their contents (as in [Bou92]) or the context (as in [Ama+16]).

Another kind of information that may be relevant for abstraction is the history of objects. This forms also a non-standard semantics since it keeps in the current state events that are in the past, and thus are only visible in traces according to the standard semantics. [Fer00; Ven98; Ven99] use such history. In [CL05], ghost variables are used to represent expressions with unevaluated external symbols.

In all these enriched semantics, the role of ghost variables is very static. [Pér10; HP08; AČW09] are examples of works where ghost variables have a more dynamic semantics, but they are local to the domain and are not shared with outside domains.

Non-standard semantics make some pieces of information explicit to ease the abstraction, but we still need domains to abstract precisely this newly explicit quantities. A way to improve precision of domains is to make them work together. We have already explained product of domains in chapter 12 “Product of Abstract Domains” (page 155) (see also [CC79; CCF13]), and how to make a realistic reduction operator in section 12.3 “Partially Reduced product” (page 159) (see also [Cou+06b]).

There are other works on domain cooperation with dynamic support, like cofibered domains [Ven96]. They have some limitations that we try to overcome. The product we propose does not enforce a hierarchy between domains and the current support is known by every domain to improve precision.

[JMO20] also uses transformation of statements but based on expression rewriting. In this approach, several abstraction levels use rewriting rules to gradually simplify expressions. This strategy is well-fitted for function resolution in a context where there are function overloading and dynamic typing, which is crucial to analyze PYTHON source code. Given our application, we have other priorities. This is why we choose to promote a more flexible framework where statement transformation may depend on the history (see chapter 22 “Slices Domain” (page 281) for an example). It allows domains to declare new ghost variables without knowing *a priori* when they will be used; their value will be updated during the following computation steps, and used when useful without knowing precisely when they have been declared. Another requirement is to allow arbitrary predicate nesting (see section 18.1 “Motivation” (page 230)): domains are free to use ghost variables to represent any available property, even if it means that some kinds of transformations are not so straightforward.

18.3 Difficulties

We need a framework to build a composite domain that does not enforce a hierarchy. More precisely, that means that domains may allocate ghost variables dynamically and make all other domains know about these new variables. This way, ghost variables can be precisely abstracted by any involved domain.

Distributed handling of ghost variables implies that domains must communicate to know which variables are added, deleted, modified.... Each domain emits such constraints, but there is no way to order them meaningfully as there is no hierarchy. We need every ordering of these constraints to be correct.

This framework must perform reduction. In particular, with domains that need no ghost variables, it should behave just as the partially reduced product explained in [Cou+06b], and implemented in ASTRÉE.

This framework should also make soundness and termination provable modularly. Domains must individually meet some constraints that guarantee soundness and termination of the whole composite domain. This is crucial both from a theoretical and an implementation point of view. Indeed, if each domain match local conditions, any assembly of them forms a composite domain that should be sound and terminating. This reduces the complexity of proofs and it should not be redone each time another set of domains is needed. We can have a library of domains and just select the ones we need to have required precision with the best performance. Similarly, each domain can be implemented independently, which is a nice property for software engineering.

The set of living ghost variables may depend on the history on the state. For instance, we need an offset for a variable if it was assigned a pointer value. Even states with the same concretization can use different sets of ghost variable depending on the way these states were obtained. During binary operations on states (like join), even if the language guarantees that the set of living (real) variables is the same in both operands (like C does), the set of ghost variables may be different. This means that we need to be able to perform binary operations with different set of living variables. The method that we will use is to start with a unification step before performing the binary operation on resulting states that have the same set of ghost variables.

Chapter 19

Concrete Framework

While termination is an intrinsic notion, soundness of an abstraction is relative to a concrete semantics. We explain here how to insert ghost variables in the concrete semantics. To be as general as possible, we will not use a precise language, but just define its semantics as a composition of opaque primitives.

19.1 Abstraction Level and Expressions

The use case of our new framework is to represent complex properties on pointers, and thus it will replace the pointer domain in the composite domain of Astrée (see Figure 14.1 "New structure of the composite domain of Astrée" (page 179)). This has some implications on the framework. In particular, we can ignore features that have been abstracted away in overlaying domains.

In particular, struct domain has already resolved pointer dereferences, thus expressions cannot contain indirections (C prefix operator $*$). Nevertheless, operator "address of" (C prefix operator `&`) is allowed: since the new framework is made for pointer abstraction, we should be able to build pointer values. We name $\mathbb{V}$ the set of variables and $\mathbb{I}$ a set of associated values. Concrete memory states are maps of type $\mathbb{S} := \mathbb{V} \rightharpoonup \mathbb{I}$. The support of the map is the set of living variables. The set $\mathbb{I}$ shall contain all the values required to evaluate a variable in an expression. Given our abstraction level, it is usually a set of pairs made of the value of a variable and its address.

We should emphasize that variables are assumed not to alias from the point of view of pointer domain: assigning a variable never changes the value of another one. If some variables are used to represent the same memory location, it is the struct domain's responsibility to emit multiple instructions to update each variable corresponding to this location. For instance, in Listing 19.1, variable `a` can be represented in two ways: a single 4-byte variable (to store `int x`) or 4 1-byte variables (to store `char y[4]`). We respectively name them x and $y_0, \ldots, y_3$. The first assignment `a.x = 0xabcdef01;` is

```
union {
    int x;
    char y[4];
} a;
int f() {
   a.x = 0xabcdef01;
   a.y[0] = 0x02;
   // a.x == 0xabcdef02;
}
```

Listing 19.1: A single memory zone with multiple representations

translated to the assignments

$$\begin{aligned} x &\leftarrow \text{ABCDEF01H} \\ y_0 &\leftarrow x\,[0:7] \\ y_1 &\leftarrow x\,[8:15] \\ y_2 &\leftarrow x\,[16:23] \\ y_3 &\leftarrow x\,[24:31] \end{aligned}$$

using notations for slices and hexadecimal values introduced in subsection 2.1.3 "Conventions" (page 9) and section A.1 "Architecture" (page 369). Then the second assignment `a.y[0] = 0x02;` is translated to

$$\begin{aligned} y_0 &\leftarrow \text{02H} \\ x &\leftarrow \text{02H} + 2^8 y_1 + 2^{2\cdot 8} y_2 + 2^{3\cdot 8} y_3 \end{aligned}$$

The underlying domain receives 5 directives for the first assignment and 2 for the second, but from its point of view, $y_0, \ldots, y_3$ and x are independent.

Since variables do not alias, the independence of statements is easy to check. For instance, let x and y be variables, and e an expression if e does not contain x or y then assignments

$$\begin{aligned} x &\leftarrow e \\ y &\leftarrow e \end{aligned}$$

can safely be commuted.

Since we want to stay fully general about the language, we will not define the set of expressions. We only assume that they are trees using a given set of constructors, where leaves may be variables.

Let 𐀨 be a set of constructors*. Elements of 𐀨 are written $c^{/n}$ where c is the name of the constructor and $n \in \mathbb{N}$ the arity.

*This is a Linear B character. This alphabet will be used to name any mathematical object locally

Notation 19.1 – Expressions

Let V be a set of variables. We denote $\mathbb{E}_V$ the set of expression defined inductively with constructors $\text{𐀒} \uplus V$ where elements of V are used as niladic constructors.

The set V is arbitrary, and not necessarily a subset of $\mathbb{V}$. In fact, it is useful with the set of real variables and with the set of ghost variables.

Notation 19.2 – Classical expressions

We denote

$$\mathbb{E} := \mathbb{E}_{\mathbb{V}}$$

the set of expressions with variables in $\mathbb{V}$.

Later, we will need haunted (or ghost) expressions, that is expressions that involve ghost variables. The standard semantics involve only classical expressions. Haunted expressions will be useful in the enriched semantics, to express constraints on ghost variables.

Definition 19.1 – Structural inclusion

Let V be a set of variables and $(e, e') \in \mathbb{E}_V{}^2$. Let $\twoheadrightarrowtail \subseteq \mathbb{E}_V{}^2$, the relation such that[†]:

$$e \twoheadrightarrowtail e' :\Leftrightarrow \begin{pmatrix} \exists c^{/n} \in \text{𐀒} : \exists p \in [\![1, n]\!] : \exists (e_i)_{i \in [\![1, n-1]\!]} \in \mathbb{E}_V{}^{n-1} : \\ e' = c^{/n}\left(e_1, \ldots, e_{p-1}, e, e_p, \ldots, e_{n-1}\right) \end{pmatrix}$$

We write $\sqsupseteq$ the reflexive and transitive closure of $\twoheadrightarrowtail$. If $e \sqsupseteq e'$, we say that e is structurally included in e'.

This means that e is a subtree of e'.

Proposition 19.1 – Well-foundedness of structural inclusion

$\sqsupseteq$ is a well-founded order relation.

Proof. $\sqsupseteq$ is a preorder as it is reflexive and transitive by definition.

Let $(e, e') \in \mathbb{E}_V{}^2$. We assume that $e \sqsupseteq e'$ and $e' \sqsupseteq e$. If $e \twoheadrightarrowtail a_1 \twoheadrightarrowtail \cdots \twoheadrightarrowtail a_l \twoheadrightarrowtail e'$ then the height of the tree of e' is greater than the height of e, which is not possible since $e' \sqsupseteq e$. Thus, both heights are equal, and $e = e'$. The relation is thus antisymmetric, thus it is an order relation.

useful but quickly forgotten afterward, like Linear B alphabet was useful once, but gave nothing and was replaced by Greek alphabet. 𐀒 is the character for syllable "ko" or "co", like "constructors". Another benefit of Linear B characters is that they have very little background in notations, while most Greek and Hebrew letters are heavily used and almost have intrinsic meaning, now.

[†]The symbol $\twoheadrightarrowtail$ is also a Linear B character (whose shape is explicit enough for a transition relation), consequently, we will not speak about it again.

Let $(e_i)_{i\in\mathbb{N}} \in \mathbb{E}_V{}^{\mathbb{N}}$. If e_i is decreasing (according to $\sqsupseteq$), so is the sequence of heights. Since $(\mathbb{N}, \leqslant)$ is well-founded, so is $(\mathbb{E}_V, \sqsupseteq)$. □

Notation 19.3 – Variables in an expression

Let V be a set of variables and $e \in \mathbb{E}_V$. We write $\mathrm{Var}\,(e) \subseteq V$ the set of variables in e, that is

$$\begin{aligned}\mathrm{Var} : \mathbb{E}_V &\to \mathcal{P}\,(V)\\ e &\mapsto \begin{cases}\bigcup_{i=1}^{n} \mathrm{Var}\,(e_i) & \text{if } e = c^{/n}\,(e_1,\dots,e_n)\\ \{e\} & \text{if } e \in V\end{cases}\end{aligned}$$

Usually, we would define the semantics of expressions by induction over the structure. This is not necessary here, we only want the result of an affectation to update the left-hand side variable only depending on the variables in the right-hand side expression. This way, we do not even need the semantics of expressions to be compositional.

The other fundamental objects built from variables are conditions. Indeed, without conditions, a programming language is no more than a linear sequence of assignments, which is not very expressive. For the sake of simplicity, we only use conditions made of a comparison operator (with arbitrary arity) and expressions.

Notation 19.4 – Conditions

Let $\mathbb{O}$ be a set of comparison operators. For all set of variables V, we denote

$$\mathbb{C}_V := \left\{ op^{/n}\,(e_1,\dots,e_n) \;\middle|\; op^{/n} \in \mathbb{O}, (e_i)_{i\in[\![1,n]\!]} \in \mathbb{E}_V{}^n \right\}$$

and

$$\mathbb{C} := \mathbb{C}_{\mathbb{V}}$$

$\mathbb{O}$ is usually $\left\{=^{/2}, \neq^{/2}, <^{/2}, \leqslant^{/2}, >^{/2}, \geqslant^{/2}\right\}$. They are often used in infix notation, while omitting the arity.

Notation 19.5 – Variables in a condition

Let V a set of variables. For $c = op^{/n}\,(e_1,\dots,e_n) \in \mathbb{C}_V$, we write $\mathrm{Var}\,(c) \subseteq V$ the set of variables that appear in c, i.e.

$$\mathrm{Var}\,(c) := \bigcup_{i=1}^{n} \mathrm{Var}\,(e_i)$$

19.2 Semantics

We assume the concrete semantics can be built by composing the following primitives:

- *Alloc*: allocates a variable,
- *Kill*: removes a variable,
- *Assign*: assigns a variable the result of an expression,
- *Guard*: enforces a condition.

```python
if non_deterministic():
    x = 1
    y = 2
else:
    x = 2
```

Listing 19.2: A set of living variables that cannot be statically inferred

Despite our quest of generality, we need some hypotheses on the language and on primitives. The first important point to notice is that the support (i.e. set of living variables) is dynamic. We recall that deterministic memory states are elements of $\mathbb{S} := \mathbb{V} \rightharpoonup \mathbb{I}$. But in a non-deterministic semantics, we might be tempted to use $\mathcal{P}(\mathbb{S})$, but this allows states containing maps with different support. This exists in some programming language with very few static guarantees, like PYTHON. For instance, in Listing 19.2, after line 5, the non-deterministic memory state is

$$\left\{ \begin{array}{l} (x \mapsto 1, y \mapsto 2) \\ (x \mapsto 2) \end{array} \right\}$$

For the sake of simplicity, and because we are interested in C and similar languages, we restrict the set of non-deterministic states to sets in which all elements have the same support:

$$\mathbb{D} := \left\{ \mathcal{S} \in \mathcal{P}(\mathbb{S}) \,\middle|\, \forall (s, s') \in \mathcal{S}^2, \operatorname{supp}(s) = \operatorname{supp}(s') \right\}$$

Notation 19.6 – Support

For all $a \in \mathbb{D}$, we write $\operatorname{supp}(a)$ the common support of all maps in a.

Before giving the signature of primitives, we need another last notation.

Notation 19.7 – Assignments

Let V be a set of variables. We denote

$$\mathbb{A}_V := V \times \mathbb{E}_V$$

and

$$\mathbb{A} := \mathbb{A}_{\mathbb{V}}$$

The types of primitives are:
- $Alloc : \mathbb{V} \times \mathbb{D} \to \mathbb{D}$
- $Kill : \mathbb{V} \times \mathbb{D} \to \mathbb{D}$
- $Assign : \mathbb{A} \times \mathbb{D} \to \mathbb{D}$
- $Guard : \mathbb{C} \times \mathbb{D} \to \mathbb{D}$

We will give the hypotheses that we need on these primitives by category. First, let us see the support-related assumptions.

Informally, we expect *Alloc* to add a variable to the support, and *Kill* to remove a variable. The other primitives should not change the support.

Axiom 19.1 – Effect of primitives on the support

- $\forall v \in \mathbb{V}, \forall d \in \mathbb{D}, \operatorname{supp}(Alloc(v,d)) = \operatorname{supp}(d) \cup \{v\}$
- $\forall v \in \mathbb{V}, \forall d \in \mathbb{D}, \operatorname{supp}(Kill(v,d)) = \operatorname{supp}(d) \setminus \{v\}$
- $\forall a \in \mathbb{A}, \forall c \in \mathbb{C}, \forall d \in \mathbb{D}, \operatorname{supp}(Assign(a,d)) = \operatorname{supp}(d)$
- $\forall a \in \mathbb{A}, \forall c \in \mathbb{C}, \forall d \in \mathbb{D}, \operatorname{supp}(Guard(c,d)) = \operatorname{supp}(d)$

We also need to assume that binary operations, like unions at the end of branching, are performed with operands with the same support and that variables that appear in assignments and guards are all living at this point.

The second class of hypotheses is about variables that are read or written by each primitive. We implicitly lift notations on functions on elements of $\mathbb{D}$. For instance, given $d \in \mathbb{D}$ and $V \subseteq \mathbb{V}$, we write $d_{|v}$ for $\left\{f_{|v} \;\middle|\; f \in d\right\}$, and $d[a \mapsto b]$ for $\{f[a \mapsto b] \mid f \in d\}$.

Axiom 19.2 – Variable allocation and killing

- $\forall v \in \mathbb{V}, \forall d \in \mathbb{D}, v \in \operatorname{supp}(d) \Rightarrow Alloc(v,d) = d$
- $\forall v \in \mathbb{V}, \forall d \in \mathbb{D}, v \notin \operatorname{supp}(d) \Rightarrow Alloc(v,d)_{|\operatorname{supp}(d)} = d$
- $\forall v \in \mathbb{V}, \forall d \in \mathbb{D}, v \notin \operatorname{supp}(d) \Rightarrow Kill(v,d) = d$
- $\forall v \in \mathbb{V}, \forall d \in \mathbb{D}, v \in \operatorname{supp}(d) \Rightarrow Kill(v,d) = d_{|\operatorname{supp}(d)\setminus\{v\}}$

We need primitives to be $\cup$-preserving. This implies that they are lifts of primitives whose second parameter has type $\mathbb{S}$ rather than $\mathbb{D}$. This condition makes following axioms more powerful as they should stand on each map of any state of $\mathbb{D}$.

Axiom 19.3 – Primitives are $\cup$-morphisms

Let I be a set. Let $(d_i)_{i\in I} \in \mathbb{D}^I$.

- $\forall v \in \mathbb{V}, Alloc\left(v, \bigcup_{i\in I} d_i\right) = \bigcup_{i\in I} Alloc(v, d_i)$
- $\forall v \in \mathbb{V}, Kill\left(v, \bigcup_{i\in I} d_i\right) = \bigcup_{i\in I} Kill(v, d_i)$
- $\forall v \in \mathbb{V}, \forall e \in \mathbb{E}, Assign\left((v,e), \bigcup_{i\in I} d_i\right) = \bigcup_{i\in I} Assign((v,e), d_i)$

– $\forall c \in \mathbb{C}, \mathit{Guard}\left(c, \bigcup_{i\in I} d_i\right) = \bigcup_{i\in I} \mathit{Guard}\,(c, d_i)$

Following requirements are common to all reasonable languages. They specify that the only variables read or written are those that appear in expressions. This is possible since there is no aliasing and no dereference.

Axiom 19.4 – Variables read and written by *Assign*

$\mathit{Assign}\,((v,e),d)$ writes only variable v:

$$\begin{aligned}&\forall v \in \mathbb{V}, \forall e \in \mathbb{E}, \forall d \in \mathbb{D},\\ &\qquad (\{v\} \cup \mathrm{Var}\,(e) \subseteq \mathrm{supp}\,(d)) \Rightarrow d_{|\mathrm{supp}(d)\setminus\{v\}} = \mathit{Assign}\,((v,e),d)_{|\mathrm{supp}(d)\setminus\{v\}}\end{aligned}$$

$\mathit{Assign}\,((v,e),d)$ reads only variables in $\mathrm{Var}\,(e)$:

$$\begin{aligned}&\forall v \in \mathbb{V}, \forall e \in \mathbb{E}, \forall d \in \mathbb{D}, \forall V \subseteq \mathrm{supp}\,(d) \setminus (\mathrm{Var}\,(e) \cup \{v\}), \forall (a_i)_{i\in V} \in \mathbb{I}^V,\\ &(\{v\}\cup\mathrm{Var}\,(e) \subseteq \mathrm{supp}\,(d)) \Rightarrow \mathit{Assign}\,((v,e),d)_{|\{v\}} = \mathit{Assign}\left((v,e), d\,[i \mapsto a_i]_{i\in V}\right)_{|\{v\}}\end{aligned}$$

Axiom 19.5 – Variables read and written by *Guard*

$\mathit{Guard}\,(c,d)$ enforces a condition but does not modify the state:

$$\forall c \in \mathbb{C}, \forall d \in \mathbb{C}, \mathrm{Var}\,(c) \subseteq \mathrm{supp}\,(d) \Rightarrow \mathit{Guard}\,(c,d) \subseteq d$$

$\mathit{Guard}\,(c,d)$ only depends on value of variables in $\mathrm{Var}\,(c)$:

$$\begin{aligned}&\forall c \in \mathbb{C}, \forall d \in \mathbb{C}, \forall V \subseteq \mathrm{supp}\,(d) \setminus \mathrm{Var}\,(e), \forall (a_i)_{i\in V} \in \mathbb{I}^V,\\ &\qquad \mathrm{Var}\,(c) \subseteq \mathrm{supp}\,(d) \Rightarrow \mathit{Guard}\,(c,d)\,[i \mapsto a_i]_{i\in V} = \mathit{Guard}\left(c, d\,[i \mapsto a_i]_{i\in V}\right)\end{aligned}$$

Domains communicate constraints about variables in an arbitrary order. Since this order has nothing to do with the semantics, we need our construction to be correct independently of the chosen order. Since constraints are enforced using primitives, we need to be able to commute them.

Axiom 19.6 – Commutation of *Assign*

$$\begin{aligned}&\forall a \in \mathbb{D}, \forall (e,f) \in \mathbb{E}^2, \forall (v,w) \in \mathbb{V}^2, v \neq w \wedge \{v,x\} \cap (\mathrm{Var}\,(e) \cap \mathrm{Var}\,(f)) = \varnothing \Rightarrow\\ &\qquad \mathit{Assign}\,((v,e), \mathit{Assign}\,((w,f),a)) = \mathit{Assign}\,((w,f), \mathit{Assign}\,((v,e),a))\end{aligned}$$

Axiom 19.7 – Commutation of *Guard*

$$\forall d \in \mathbb{D}, \forall (c, d) \in \mathbb{C}^2, \mathit{Guard}\,(c, \mathit{Guard}\,(d, a)) = \mathit{Guard}\,(d, \mathit{Guard}\,(c, a))$$

Axiom 19.8 – Commutation of *Assign* and *Guard*

$$\forall a \in \mathbb{D}, \forall e \in \mathbb{E}, \forall op^{/n} \in \mathbb{O}, \forall (f_i)_{i \in [\![1,n]\!]} \in \mathbb{E}^n, \forall v \in \mathbb{V}, v \notin \mathrm{Var}\,(e) \cup \bigcup_{i=1}^{n} f_i \Rightarrow$$
$$\mathit{Assign}\,\big((v, e), \mathit{Guard}\,\big(op^{/n}\,(f_1, \ldots, f_n)\,, a\big)\big) =$$
$$\mathit{Guard}\,\big(op^{/n}\,(f_1, \ldots, f_n)\,, \mathit{Assign}\,((v, e), a)\big)$$

These assumptions are quite reasonable and allow commuting statements as long as they are independent enough. Once again, we rely strongly on the fact that variables do not alias each other. Otherwise, these syntactic conditions would not have been strong enough.

For the abstraction, we will need a less common assumption that does not restrict the expressive power.

Assumption 19.1 – Restriction on assignments

We assume that any assignment, the left-hand side does not appear in the right-hand side.

For instance, this assumption prevents to run `x = x + 1;`. But there is a trivial workaround: `y = x + 1; x = y;`, where `y` is a helper variable used only for such assignment. A fresh variable can be picked each time as an extra guarantee of non-interference through this intermediate variable.

19.3 Ghost Variables

Ghost variables are used to help to abstract other variables. In subsection 18.1.2 "GDT Entries" (page 231), we have illustrated how to use ghost variables to handle sliced variables. In Listing 18.2 "A complicated way to do nothing" (page 231), we have seen that we needed one ghost variable for `low` and one for `high`. But if `x` has a pointer value, so are ghost variables $\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}$ (`low`) and $\mathscr{S}\mathrm{lice}^{[16:31]}_{[0:15]}$ (`high`). Thus, these three variables need ghost variables to store their offsets: more generally, even ghost variables may need other ghost variables. In that goal, we define ghost variables inductively: each ghost variable is made of a constructor and a variable. The variable is the one that the ghost variable help to represent, and the constructor explains the relation between them (what the ghost variable means). For instance, given a variable `x`, $\mathscr{O}\mathrm{ffset}$ (`x`) is a variable representing the offset of `x`: $\mathscr{O}\mathrm{ffset}$ explains the relation between $\mathscr{O}\mathrm{ffset}$ (`x`) and `x`: the former is the offset of the latter.

Definition 19.2 – Roles

Let $\mathcal{R}$ be a set of unary constructors. These constructors are called roles.

For instance $\mathscr{O}$ffset and $\mathscr{S}$lice$^{[a:b]}_{[c:c+b-a]}$ (for all $a \in [\![0, 31]\!]$, $b \in [\![a, 31]\!]$ and $c \in [\![0, 31-b+a]\!]$) are roles. The name "role" is because these constructors hold the meaning of ghost variables we build with them. From now on, everything written in this Round Hand style (like $\mathscr{O}$, $\mathscr{R}$ or $\mathscr{S}$) is about ghost variables and haunted objects (everything that contains ghost variables).

Notation 19.8 – Ghost variables

We denote by $\mathscr{V}$ the inductively defined set of all variables, starting with real variables (in $\mathbb{V}$).
Formally, for all $i \in \mathbb{N}$,

$$\mathscr{V}_i := \begin{cases} \mathbb{V} & \text{if } i = 0 \\ \{\mathscr{R}(v) \mid \mathscr{R} \in \mathcal{R}, v \in \mathscr{V}_{i-1}\} & \text{otherwise} \end{cases}$$

and

$$\mathscr{V} := \bigcup_{i \in \mathbb{N}} \mathscr{V}_i$$

Elements of $\mathscr{V}$ are called ghost variables.

Notation 19.9 – Ghost expressions, conditions and assignments

$$\mathscr{E} := \mathbb{E}_{\mathscr{V}}$$
$$\mathscr{C} := \mathbb{C}_{\mathscr{V}}$$
$$\mathscr{A} := \mathbb{A}_{\mathscr{V}}$$

Definition 19.3 – Ghostly ordering

Let $\lhd \subseteq \mathscr{V}^2$ such that

$$\forall (v, w) \in \mathscr{V}^2, \Big(v \lhd w \Leftrightarrow \exists n \in \mathbb{N}^* : \exists (\mathscr{R}_i)_{i \in [\![1,n]\!]} \in \mathcal{R}^n : v = \mathscr{R}_1(\dots \mathscr{R}_n(w) \dots)\Big)$$

Let $\trianglelefteq$ be the reflexive closure of $\lhd$, i.e.

$$\forall (v, w) \in \mathscr{V}^2, \Big(v \trianglelefteq w \Leftrightarrow \exists n \in \mathbb{N} : \exists (\mathscr{R}_i)_{i \in [\![1,n]\!]} \in \mathcal{R}^n : v = \mathscr{R}_1(\dots \mathscr{R}_n(w) \dots)\Big)$$

If $x \lhd y$, we say that x is less real, or ghostlier, than y.
We extend this relation to $\mathscr{E}^2$ by

$$\forall (e, e') \in \mathscr{E}^2, \big(\forall v' \in \mathrm{Var}\,(e'), \exists v \in \mathrm{Var}\,(e) : v' \trianglelefteq v\big) \Leftrightarrow e' \trianglelefteq e$$

That is every variable in e' is ghostlier than a variable in e.

Proposition 19.2

Relation $\trianglelefteq$ is an order on $\mathscr{V}$.

Proof. We can see a ghost variable as a list of roles and a real variable. Given two variables v and w, $v \trianglelefteq w$ if they share the same real variable and if the list of roles of w is a suffix of the list of roles of v. From this interpretation, the result is obvious:

– reflexivity: a list is a suffix of itself;
– antisymmetry: if each list is a suffix of the other, they are equal;
– transitivity: if a list is a suffix of a second and the second is a suffix of a third, then the first list is a suffix of the third.

□

Proposition 19.3

Relation $\trianglelefteq$ is a preorder on $\mathscr{E}$.

Proof. Given expressions e and e', $e' \trianglelefteq e$ if e' is made of variables that appear in e of variables ghostlier than the ones that appear in e. Reflexivity is thus obvious: e is made of variables contained in e.

Transitivity is hardly more complicated. Let e'' a third expression and let us assume that $e'' \trianglelefteq e'$ and $e' \trianglelefteq e$, then variables of e' are ghostlier than variables in e and, a fortiori, variables ghostlier than variables of e' are ghostlier than variables of e. Thus, variables in e'' are ghostlier than variables of e, hence $e'' \trianglelefteq e$. □

Ghost variables have a meaning: we introduced them to keep some quantities and this should be enforced.

Definition 19.4 – Semantics of ghost variables

Let $\llparenthesis \_ \rrparenthesis : (\mathscr{V} \setminus \mathbb{V}) \to \mathcal{P}(\mathscr{V} \rightharpoonup \mathbb{I})$ be the semantics of ghost variables, such that

$$\forall v \in \mathscr{V}, \forall \mathscr{R} \in \mathcal{R}, \llparenthesis \mathscr{R}(v) \rrparenthesis = \left\{ s \in \mathbb{S} \;\middle|\; \exists t \in \llparenthesis \mathscr{R}(v) \rrparenthesis : s_{|\{r(v) \,|\, r \in \mathcal{R}\}} = t_{|\{r(v) \,|\, r \in \mathcal{R}\}} \right\}$$

The inclusion is obvious, we just need to take $s = t$, but the converse inclusion is not. It means that the semantics of a ghost variable is not sensitive to the value of non-parent or non-sibling variables. If a state is in the semantics of a ghost variable, changing such an unrelated variable gives a state that should still be in the semantics. In other words, the semantics of a ghost variable only involves this variable, its immediate parent and siblings. As a direct consequence, the value of a real variable constrains only ghost variables under it. For instance, the semantics of $\mathscr{A}\mathrm{lice}^{[a:b]}_{[c:d]}(x)$ is the subset of $\mathscr{V} \rightharpoonup \mathbb{I}$ where bits a to b of $\mathscr{A}\mathrm{lice}^{[a:b]}_{[c:d]}(x)$ are indeed bits c to d of variable x. The semantics is usually carried by the outermost role: it may be more general, but in this case, the following may not be easy to implement in the framework we will introduce.

Each domain brings its own set of roles and, to keep modularity, it should not depend on roles defined by other domains.

As we add new variables, we need to take them into account in the memory state. Thus, we extend $\mathbb{S}$ so that its domain contains ghost variables. We need to add a restriction: if a ghost variable is living, all less ghostly (more real) variables are living too.

$$\mathbb{S} := \left\{ s : \mathscr{V} \rightharpoonup \mathbb{I} \;\middle|\; \begin{array}{l} \forall v \in \mathscr{V}, v \in \operatorname{supp}(s) \Rightarrow v^{\uparrow \trianglelefteq} \subseteq \operatorname{supp}(s) \\ \forall v \in \operatorname{supp}(s) \setminus \mathbb{V}, s \in \llparenthesis v \rrparenthesis \end{array} \right\}$$

where $v^{\uparrow \trianglelefteq}$ is the set of variables greater than v with respect to $\trianglelefteq$ (see notation 11.2 (page 138) and section A.3 “Set Theory” (page 371)). Likewise, we need to extend $\mathbb{D}$. We have explained that elements of $\mathbb{D}$ must be set of maps with the same support. Here, this condition is slightly relaxed as we only require that all maps must have the same set of living real variables. Yet, maps contained in a single element of $\mathbb{D}$ may have different sets of living ghost variables.

$$\mathbb{D} := \left\{ \mathcal{S} \in \mathcal{P}(\mathbb{S}) \;\middle|\; \forall (s, s') \in \mathcal{S}^2, \operatorname{supp}(s) \cap \mathbb{V} = \operatorname{supp}\left(s'\right) \cap \mathbb{V} \right\}$$

We assume that we can naturally extend primitives to ghost variables, including their axioms. The types of primitives become:

- $\mathit{Alloc} : \mathscr{V} \times \mathbb{D} \to \mathbb{D}$
- $\mathit{Kill} : \mathscr{V} \times \mathbb{D} \to \mathbb{D}$
- $\mathit{Assign} : \mathscr{A} \times \mathbb{D} \to \mathbb{D}$
- $\mathit{Guard} : \mathscr{C} \times \mathbb{D} \to \mathbb{D}$

19.4 Ghost Constraints

Values of ghost variables are determined by their semantics and values of real variables. In our framework, for modularity reasons, each domain can ask to create new ghost variables. Then, each ghost variable is managed by the domain that created it: we assume this is the only domain that knows the semantics of the ghost variable. Since other domains do not know the semantics of ghost variables that come from this domain, they would be supremely imprecise about the value of this ghost variable (they will default to $\top$). In this case, it would be useless to share its ghost variables with other domains as they cannot say non-trivial properties about them. To do better, domains must communicate information about ghost variables.

We will now take a look at the constraints exchanged between domains. The first, obvious, kind of constraints are simply conditions as they express relations between variables. The second kind of constraints are elements of $\mathscr{A}$, but with a slightly different interpretation. The ordered pair (x, e) should be interpreted as the comparison $x = e$, but with a restriction: when this constraint is used x should be perfectly unknown ($\top$) and $x \notin \operatorname{Var}(e)$. In this context, guarding by the condition $x = e$ is just as the assignment $x \leftarrow e$. They are called directed constraints. This is interesting because running an assignment is much simpler and faster than enforcing an arbitrary comparison. This may look like a mere implementation concern, but reasons that allow using directed constraints are quite fundamental and should be shown in the formalization.

Definition 19.5 – Elementary ghost constraints

Let V be a set of variables. We denote

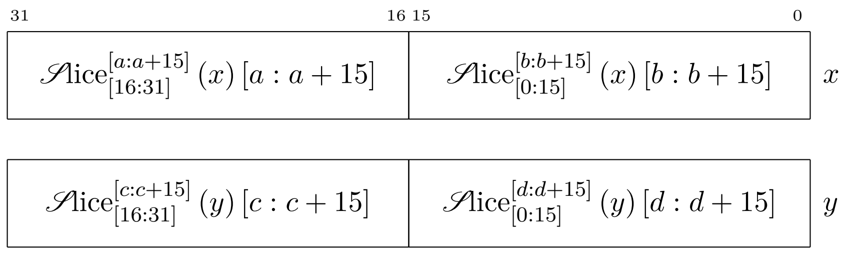
$$\mathbb{U}_V := \mathbb{A}_V \uplus \mathbb{C}_V$$

and, in particular,

$$\mathscr{U} := \mathbb{U}_{\mathscr{V}}$$

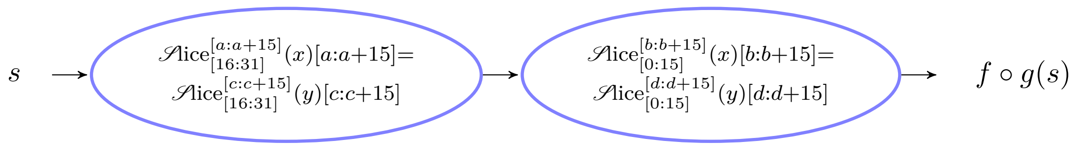
the set of elementary ghost constraints.

Letter $\mathscr{U}$ is due to the unary nature of these constraints: when enforced, they take a single precondition as parameter and return a single postcondition.



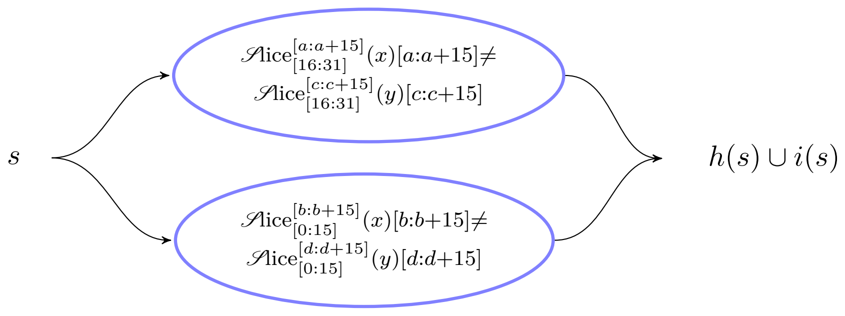
31
16 15
0
$\mathscr{A}\mathrm{lice}^{[a:a+15]}_{[16:31]}(x)\,[a:a+15]$
$\mathscr{A}\mathrm{lice}^{[b:b+15]}_{[0:15]}(x)\,[b:b+15]$
x
$\mathscr{A}\mathrm{lice}^{[c:c+15]}_{[16:31]}(y)\,[c:c+15]$
$\mathscr{A}\mathrm{lice}^{[d:d+15]}_{[0:15]}(y)\,[d:d+15]$
y

Figure 19.1: Two sliced variables with same layout



s
$\mathscr{A}\mathrm{lice}^{[a:a+15]}_{[16:31]}(x)[a:a+15]=$
$\mathscr{A}\mathrm{lice}^{[c:c+15]}_{[16:31]}(y)[c:c+15]$
$\mathscr{A}\mathrm{lice}^{[b:b+15]}_{[0:15]}(x)[b:b+15]=$
$\mathscr{A}\mathrm{lice}^{[d:d+15]}_{[0:15]}(y)[d:d+15]$
$f \circ g(s)$

Figure 19.2: An example of a constraint-DAG for a conjunction



s
$\mathscr{A}\mathrm{lice}^{[a:a+15]}_{[16:31]}(x)[a:a+15]\neq$
$\mathscr{A}\mathrm{lice}^{[c:c+15]}_{[16:31]}(y)[c:c+15]$
$\mathscr{A}\mathrm{lice}^{[b:b+15]}_{[0:15]}(x)[b:b+15]\neq$
$\mathscr{A}\mathrm{lice}^{[d:d+15]}_{[0:15]}(y)[d:d+15]$
$h(s) \cup i(s)$

Figure 19.3: An example of a constraint-DAG for a disjunction

Constraints that domains communicate are not elementary: it would not have been expressive enough. For instance, let us assume we have two (real) 32-bit variables x and y that have been built using 16-bit slices. This is illustrated on Figure 19.1, where a, b, c and d are arbitrary known integers. The guard $x = y$ can be translated as

$$\begin{gathered}\mathscr{S}\text{lice}_{[16:31]}^{[a:a+15]}(x)\,[a : a+15] = \mathscr{S}\text{lice}_{[16:31]}^{[c:c+15]}(y)\,[c : c+15]\\ \bigwedge \\ \mathscr{S}\text{lice}_{[0:15]}^{[b:b+15]}(x)\,[b : b+15] = \mathscr{S}\text{lice}_{[0:15]}^{[d:d+15]}(y)\,[d : d+15]\end{gathered}$$

which is, using C operators

$$\begin{gathered}\left(\mathscr{S}\text{lice}_{[16:31]}^{[a:a+15]}(x) \;\texttt{>>}\; a\right) \;\texttt{\&}\; \left(2^{16}-1\right) = \left(\mathscr{S}\text{lice}_{[16:31]}^{[c:c+15]}(y) \;\texttt{>>}\; c\right) \;\texttt{\&}\; \left(2^{16}-1\right)\\ \bigwedge \\ \left(\mathscr{S}\text{lice}_{[0:15]}^{[b:b+15]}(x) \;\texttt{>>}\; b\right) \;\texttt{\&}\; \left(2^{16}-1\right) = \left(\mathscr{S}\text{lice}_{[0:15]}^{[d:d+15]}(y) \;\texttt{>>}\; d\right) \;\texttt{\&}\; \left(2^{16}-1\right)\end{gathered}$$

This is a conjunction of elementary ghost constraints. This can be run by guarding by this both conditions sequentially. If we write $[\![c]\!] : \mathbb{D} \to \mathbb{D}$ the semantics of a constraint c, to enforce $x = y$ in the state $s \in \mathbb{D}$, we need to compute $f \circ g(s)$ where

$$\begin{aligned} f &= \left[\!\!\left[\mathscr{S}\text{lice}_{[16:31]}^{[a:a+15]}(x)\,[a : a+15] = \mathscr{S}\text{lice}_{[16:31]}^{[c:c+15]}(y)\,[c : c+15]\right]\!\!\right]\\ g &= \left[\!\!\left[\mathscr{S}\text{lice}_{[0:15]}^{[b:b+15]}(x)\,[b : b+15] = \mathscr{S}\text{lice}_{[0:15]}^{[d:d+15]}(y)\,[d : d+15]\right]\!\!\right]\end{aligned}$$

Similarly, the guard $x \neq y$ implies

$$\begin{gathered}\mathscr{S}\text{lice}_{[16:31]}^{[a:a+15]}(x)\,[a : a+15] \neq \mathscr{S}\text{lice}_{[16:31]}^{[c:c+15]}(y)\,[c : c+15]\\ \bigvee \\ \mathscr{S}\text{lice}_{[0:15]}^{[b:b+15]}(x)\,[b : b+15] \neq \mathscr{S}\text{lice}_{[0:15]}^{[d:d+15]}(y)\,[d : d+15]\end{gathered}$$

Thus, to enforce $x \neq y$ from the state s, we compute $h(s) \cup i(s)$ where

$$\begin{aligned} h &= \left[\!\!\left[\mathscr{S}\text{lice}_{[16:31]}^{[a:a+15]}(x)\,[a : a+15] \neq \mathscr{S}\text{lice}_{[16:31]}^{[c:c+15]}(y)\,[c : c+15]\right]\!\!\right]\\ i &= \left[\!\!\left[\mathscr{S}\text{lice}_{[0:15]}^{[b:b+15]}(x)\,[b : b+15] \neq \mathscr{S}\text{lice}_{[0:15]}^{[d:d+15]}(y)\,[d : d+15]\right]\!\!\right]\end{aligned}$$

Thus compound (non-elementary) ghost constraints should be able to express both sequencing and branching, while still being able to run it easily without worrying about termination. For these reasons, we choose DAGs as compound ghost constraints. In fact, the exact form of the intermediate language does not matter. If needed, it could include other terminating constructs like constant-bounded for-style loops. We choose this language made of DAGs because it is both simple and expressive enough. For instance the DAG corresponding to the guard $x = y$ is shown on Figure 19.2, and the DAG for $x \neq y$ is shown on Figure 19.3.

DAGs allow even more complex structures, as shown on Figure 19.4. An interesting point is that DAGs allow sharing intermediate computations. In this example, $\eta(s)$ appears twice in the expression but can be only computed once in the DAG.

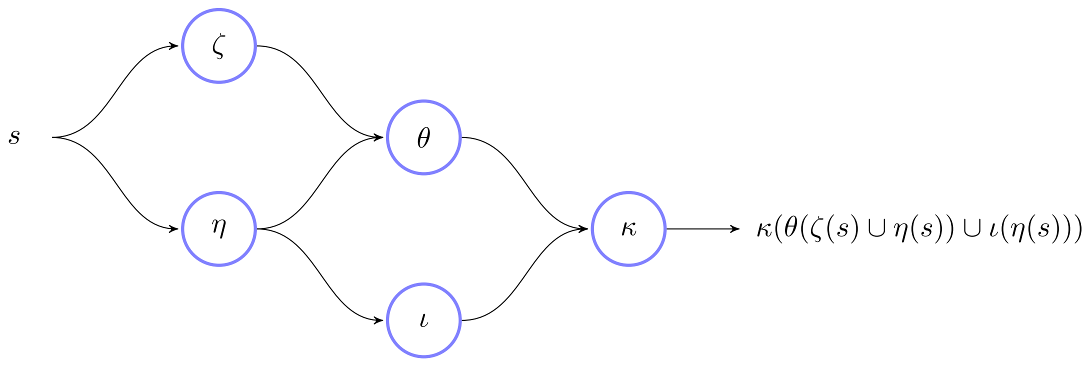


Figure 19.4: A more complicated constraint DAG

Definition 19.6 – Constraint-graph

A constraint-graph (or constraint-DAG) is a 3-tuple (N, E, u) where

- (N, E) is a DAG with exactly one source and exactly one sink:
  - N is an arbitrary finite set of nodes,
  - $E \subseteq N^2$ is the set of edges,
  - $\exists r : N \to \mathbb{N} : \forall (a, b) \in E, r(a) < r(b)$ (graph is acyclic),
  - $\exists! s \in N : (\forall (a, b) \in E, b \neq s)$ (there is exactly one source),
  - $\forall s \in N, (\forall (a, b) \in E, b \neq s) \Rightarrow (\exists (a, b) \in E : s = a)$ (the source has a successor),
  - $\exists! s \in N : (\forall (a, b) \in E, a \neq s)$ (there is exactly one sink),
  - $\forall s \in N, (\forall (a, b) \in E, a \neq s) \Rightarrow (\exists (a, b) \in E : s = b)$ (the sink has a predecessor),
- $u : N \rightharpoonup \mathscr{U}$ associates constraint to nodes of the DAG.

We can remark that constraint-graphs are always connected since there is exactly one source and one sink. Moreover, we can see that a constraint graph cannot be empty (for the same reason), and it cannot have a single node, since the source always has a successor and the sink always has a predecessor. We can still build a constraint-graph that does nothing by using 2 linked nodes, and an empty map u, as shown on Figure 19.5. Such a trivial graph is useful when there is no ghost variable to constraint. They especially come from domains that does not create ghost variable and thus have nothing to refine, for instance, see section 23.1 “Plain Old Numeric Domain” (page 293).

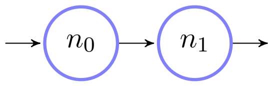


Figure 19.5: A trivial constraint-DAG

Notation 19.10 – Set of constraint-DAGs

We write $\mathscr{G}$ the set of constraint-DAGs.

Notation 19.11 – Source and sink

Given $g \in \mathscr{G}$ a constraint-DAG, we write $\overset{\circ\!-}{g}$ the source of g and $\overset{-\!\circ}{g}$ its sink.

Notation 19.12

Given $c \in \mathscr{U}$ and $g = (N, E, u) \in \mathscr{G}$, we write

$$c \in g :\Leftrightarrow \exists v \in V : u(v) = c$$

This notation allows easily writing that an elementary ghost constraint appears in a constraint-graph.

Definition 19.7 – Semantics of a constraint-DAG

Let $g = (N, E, u) \in \mathscr{G}$ be a constraint graph.
Given $n \in N$, we define $\langle\!\langle g \rangle\!\rangle_n$ the semantics of the node n as

$$\begin{aligned} \langle\!\langle g \rangle\!\rangle_n &: \mathbb{D} \to \mathbb{D} \\ a &\mapsto \begin{cases} \llbracket u(n) \rrbracket \left(\bigcup\limits_{p \in \{p \in N \mid (p,n) \in E\}} \langle\!\langle g \rangle\!\rangle_p (a) \right) & \text{if } n \neq \overset{\circ\!-}{g} \wedge n \in \operatorname{supp}(u) \\ \bigcup\limits_{p \in \{p \in N \mid (p,n) \in E\}} \langle\!\langle g \rangle\!\rangle_p (a) & \text{if } n \neq \overset{\circ\!-}{g} \wedge n \notin \operatorname{supp}(u) \\ a & \text{if } n = \overset{\circ\!-}{g} \end{cases} \end{aligned}$$

We define $\langle\!\langle g \rangle\!\rangle : \mathbb{D} \to \mathbb{D}$ the semantics of the graph g as

$$\langle\!\langle g \rangle\!\rangle := \langle\!\langle g \rangle\!\rangle_{\overset{-\!\circ}{g}}$$

Proof. This semantics is well-defined since the graph is acyclic. This can be formally stated using the map r that appears in definition 19.6 “Constraint-graph” (page 250). □

The semantics of a constraint-DAG is computed by guarding by the condition of each node, and computing the union of properties from converging edges.

Definition 19.8 – Constraint-DAG soundness

A constraint-graph $g \in \mathscr{G}$ is said to be sound if

$$\forall a \in \mathbb{D}, \left\{ s \in \mathbb{S} \;\middle|\; s \in a \wedge s \in \bigcap_{v \in \operatorname{supp}(a) \setminus \mathbb{V}} \llparenthesis v \rrparenthesis \right\} \subseteq \langle\!\langle g \rangle\!\rangle (a)$$

In other words, when one uses the constraint defined by the graph, we exclude no legal environments: all memory states that match the semantics of ghost variables are kept.

Though any sound graph can be used, there are several ways to build them systematically. Here is a non-exhaustive list that covers most common cases.

– If we test equality between variables x and y in a context where both $\mathscr{R}(x)$ and $\mathscr{R}(y)$ exist, for a given role $\mathscr{R}$, and the implication $x = y \Rightarrow f(\mathscr{R}(x)) = g(\mathscr{R}(y))$ is true in all states (with f and g two functions simple enough to be expressed as an expression in the language), we can generate the one-node graph containing the unique constraint $f(\mathscr{R}(x)) = g(\mathscr{R}(y))$. If several such implications hold, we may simply sequence all graphs in an arbitrary order (since these conditions do not interfere). This is the general case of the example $x = y$ (Figure 19.2).

– To test difference between variables x and y in a context where $(\mathscr{R}_i(x))_{i\in[\![1,n]\!]}$ and $(\mathscr{R}_i(y))_{i\in[\![1,n]\!]}$ exist, for a given family of roles $(\mathscr{R}_i)_{i\in[\![1,n]\!]}$ and the implication $x \neq y \Rightarrow \bigvee_{i\in[\![1,n]\!]} f_i(\mathscr{R}_i(x)) \neq g_i(\mathscr{R}_i(y))$ holds in any state, we can generate the graph that joins the results of these n conditions taken separately. This is like the comparison $x \neq y$ (Figure 19.3).

– Some constraints come from the language semantics but can be reinterpreted with ghost variables. E.g. since the ghost variable offset must coincide with the C standard offset, the latter can be substituted by the former.

Now we have ghost variables and tools to work on them, we can use all that to build cooperating abstract domains with ghost variables.

Chapter 20

Generic Abstract Domains and Reduced Product

Building cooperating domains that use ghost variables is not a straightforward abstraction of the concrete framework described previously. We need to make domains so that we can easily combine them, while keeping distributed the burden of ghost variable management and proofs of soundness and termination.

We will first exhibit the signature of such domains with required assumptions. In next sections, we will see how we can use components of this signature to make them cooperate in a correct and terminating way.

20.1 Generic Abstract Domain

20.1.1 A Detailed Execution of an Example with Ghost Variables

In this chapter, we will study an example that sublimates the difficulties we highlighted, while being short and quite simple. It is given on Listing 20.1. This snippet shows how to cut and recombine a pointer using bitwise operations. At the end, we want to show that $x = y$. We have already seen how ghost variables help to solve this kind of examples, but we have not shown how to do it with separate domains.

```c
int* x = &t[idx]; // Given int t[] and int idx
int low = x & 0xffff;
int high = x >> 16;
int* y = low | (high << 16);
```

Listing 20.1: A simple example that needs ghost variables

Since the management of ghost variables is distributed, we have to associate each of them with a single domain that will have the right to create and delete them, and which is the only one that is assumed to know the semantics. There is a formal rub: we need this belonging relation in the definition of domains, but we need domains to define belonging. To break the cycle, since we are not yet able to define the signature of an abstract domain, we introduce domain names.

Definition 20.1 – Domain names

Let $\mathfrak{N}$ a set of domain names.

For the purpose of Listing 20.1, we need 3 domains:

– equality, because we aim to prove an equality property;
– base-offset pointers, to represent pointer values;
– slices domain, to handle bitwise operations.

Thus, we assume $\mathfrak{N}$ contains the names of these domains. We will call them respectively $\mathfrak{Equality}$, $\mathfrak{Offset}$ and $\mathfrak{Slice}$. For this example, we can indifferently assume that $\mathfrak{N}$ is restricted to these only 3 names, or contains additional useless names.

Of course, when using several domains, they should have distinct names.

Definition 20.2 – Owner of a role

We define $\mathfrak{o} : \mathcal{R} \to \mathfrak{N}$ the map that, given a role, returns the domain to which it belongs.
Given $\mathscr{R} \in \mathcal{R}$ and $d \in \mathfrak{N}$, when $\mathfrak{o}(\mathscr{R}) = d$, we say that d is the owner of $\mathscr{R}$, or that $\mathscr{R}$ belongs to domains d. We write $\mathscr{R} \Subset d$.
We extend this notion to variables:

$$\forall v \in \mathscr{V}, \forall \mathscr{R} \in \mathcal{R}, \mathscr{R}(v) \Subset \mathfrak{o}(\mathscr{R})$$

The belonging relation is purely syntactic: the outermost role is enough to decide the owner of a ghost variable. Real variables are not owned. We can also emphasize that there is only one owner for each ghost variable (i.e. variables of $\mathscr{V} \setminus \mathbb{V}$).

Domain $\mathfrak{Slice}$ owns all roles of the form $\mathscr{S}\text{lice}^{[a:b]}_{[c:d]}$ that we have seen previously. Domain $\mathfrak{Offset}$ needs only one role: $\mathscr{O}\text{ffset}$. The last domain, $\mathfrak{Equality}$ is only in charge of remembering equivalence classes and thus does not need any ghost variable, and consequently owns no role.

We will detail the execution of line 4. Since this statement demonstrate all features the first three lines need, we will not show detail execution of them. Using our three domains, after line 3, the abstract state can be written

$$\left.\begin{array}{rc}
 & \begin{array}{cc} 31 \hspace{12em} 16 & 15 \hspace{12em} 0 \end{array} \\
\mathtt{low} = & \begin{array}{|c|c|} \hline \hspace{5em} 0 \hspace{5em} & \mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}(\mathtt{low})\,[0:15] \\ \hline \end{array} \\
 & \begin{array}{cc} 31 \hspace{12em} 16 & 15 \hspace{12em} 0 \end{array} \\
\mathtt{high} = & \begin{array}{|c|c|} \hline \hspace{5em} 0 \hspace{5em} & \mathscr{S}\mathrm{lice}^{[16:31]}_{[0:15]}(\mathtt{high})\,[16:31] \\ \hline \end{array}
\end{array}\right\}\ \mathfrak{Slice}$$

$$\left.\begin{array}{r}
\mathtt{x} = \mathtt{t} + \mathscr{O}\mathrm{ffset}\,(\mathtt{x}) \\
\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}(\mathtt{low}) = \mathtt{t} + \mathscr{O}\mathrm{ffset}\left(\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}(\mathtt{low})\right) \\
\mathscr{S}\mathrm{lice}^{[16:31]}_{[0:15]}(\mathtt{high}) = \mathtt{t} + \mathscr{O}\mathrm{ffset}\left(\mathscr{S}\mathrm{lice}^{[16:31]}_{[0:15]}(\mathtt{high})\right)
\end{array}\right\}\ \mathfrak{Offset}$$

$$\left.\begin{array}{r}
\mathscr{O}\mathrm{ffset}\left(\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}(\mathtt{low})\right) = \mathscr{O}\mathrm{ffset}\,(\mathtt{x}) \\
\mathscr{O}\mathrm{ffset}\left(\mathscr{S}\mathrm{lice}^{[16:31]}_{[0:15]}(\mathtt{high})\right) = \mathscr{O}\mathrm{ffset}\,(\mathtt{x})
\end{array}\right\}\ \mathfrak{Equality}$$

Let us explain it in natural language. The $\mathfrak{Slice}$ domain created two ghost variables: $\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}(\mathtt{low})$ and $\mathscr{S}\mathrm{lice}^{[16:31]}_{[0:15]}(\mathtt{high})$. Moreover, it remembers that bits 0 to 15 of `low` and $\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}(\mathtt{low})$ are equal and that bits 16 to 31 of `low` are 0. Likewise, bits 0 to 15 of `high` are equal to bits 16 to 31 of $\mathscr{S}\mathrm{lice}^{[16:31]}_{[0:15]}(\mathtt{high})$ and bits 16 to 31 of `high` 0.

Domain $\mathfrak{Offset}$ knows that `x`, $\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}(\mathtt{low})$ and $\mathscr{S}\mathrm{lice}^{[16:31]}_{[0:15]}(\mathtt{high})$ are pointers. They all point to the same block (array `t`), and each one has its own offset.

Domain $\mathfrak{Equality}$ knows that offset of `x` $\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}(\mathtt{low})$ and $\mathscr{S}\mathrm{lice}^{[16:31]}_{[0:15]}(\mathtt{high})$ are equal. Since they point to the same block, this also means that these pointers are equal. From this, we can deduce that bits 0 to 15 of `low` are equal to bits 0 to 15 of `x`, which is what we expect from the source code. But for the sake of the example, we will assume that this reduction did not occur, i.e. we only know that offsets are equal, not whole pointers.

At this point, we have the following support, made of 3 trees, one for each real variable:

$$\begin{tikzcd}
\mathtt{x} \arrow[d, dash] & \mathtt{low} \arrow[d, dash] & \mathtt{high} \arrow[d, dash] & \left.\right\}\ \mathscr{V}_0 = \mathbb{V} \\
\mathscr{O}\mathrm{ffset}\,(\mathtt{x}) & \mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}(\mathtt{low}) \arrow[d, dash] & \mathscr{S}\mathrm{lice}^{[16:31]}_{[0:15]}(\mathtt{high}) \arrow[d, dash] & \left.\right\}\ \mathscr{V}_1 \\
 & \mathscr{O}\mathrm{ffset}\left(\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}(\mathtt{low})\right) & \mathscr{O}\mathrm{ffset}\left(\mathscr{S}\mathrm{lice}^{[16:31]}_{[0:15]}(\mathtt{high})\right) & \left.\right\}\ \mathscr{V}_2
\end{tikzcd}$$

There are three real variables and five ghost variables: three of them belong to $\mathfrak{Offset}$ and two belong to $\mathfrak{Slice}$. We can remark that a previously mentioned assumptions is satisfied: when a variable exists, all less ghostly variables are alive, too.

Now, let us run the last statement: `int* y = low | (high << 16);`. $\mathfrak{Offset}$ and $\mathfrak{Equality}$ domains get nothing interesting from that. However, the $\mathfrak{Slice}$ domain is more clever with bitwise operations. It can deduce that bits 0 to 15 of $\mathtt{y}$ are bits 0 to 15 of low, which are also bits 0 to 15 of $\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{low})$. Thus, it can just copy $\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{low})$ into $\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{y})$. Similarly, bits 16 to 31 of $\mathtt{y}$ are bits 0 to 15 of $\mathtt{high}$, which are bits 16 to 31 of $\mathscr{S}\text{lice}_{[0:15]}^{[16:31]}(\mathtt{high})$. Thus, it can copy $\mathscr{S}\text{lice}_{[0:15]}^{[16:31]}(\mathtt{high})$ into $\mathscr{S}\text{lice}_{[16:31]}^{[16:31]}(\mathtt{y})$ and remember that bits 16 to 31 of both variables are equal. We can write

$$\begin{array}{|c|c|l}
\multicolumn{1}{l}{\scriptstyle 31 \hfill \scriptstyle 16} & \multicolumn{1}{l}{\scriptstyle 15 \hfill \scriptstyle 0} & \\
\hline
\mathscr{S}\text{lice}_{[16:31]}^{[16:31]}(\mathtt{y})\,[16:31] & \mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{y})\,[0:15] & = \mathtt{y} \\
\hline
\end{array}$$

Moreover, the domain needs two new ghost variables: $\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{y})$ and $\mathscr{S}\text{lice}_{[16:31]}^{[16:31]}(\mathtt{y})$. It also emits two constraints:

$$\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{y}) \leftarrow \mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{low})$$
$$\mathscr{S}\text{lice}_{[16:31]}^{[16:31]}(\mathtt{y}) \leftarrow \mathscr{S}\text{lice}_{[0:15]}^{[16:31]}(\mathtt{high})$$

We can use directed constraints here, since newly created variables get $\top$ value. Directives of variable allocation and directed reductions are communicated to other domains.

Since $\mathfrak{Offset}$ has some information on $\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{low})$ and $\mathscr{S}\text{lice}_{[0:15]}^{[16:31]}(\mathtt{high})$, it can deduce that they are respectively equal to $\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{y})$ and $\mathscr{S}\text{lice}_{[16:31]}^{[0:15]}(\mathtt{y})$. This is written

$$\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{y}) = \mathtt{t} + \mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{y})\right)$$
$$\mathscr{S}\text{lice}_{[16:31]}^{[16:31]}(\mathtt{y}) = \mathtt{t} + \mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}_{[16:31]}^{[16:31]}(\mathtt{y})\right)$$

We can see that the domain requires two new variables: $\mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{y})\right)$ and $\mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}_{[16:31]}^{[16:31]}(\mathtt{y})\right)$. They are reduced using the directed reduction

$$\mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{y})\right) \leftarrow \mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{low})\right)$$
$$\mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}_{[16:31]}^{[16:31]}(\mathtt{y})\right) \leftarrow \mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}_{[0:15]}^{[16:31]}(\mathtt{high})\right)$$

Such constraints are not interesting for $\mathfrak{Slice}$. But $\mathfrak{Equality}$ domain already knows that

$$\mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{low})\right) = \mathscr{O}\text{ffset}(\mathtt{x})$$
$$\mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}_{[0:15]}^{[16:31]}(\mathtt{high})\right) = \mathscr{O}\text{ffset}(\mathtt{x})$$

and deduces from previous constraints that

$$\mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}(\mathtt{y})\right) = \mathscr{O}\text{ffset}(\mathtt{x})$$
$$\mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}_{[31:16]}^{[31:16]}(\mathtt{y})\right) = \mathscr{O}\text{ffset}(\mathtt{x})$$

This domain has no ghost variables and does not emit any reduction, the computation stops here.

To run this statement, domains have created this new tree of variables:

$$\begin{tikzcd}
& \mathtt{y} \arrow[dl, dash] \arrow[dr, dash] & & \mathscr{V}_0 = \mathbb{V} \\
\mathscr{S}\mathrm{lice}_{[0:15]}^{[0:15]}(\mathtt{y}) \arrow[d, dash] & & \mathscr{S}\mathrm{lice}_{[16:31]}^{[16:31]}(\mathtt{y}) \arrow[d, dash] & \mathscr{V}_1 \\
\mathscr{O}\mathrm{ffset}\left(\mathscr{S}\mathrm{lice}_{[0:15]}^{[0:15]}(\mathtt{y})\right) & & \mathscr{O}\mathrm{ffset}\left(\mathscr{S}\mathrm{lice}_{[16:31]}^{[16:31]}(\mathtt{y})\right) & \mathscr{V}_2
\end{tikzcd}$$

This state is correct but not interesting. We now enter the second phase, which is regular reduction just as in [Cou+06b]. Thanks to $\mathfrak{Equality}$ domain, $\mathfrak{Offset}$ domain can infer that

$$\mathscr{S}\mathrm{lice}_{[0:15]}^{[0:15]}(\mathtt{y}) = \mathtt{x}$$
$$\mathscr{S}\mathrm{lice}_{[16:31]}^{[16:31]}(\mathtt{y}) = \mathtt{x}$$

and communicates it to other domains. That is bits 0 to 15 of `y` are bits 0 to 15 of `x`, and bits 16 to 31 of `y` are bits 16 to 31 of `y`. From these equalities, $\mathfrak{Slice}$ domain can deduce that $\mathtt{x} = \mathtt{y}$, which is the expected result.

This illustrates why we need distributed handling of ghost variables and that why they should be known by other domains. In fact, when we use a ghost variable, it is likely that another domain can do better about it, otherwise we would not need a ghost variable. For instance, parts of `low`, `high` and `y` are pointers: it is more suitable to ask the $\mathfrak{Offset}$ domain to represent them. Likewise, the offset is a numerical value so, while the $\mathfrak{Offset}$ domain associates it to the pointer of which it is the offset, it is worth to let a numerical domain represents the offset value. But, if we know for offset variables, in general we do not *a priori* know the kind of value the ghost variable stores, for instance, in the case of bitwise slices that can be pointers, numerical values or anything else. Furthermore, we do not know in advance the list of available domains, so we cannot decide which domain is the best to represent a ghost variable. Thus, every ghost variable must be known by all domains.

Let us discuss the problems that we have highlighted. The termination is satisfied: directed reductions of offsets do not trigger new constraints. We can observe that each constraint is more elementary than the previous, and there is nothing simpler than copying a numerical variable. The decreasing complexity is a good approach to ensures termination. The other problem was the execution order. Here, we can see that the assignments (from directed constraints) are disjoint: we assign variables under `y` from variables under `low` and `high`. So, an assigned variable is never in the right-hand side of an assignment and since we always go deeper in ghost variables, each variable is assigned only once, making directed reductions legal. Both these points will be detailed and generalized later.

20.1.2 Generic Abstract Domain Signature

We can be very general on the form of the abstract domains. Classically, a domain needs a set of abstract states and their meaning in the concrete world, fixpoint approximation and abstract counterparts of concrete primitives. In addition, we add two other kinds of function.

First, there are two maps to decide what to do with ghost variables: one to react to the execution of a unary statement or a ghost constraint, and one to unify supports before performing a binary operation (typically, the join).

Moreover, domains include primitives to communicate information about their abstract state *à la* [Cou+06b]. This allows domains to refine themselves (as a reduced product is meant to) but also to communicate their policy on ghost variables management: since all domains must care about the ghost variables of the other domains, they need to communicate what to do.

We are given a lattice $IO^\sharp$ with a concretization $\gamma_{IO} : IO^\sharp \to \mathbb{D}$. This is a lattice common to all domains that will be the middleman for all communication.

Definition 20.3 – Generic abstract domain

A generic abstract domain with dynamic support is a tuple

$$\left(n, \mathbb{D}^\sharp, \gamma, \sqcup, \mathrm{lfp}^\sharp, \textsc{Assign}, \textsc{Guard}, \textsc{Alloc}, \textsc{Kill}, \textsc{Extract}, \textsc{Refine}, \lessdot, \mathcal{U}, \mathcal{B}\right)$$

where:

- $n \in \mathfrak{N}$ is the domain name,
- $\mathbb{D}^\sharp$ is the set of abstract properties,
- $\gamma : \mathbb{D}^\sharp \to \mathbb{D}$ is the concretization,
- $\sqcup : \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$
- $\mathrm{lfp}^\sharp : (\mathbb{D}^\sharp \to \mathbb{D}^\sharp) \to (\mathbb{D}^\sharp \to \mathbb{D}^\sharp)$
- $\textsc{Assign} : \mathscr{A} \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$
- $\textsc{Guard} : \mathscr{C} \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$
- $\textsc{Alloc} : \mathscr{V} \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$
- $\textsc{Kill} : \mathscr{V} \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$
- $\textsc{Extract} : IO^\sharp \times \mathbb{D}^\sharp \to IO^\sharp$
- $\textsc{Refine} : IO^\sharp \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$
- $\lessdot \subseteq \mathscr{U} \times \mathscr{U}$
- $\mathcal{U} : \mathscr{U} \times \mathcal{P}(\mathscr{U}) \times \mathbb{D}^\sharp \to \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathscr{G} \times \mathcal{P}(\mathscr{U})$
- $\mathcal{B} : \mathbb{N}^* \times \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to (\mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathscr{G})^2$

Assumptions that makes the domain correct and terminating are quite long and cannot be decently given in a single definition. They will be exposed in the following of this subsection.

To avoid redundant redundancy, all the following should be understood in the context of a domain

$$\left(n, \mathbb{D}^\sharp, \gamma, \sqcup, \mathrm{lfp}^\sharp, \textsc{Assign}, \textsc{Guard}, \textsc{Alloc}, \textsc{Kill}, \textsc{Extract}, \textsc{Refine}, \lessdot, \mathcal{U}, \mathcal{B}\right)$$

Assumption 20.1 – Soundness of $\sqcup$

$$\forall (a,b) \in {\mathbb{D}^\sharp}^2, \gamma(a) \cup \gamma(b) \subseteq \gamma(a \sqcup b)$$

This means $\sqcup$ is a sound approximation of $\cup$ this is useful, for instance, at the end of if-then-else structures.

Assumption 20.2 – Soundness of $\mathrm{lfp}^\sharp$

$$\begin{aligned}&\forall f : \mathbb{D} \to \mathbb{D}, \forall c \in \mathbb{D}, \forall f^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp, \forall a \in \mathbb{D}^\sharp,\\ &\qquad (f \circ \gamma \subseteq \gamma \circ f^\sharp \wedge c \subseteq \gamma(a)) \Rightarrow \mathrm{lfp}_c f \subseteq \gamma({\mathrm{lfp}^\sharp}_a f^\sharp)\end{aligned}$$

Usually, to find a sound approximation of the concrete least fixpoint, we use a widening operator. Though, there are many strategies to increase precision, quite general ones (delayed widening, narrowing) or very ad hoc heuristics. The way by which we find this fixpoint approximation is not important here. By using a $\mathrm{lfp}^\sharp$ operator, we are the most general. Each implementation of a generic domain can use its own strategy.

Assumption 20.3 – Soundness of Assign

$$\begin{aligned}\forall (v,e) \in \mathscr{A}, \forall a \in \mathbb{D}^\sharp, \quad & \mathit{Assign}\,((v,e), \gamma(a)) \subseteq \gamma(\textsc{Assign}\,((v,e),a))\\ \wedge \quad & \textbf{let } d := \gamma(\textsc{Assign}\,((v,e),a)) \textbf{ in}\\ & \forall (w,s,x) \in v^{\downarrow\lhd} \times d \times \mathbb{I}, s\,[w \mapsto x] \in d\end{aligned}$$

There are two parts here: when there is an assignment, the assigned variable is indeed modified accordingly. But old values of ghostlier variables may not be correct anymore. To be sound, Assign should set all ghostlier variables than the assigned one to $\top$.

Another point of view on assignment (v, e) is to decompose it into two successive computations. The first sets all variables of $v^{\downarrow\lhd}$ to $\top$. The second is to apply the directed constraint $v \leftarrow e$, that refines the value of variable v.

Following assumptions are simply the soundness of various components of an abstract domain. There are no subtleties here.

Assumption 20.4 – Soundness of Guard

$$\forall c \in \mathscr{C}, \forall a \in \mathbb{D}^\sharp, \mathit{Guard}\,(c, \gamma(a)) \subseteq \gamma(\textsc{Guard}\,(c,a))$$

Assumption 20.5 – Soundness of Alloc

$$\forall v \in \mathscr{V}, \forall a \in \mathbb{D}^\sharp, \mathit{Alloc}\,(v, \gamma(a)) \subseteq \gamma(\textsc{Alloc}\,(v,a))$$

Assumption 20.6 – Soundness of Kill

$$\forall v \in \mathscr{V}, \forall a \in \mathbb{D}^\sharp, \mathit{Kill}\left(v, \gamma(a)\right) \subseteq \gamma(\textsc{Kill}\left(v, a\right))$$

In particular, Alloc and Kill have the same effect on the support as their concrete counterparts.

Assumption 20.7 – Soundness of Extract

$$\forall io \in IO^\sharp, \forall a \in \mathbb{D}^\sharp, \gamma(a) \cap \gamma_{IO}\left(io\right) \subseteq \gamma_{IO}\left(\textsc{Extract}\left(io, a\right)\right)$$

Assumption 20.8 – Soundness of Refine

$$\forall io \in IO^\sharp, \forall a \in \mathbb{D}^\sharp, \gamma(a) \cap \gamma_{IO}\left(io\right) \subseteq \gamma\left(\textsc{Refine}\left(io, a\right)\right)$$

These functions are used to perform communications across domains, as done in [Cou+06b].

The next assumption is about $\mathcal{U}$. This function is quite special, as it is the core of ghost variable handling: given a ghost constraint, it yields sets of variables to allocate and kill, and it generates new constraints (as a constraint-DAG). We must ensure that this graph is sound. Moreover, since each constraint generates new constraints, we need assumptions to enforce termination. Lastly, there is the problem of directed reductions: one can emit a directed reduction $x \leftarrow e$ only if x is $\top$. This is guaranteed by some requirements on $\mathcal{U}$.

Before giving this big list of hypotheses, let us define a relation of $\mathcal{U}$. If $u \succ_n v$, then the domain of name $n \in \mathfrak{N}$ is allowed to produce constraint v given the constraint u. This relation is useful for soundness, but it is not enough to ensure termination.

Definition 20.4

Let $n \in \mathfrak{N}$. Let $\succ_n$ be the smallest relation on $\mathscr{U}$ satisfying:

– $\forall v \in \mathscr{V}, \forall \mathscr{R} \in \mathcal{R}, \forall (e, e') \in \mathscr{E}^2$,

$$\mathscr{R} \Subset n \wedge e' \trianglelefteq e \Rightarrow (v, e) \succ_n (\mathscr{R}(v), e')$$

– $\forall \left(P^{/p}, Q^{/q}\right) \in \mathbb{O}^2, \forall (a_i)_{i \in [\![1,p]\!]} \in \mathscr{E}^p, \forall (b_i)_{i \in [\![1,q]\!]} \in \mathscr{E}^q$,

$$\left(\begin{array}{c} p \geqslant q \wedge \\ \forall i \in [\![1, q]\!], \exists j \in [\![1, p]\!], b_i \trianglelefteq a_j \end{array}\right) \Rightarrow P^{/p}\left(a_1, \ldots, a_p\right) \succ_n Q^{/q}\left(b_1, \ldots, b_q\right)$$

– $\forall P^{/p} \in \mathbb{O}, \forall e \in \mathscr{E}, \forall (a_i)_{i \in [\![1,p]\!]} \in \mathscr{E}^p, \forall v \in \mathscr{V}$,

$$(\forall i \in [\![1, p]\!], a_i \trianglelefteq e) \Rightarrow (v, e) \succ_n P^{/p}\left(a_1, \ldots, a_p\right)$$

This definition is quite austere, let us rewrite the three cases:

– given a directed constraint, one can generate a directed constraint on an immediately ghostlier variable, using a ghostlier expression;
– given a condition, one can generate another condition using only ghostlier expressions;
– given a directed constraint, one can generate a condition using expressions that are ghostlier than the right-hand side of the directed constraint.

All these cases induce increasing ghostliness. From a directed constraint $x \leftarrow e$, we can generate directed constraints $y \leftarrow f$ where y is ghostlier than x and f ghostlier than e, and thanks to assumption 19.1 "Restriction on assignments" (page 244) no left-hand side of such directed constraint can appear in right-hand sides. This is a crucial argument in following proofs.

Notation 20.1 – Support of an abstract value

Let $a \in \mathbb{D}^\sharp$. We write

$$\operatorname{supp}(a) := \operatorname{supp}(\gamma(a))$$

Assumption 20.9

Relation $\lessdot$ is a well-founded order on $\mathscr{U}$.

Assumption 20.10 – Hypotheses on constraint triggering

$\forall a \in \mathbb{D}^\sharp, \forall u \in \mathscr{U}, \forall U \in \mathcal{P}(\mathscr{U})$,
let $(\mathit{new}, \mathit{old}, g, L) := \mathcal{U}(u, U, a)$ **in**
let $(N, E, f) := g$ **in**

a) $\mathit{new} \cap \operatorname{supp}(a) = \varnothing$
b) $\mathit{old} \subseteq \operatorname{supp}(a)$
c) $\forall c \in g, c \in \mathscr{A} \Rightarrow \pi_1(c) \in \operatorname{supp}(a) \cup \mathit{new} \wedge \operatorname{Var}(\pi_2(c)) \subseteq \operatorname{supp}(a)$
d) $\forall c \in g, c \in \mathscr{C} \Rightarrow \operatorname{Var}(c) \in \operatorname{supp}(a) \cup \mathit{new}$
e) g is sound,
f) $\forall v \in g, u \succ_n v$
g) $\forall (i,j) \in N^2, f(i) \in \mathscr{A} \wedge f(j) \in \mathscr{A} \wedge \pi_1(f(i)) = \pi_1(f(j)) \Rightarrow i = j$
h) $\forall v \in g, v \in U$
i) $L = U \cap u^{\downarrow\lessdot}$

This is quite a long assumption. Items a and b mean that we allocate variables that were not alive and kill only living variables. Items c and d are sanity hypotheses: constraints only involve living or new variables.

Following items are less obvious and more interesting. Item e ensures that we will only refine the abstract state with respect to ghost the semantics of ghost variables. Items f and g are useful to prove that we use correctly directed constraints: the former means that a domain trigger only ghostlier constraints, and the latter means that in a graph, there cannot be two directed constraints about the same variable. Items h and i are useful to ensure termination. Termination criterion may seem more flexible, but in practice, it is not that much. This is in fact the least modular point. This will be explained in section 20.4 "Termination" (page 266).

The last component of generic abstract domains is the function

$$\mathcal{B} : \mathbb{N} \times \mathcal{P}(\mathscr{U}) \times \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to (\mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathscr{G})^2 \times \mathcal{P}(\mathscr{U})$$

Its purpose is to unify the support of two abstract states, in order to perform a binary operation.

Here, we do not want to refine the state, thus we can use simpler assumption: we only need the result to be bigger. To be able to compute the result, we introduce the abstract semantics of constraint-graphs and graphs with set of variables to add and kill.

Definition 20.5 – Abstract semantics of a constraint-DAG

Let $g \in \mathscr{G}$ be a constraint graph. We define $\langle\!\langle g \rangle\!\rangle^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ the abstract semantics of the graph g.

Definition 20.6

Let $(new, old, g) \in \mathscr{V} \times \mathscr{V} \times \mathscr{G}$. We define $\llparenthesis (new, old, g) \rrparenthesis^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ by

$$\llparenthesis (new, old, g) \rrparenthesis^\sharp = \mathop{\bigcirc}\limits_{v \in old} \textsc{Kill}\,(v, \_) \circ \langle\!\langle g \rangle\!\rangle^\sharp \circ \mathop{\bigcirc}\limits_{v \in new} \textsc{Alloc}\,(v, \_)$$

where _ is a placeholder (see section A.5 "Function Theory" (page 374)).

A 3-tuple made of sets of variables to allocate and kill and a constraint-graph is called an enriched graph.

Assumption 20.11 – Assumptions on support unification function

$\forall (a, b) \in \mathbb{D}^{\sharp 2}, \forall i \in \mathbb{N}^*,$
let $(G_a, G_b) := \mathcal{B}(i, a, b)$ **in**
let $a' := \llparenthesis G_a \rrparenthesis^\sharp (a)$ **in**
let $b' := \llparenthesis G_b \rrparenthesis^\sharp (b)$ **in**
let $(new_a, old_a, g_a) := G_a$ **in**
let $(new_b, old_b, g_b) := G_b$ **in**
let $\mathfrak{V} := \{\mathscr{R}(v) \in \mathscr{V} \setminus \mathbb{V} \mid v \in \mathscr{V}, \mathscr{R} \in \mathcal{R}, \mathscr{R} \Subset n\}$ **in**

a) $new_a \cap \operatorname{supp}(a) = \varnothing$
b) $old_a \subseteq \operatorname{supp}(a)$
c) $new_b \cap \operatorname{supp}(b) = \varnothing$
d) $old_b \subseteq \operatorname{supp}(b)$
e) $\gamma(a) \subseteq \gamma(a')$
f) $\gamma(b) \subseteq \gamma(b')$
g) $(\forall j \in [\![0, i-1]\!], \operatorname{supp}(a) \cap \mathscr{V}_j \cap \mathfrak{V} = \operatorname{supp}(b) \cap \mathscr{V}_j \cap \mathfrak{V}) \Rightarrow$
$(\forall j \in [\![0, i-1]\!], \operatorname{supp}(a) \cap \mathscr{V}_j \cap \mathfrak{V} = \operatorname{supp}(a') \cap \mathscr{V}_j \cap \mathfrak{V} = \operatorname{supp}(b') \cap \mathscr{V}_j \cap \mathfrak{V} \wedge$

$$\text{supp}\,(a') \cap \mathscr{V}_i \cap \mathfrak{V} = \text{supp}\,(b') \cap \mathscr{V}_i) \cap \mathfrak{V}$$

Here, G_a is an enriched graph to apply on a, and G_b is to apply on b. The integer i stands for the level of ghost variables we want to unify. By performing successive calls to $\mathcal{B}$ with increasing i, we unify the whole support.

Items a to d are similar to the first two of assumption 20.10. Items e and f ensure that this unification process is sound by being an overapproximation of the identity. Item g is the interesting point: if the support of a and b, restricted to $\mathscr{V}_0 \cup \cdots \cup \mathscr{V}_{i-1}$ and to variables belonging to the domain of name n, are equal, then $\mathcal{B}$ provides operations that does not change this part of the support, but also unify the i^{th} level of ghost variables that belongs to the domain. Using the $\mathcal{B}$ function of each domain, we unify the whole i^{th} level of ghost variables.

20.2 Product of Domains

Now we have seen the whole signature of a generic domain, with assumptions, let us explain how to use all these functions to build a composite domain that handles shared ghost variables. In this section, we are only interested in the working, how to connect everything. Correctness and termination will be explained later. We will also see later how to handle binary operations.

Given a constraint, function $\mathcal{U}$ provides new variables to allocate, variables to kill and more constraints about ghost variables, according to their semantics. The fourth returned value is a set of constraints that can be triggered by recursive calls to $\mathcal{U}$. This is useful to ensure termination.

To get the better precision, we shall call $\mathcal{U}$ recursively to get even more constraints. We recall that $\mathcal{U}$ has type $\mathscr{U} \times \mathcal{P}(\mathscr{U}) \times \mathbb{D}^\sharp \to \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathscr{G} \times \mathcal{P}(\mathscr{U})$. Let $a \in \mathbb{D}^\sharp$ be the abstract state we want to refine. Let $N \subseteq \mathfrak{N}$ be the set of the names of abstract domains in the product. We let $(\mathcal{U}^n)_{n\in N}$ the "$\mathcal{U}$ map" of each domain. Function

$$\begin{aligned} \mathcal{U}^n_a : \mathscr{U} \times \mathcal{P}(\mathscr{U}) &\to \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathscr{G} \times \mathcal{P}(\mathscr{U}) \\ (u, U) &\mapsto \mathcal{U}^n(u, U, a) \end{aligned}$$

where $n \in N$, is the partial application of $\mathcal{U}^n$ where the third argument is a.

We want to combine all functions $\mathcal{U}^n_a$ in a single function $\mathcal{U}_a$ that collects constraints from all domains, and, from that, we build a function $\mathcal{U}^*_a$ that performs this operation recursively. The first step into this direction is to share all $\mathcal{U}^n_a$ functions between all domains. This can be achieved through Extract and Refine. Given a communication domain IO and an abstract domain $\mathbb{D}^\sharp$, we can enrich them into

$$\begin{aligned} IO' &:= IO \times \mathcal{P}\left(\mathscr{U} \times \mathcal{P}(\mathscr{U}) \times \mathbb{D}^\sharp \to \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathscr{G} \times \mathcal{P}(\mathscr{U})\right) \\ \mathbb{D}^{\sharp\prime} &:= \mathbb{D}^\sharp \times \mathcal{P}\left(\mathscr{U} \times \mathcal{P}(\mathscr{U}) \times \mathbb{D}^\sharp \to \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathscr{G} \times \mathcal{P}(\mathscr{U})\right) \end{aligned}$$

and update Extract and Refine accordingly:

$$\begin{aligned}\textsc{Extract}'_n &: IO' \times \mathbb{D}^\sharp \to IO'\\ &((io, F), (a, F')) \mapsto (\textsc{Extract}(io, a), F \cup \{\mathcal{U}^n_a\})\end{aligned}$$

and

$$\begin{aligned}\textsc{Refine}'_n &: IO' \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp\\ &((io, F), (a, F')) \mapsto (\textsc{Refine}(io, a), F \cup F')\end{aligned}$$

Once each domain gets all $\mathcal{U}^n_a$ function, we combine them using

$$\begin{aligned}C &: (\mathscr{U} \times \mathcal{P}(\mathscr{U}) \to \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathscr{G} \times \mathcal{P}(\mathscr{U}))^N\\ &\quad\to (\mathscr{U} \times \mathcal{P}(\mathscr{U}) \to \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{G}) \times \mathcal{P}(\mathscr{U}))\\ &(\mathcal{U}^n_a)_{n\in N} \mapsto \left((u, U) \mapsto \left(\begin{array}{l}\bigcup_{n\in N} \pi_1(\mathcal{U}^n_a(u, U)),\\ \qquad\bigcup_{n\in N} \pi_2(\mathcal{U}^n_a(u, U)),\\ \qquad\qquad\{\pi_3(\mathcal{U}^n_a(u, U)) \mid n \in N\},\\ \qquad\qquad\qquad\bigcap_{n\in N} \pi_4(\mathcal{U}^n_a(u, U))\end{array}\right)\right)\end{aligned}$$

and we denote

$$\mathcal{U}_a := C\left((\mathcal{U}^n_a)_{n\in N}\right)$$

Now, we can iterate the collection of constraints. Given $\mathcal{U}_a : \mathscr{U} \times \mathcal{P}(\mathscr{U}) \to \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{G}) \times \mathcal{P}(\mathscr{U})$, we define

$$\begin{aligned}\mathcal{U}^*_a &: \mathscr{U} \to \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{G}) \times \mathcal{P}(\mathscr{U})\\ &u \mapsto \left(\begin{array}{l}\textbf{let } g := \mathcal{U}_a(u, \mathscr{U}) \textbf{ in}\\ \mathrm{lfp}_g\left((N, O, G, U) \mapsto \begin{array}{l}\textbf{let } R := \left\{\mathcal{U}_a(u', U) \mid \exists g' \in \mathscr{G} : u' \in g' \in G\right\} \textbf{ in}\\ \left(\begin{array}{l}N \cup \{\pi_1(r) \mid r \in R\},\\ \quad O \cup \{\pi_2(r) \mid r \in R\},\\ \qquad G \cup \{\pi_3(r) \mid r \in R\},\\ \qquad\quad \pi_4(R)\end{array}\right)\end{array}\right)\end{array}\right)\end{aligned}$$

where the order of the least fixpoint is the pointwise order induced by $\subseteq$ for the first three components and $\supseteq$ for the fourth. Function $\mathcal{U}^*_a$ is a vague analog of a transitive closure, hence the notation. Thanks to it, we can collect all constraints and reduce as much as possible.

To execute a unary operation (an assignment or a guard), we start by running it using the appropriate abstract transfer function. Then we need to build $\mathcal{U}^*_a$: domains communicate their $\mathcal{U}^n_a$, compute $\mathcal{U}_a$ and iterate. In the case of a guard, the condition is the initial constraint given to $\mathcal{U}_a^*$. In the case of an assignment $x := e$, the initial constraint is $x \leftarrow e$. Indeed, we recall that the assignment $x := e$ is equivalent to make all variables of $x^{\downarrow\triangleleft}$ unknown, and enforcing the constraint $x \leftarrow e$, which is legal since x is $\top$ (since $x \in x^{\downarrow\triangleleft}$).

20.3 Soundness

Soundness is guaranteed because our method relies on reduction: since all reduction steps are individually sound, every combination of them is sound as well. The main problem comes from directed assignments.

We have introduced two kinds of constraints: general guards, and the special case of directed constraints. We recall that the directly constraint $x \leftarrow e$ stands for the constraint $x = e$, but it can only be used when $x = \top$, so that reduction only happens in one direction, and thus can be run as an assignment, which is much more efficient than an arbitrary comparison. We shall make sure that no reduction involving x happens from the point where x is made unknown to the moment we run the directed constraint.

In the example detailed in subsection 20.1.1, we see that directed constraints are correctly used by increasing the ghostliness of involved variables. This is the point of item f of assumption 20.10 "Hypotheses on constraint triggering" (page 261).

Theorem 20.1

Let $u \in \mathscr{U}$ and $G := \pi_3\left(\mathcal{U}_a^*(u)\right) \in \mathcal{P}\left(\mathscr{G}\right)$. We have

a) if $u \in \mathscr{C}$, $\forall g \in G, \forall u' \in g, u' \in \mathscr{C}$

b) if $u \in \mathscr{A}$, $\forall (g_1, g_2) \in G^2$,
let $(N_1, E_1, f_1) := g_1$ **in let** $(N_2, E_2, f_2) := g_2$ **in** $\forall (n_1, n_2) \in N_1 \times N_2$,

$$\left(f_1(n_1) \in \mathscr{A} \wedge f_2(n_2) \in \mathscr{A} \wedge \pi_1\left(f_1(n_1)\right) = \pi_1\left(f_2(n_2)\right)\right) \Rightarrow \left(n_1 = n_2 \wedge (N_1, E_1, f_1) = (N_2, E_2, f_2) \wedge \pi_1\left(f_1(u')\right) \in \pi_1\left(u\right)^{\downarrow\triangleleft}\right)$$

Let us reword this theorem, before justifying it. Let u be a constraint. We name G, the set of constraint graph generated from constraint u. If u is a simple constraint, all constraints generated are simple constraints as well. If u is a directed constraint $x \leftarrow e$, then all directed constraints in G are about variables ghostlier than x. Moreover, a ghost variable can appear at most once as the left-hand side of a directed constraint.

There are some hidden hypotheses, that are parts of the assumptions on generic abstract domains. Relevant ones are:

- each domain allocates and kills only variables that it owns;
- triggering of constraints satisfies $\succ_n$ (increasing ghostliness);
- in a constraint-DAG, a ghost variable can only appear once in the left-hand side of a directed constraint.

Let us draw a proof sketch. We made the assumption that real assignments never use their left-hand side in their right-hand side. Thus, the tree of variables that are ghostlier than the left-hand side and the forest of variables ghostlier than variables in the right-hand side are disjoint. This has a crucial consequence: all left-hand sides of assignments may never appear in the right-hand side of a directed constraint or in a comparison. Indeed, in the example, the last directed constraint assigns variables under `y` using variables under `low` and `high`.

We now have to check that an assigned variable can only be written once. Let $v \in \mathscr{V}$. We distinguish two cases:

- $v \in \mathbb{V}$. A directed constraint can only be about a ghost variable (thank to $\succ_n$). So v is only constrained by the real assignment. So, only once.
- $v \notin \mathbb{V}$. There is a variable v', a role $\mathscr{R}$ and an integer i such that $v = \mathscr{R}(v')$ and $v \in \mathscr{V}_i$. Since each directed constraint triggered by another directed constraint writes in a variable exactly one level more ghostly, a directed constraint on a variable in $\mathscr{V}_i$ can only be generated at the i^{th} step of recursion. Moreover, it can only come from the domain that owns the role $\mathscr{R}$. Consequently, there is only one constraint-DAG that may contain a directed constraint on v. As we assumed that a variable may only appear in one directed constraint in a constraint-DAG, there is only one directed constraint to v.

So, variables under the left-hand side of a directed constraint appear at most once and only as the left-hand side of a directed constraint, and these variables were previously set to $\top$ by Assign. Hence, the hypothesis on directed constraints holds.

20.4 Termination

A priori, $\mathcal{U}_a^*$ yields infinite sets, which is quite embarrassing to actually compute and to enforce these constraints. But thanks to items h and i of assumption 20.10 "Hypotheses on constraint triggering" (page 261), we can prove that these sets are in fact finite. Indeed, each domain has a well-founded order $\lessdot$. When called, $\mathcal{U}$ is given a subset of $\mathscr{U}$ (its second parameter), which is the set of constraints the domain is allowed to use. It returns a subset of these constraints, that are lower, according to order $\lessdot$. In the product, to compute $\mathcal{U}_a$ form individual $\mathcal{U}_a^n$, we need the intersection of these sets of allowed constraints. So that, at each iteration of $\mathcal{U}_a$, to compute $\mathcal{U}_a^*$, we allow only decreasing sequences of constraints, with respect to a well-founded order (defined as the intersection of well-founded orders).

Here, the hypothesis is local to each domain, making the termination proof distributed. But if the orders are poorly designed, satisfying all of them can be too constraining, and quickly, the set of allowed constraints can be empty, even before being able to perform interesting reductions. In this situation, variables that are ghostly enough will not be reduced. There is a trade-off: either, we prefer to keep termination proof distributed (even if it means that some variables will not be reduced), or we can use the same well-founded order for all domains. This second approach is more predictable but may imply to redesign the order when we add new domains to the product. A hybrid solution can be preferred: trying to stay as close as possible from a reference order, though domains can define their own order to take all their idiosyncrasies into account.

Let us define such an order that is good enough to run the example and that is very close to the order we use in the implementation, up to some technical details. It is a particular instance but that is good enough in practice and thus is worth to be explained in details. Let us examine how the example of subsection 20.1.1 "A Detailed Execution of an Example with Ghost Variables" (page 253) works. We see that ghostlier and ghostlier variables appear in the left-hand side of directed constraints with ghostlier and ghostlier expressions.

Let us take a look to the tree of ghost constraints. From `y = low | (high << 16)`, at the first step of recursion we got

- $\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\mathtt{y}) \leftarrow \mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\mathtt{low})$
- $\mathscr{S}\text{lice}^{[16:31]}_{[16:31]}(\mathtt{y}) \leftarrow \mathscr{S}\text{lice}^{[16:31]}_{[0:15]}(\mathtt{high})$

and at the second step

- $\mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\mathtt{y})\right) \leftarrow \mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\mathtt{low})\right)$
- $\mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}^{[16:31]}_{[16:31]}(\mathtt{y})\right) \leftarrow \mathscr{O}\text{ffset}\left(\mathscr{S}\text{lice}^{[16:31]}_{[0:15]}(\mathtt{high})\right)$

It is clear that the left-hand side of directed constraints decreases, as prescribed by the item f of assumption 20.10 "Hypotheses on constraint triggering" (page 261). Sadly, it is not enough to ensure termination: since we can add ghost variables at each call of $\mathcal{U}$, we may end up into adding an infinitely deep tree of ghost variables. It is also worth noticing that all new ghost variables are under the left-hand side of an assignment, which is pretty natural.

First let us define the complexity order on expressions.

Notation 20.2

For $e \in \mathscr{E}$, we denote $\text{MVar}(e)$ the multiset of variables in e, that is

$$\begin{aligned}\text{MVar} : \mathscr{E} &\to \mathcal{M}(V)\\ e &\mapsto \begin{cases}\sum_{i=1}^{n} \text{Var}(e_i) & \text{if } e = c^{/n}(e_1, \ldots, e_n)\\ \{\!|e|\!\} & \text{if } e \in \mathscr{V}\end{cases}\end{aligned}$$

As it is a multiset, it takes multiplicity into account. For instance, given a variable x, $\text{MVar}(x + x) = \{\!|x, x|\!\}$. See section A.4 "Multisets" (page 373) for basic notations and definitions about multisets, though we recall some definitions that are crucial and not so common.

Definition 20.7

Let E a set and $\sqsubseteq$ a partial order on E. We denote $\sqsubset_{\text{DM}}$, and called Dershowitz–Manna order (or multiset order) induced by $\sqsubseteq$, the relation on $\mathcal{M}(E)$

defined by

$$\forall(\mathcal{A},\mathcal{B}) \in \mathcal{M}(E)^2, \mathcal{A} \sqsubset_{\mathrm{DM}} \mathcal{B} :\Leftrightarrow \exists(\mathcal{X},\mathcal{Y}) \in \mathcal{M}(E)^2 : \begin{pmatrix} \quad \mathcal{X} \neq \varnothing \\ \wedge\ \mathcal{X} \subseteq \mathcal{B} \\ \wedge\ \mathcal{A} = (\mathcal{B} \setminus \mathcal{X}) + \mathcal{Y} \\ \wedge\ \forall y \in \mathcal{Y}, \exists x \in \mathcal{X} : y \sqsubset x \end{pmatrix}$$

We denote $\sqsubseteq_{\mathrm{DM}}$ the reflexive closure of $\sqsubset_{\mathrm{DM}}$.

In other words, to get a smaller multiset, we can remove an occurrence of an element x of B to add as many elements as we want that are lower than x. In [HO80], there is another, equivalent, definition which is also closer to the interpretation.

Proposition 20.1

Given a poset $(E, \sqsubseteq)$, $\sqsubseteq_{\mathrm{DM}}$ is a partial order on $\mathcal{M}(E)$.

Proposition 20.2

Given a poset $(E, \sqsubseteq)$, if $\sqsubseteq$ is well-founded, then $\sqsubseteq_{\mathrm{DM}}$ is a well-founded order on $\mathcal{M}(E)$.

These propositions are well-known properties of multiset ordering (see [DM79]).

Definition 20.8

Let us denote $\preccurlyeq$ the order relation on $\mathscr{E}$ defined by

$$a \preccurlyeq b :\Leftrightarrow \mathrm{MVar}(a) \trianglelefteq_{\mathrm{DM}} \mathrm{MVar}(b)$$

We denote $\prec$ the corresponding strict order.

This order compares only the multisets of variables in both operands. We do not care about the structure of expressions. For instance, we have

$$\mathtt{x\ +\ y} \prec \mathtt{x\ ==\ y\ ?\ sin(x)\ :\ ln(z)}$$

because $\mathrm{MVar}(\mathtt{x\ +\ y}) = \{\!\{\mathtt{x}, \mathtt{y}\}\!\}$, $\mathrm{MVar}(\mathtt{x\ ==\ y\ ?\ sin(x)\ ?\ ln(x+y)}) = \{\!\{\mathtt{x}, \mathtt{x}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}\!\}$ and $\{\!\{\mathtt{x}, \mathtt{y}\}\!\} \trianglelefteq_{\mathrm{DM}} \{\!\{\mathtt{x}, \mathtt{x}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}\!\}$.

But,

$$\mathtt{y\ +\ y} \not\preccurlyeq \mathtt{x\ ==\ y\ ?\ sin(x)\ :\ ln(z)}$$

because y appears twice on the left-hand side, but only once in the right-hand side, even if the left-hand side seems much simpler.

This order is useful to talk about ghostlier variables. For instance, in Listing 20.2, since pointers p and q alias each other, the computation q−p is translated into $\mathscr{O}\mathrm{ffset}(\mathtt{q})-$

$\mathscr{O}\text{ffset}\,(\texttt{p})$, and we indeed have $\{\!\!\{\mathscr{O}\text{ffset}\,(\texttt{q})\,,\mathscr{O}\text{ffset}\,(\texttt{q})\}\!\!\} \lhd_{\text{DM}} \{\!\!\{\texttt{p},\texttt{q}\}\!\!\}$ since $\mathscr{O}\text{ffset}\,(\texttt{p}) \lhd \texttt{p}$ and $\mathscr{O}\text{ffset}\,(\texttt{q}) \lhd \texttt{q}$

```c
int a[4];
int* p = &a[0];
int* q = &a[1];
int delta = q - p;
```

Listing 20.2: The difference of two pointers

But we are also interested into the structure of expression, hence the next order.

Definition 20.9

Let us denote $\preccurlyeq$ the order relation on $\mathscr{E}$ defined by

$$a \preccurlyeq b :\Leftrightarrow a \lessdot b \vee (a \lesseqdot b \wedge a \sqsupseteq b)$$

where $\sqsupseteq$ is the structural inclusion (see definition 19.1 "Structural inclusion" (page 239)).
We denote $\prec$ the corresponding strict order.

It is the lexicographic order induced by $\lesseqdot$ and $\sqsupseteq$. Unlike $\lesseqdot$, it allows keeping the same multi-set of ghost variables while simplifying the expression by cutting some branches. For instance, in Listing 20.3, we need a ghost $\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}\,(\texttt{low})$ variable whose 16 lower bits are the 16 lower bits of `low`. Thus, the assignment $\texttt{low} := \texttt{x \& FFFFH}$ is translated into the assignment $\mathscr{S}\text{lice}_{[0:15]}^{[0:15]}\,(\texttt{low}) := \texttt{x}$. Since the same multiset of ghost variables is involved, the first order is not permissive enough. But `x` is definitely simpler than `x &` FFFFH as it is structurally included. That what $\preccurlyeq$ takes into account.

```c
int x;
int low = x & 0xffff;
```

Listing 20.3: Extracting a slice

Definition 20.10 – Depth of a variable

Let $v \in \mathscr{V}$ be an abstract state. We call depth of v the integer i denoted $\text{depth}\,(v)$, and such that

$$v \in \mathscr{V}_i$$

This integer is obviously unique.

Likewise, a set of variable $V \subseteq \mathscr{V}$ is said to have finite depth if the set

$$\{\text{depth}\,(v) \mid v \in V\}$$

is bounded above.

Lemma 20.1

Let V be a subset of $\mathscr{V}$ of finite depth. The restriction to V of $\trianglelefteq$ is well-founded.

Proof. Given two variables v and w, if $v \trianglelefteq w$, then $\operatorname{depth}(v) > \operatorname{depth}(w)$. Thus, for any decreasing sequence of variables in V, the sequence of depth is increasing. Since $\geqslant$ is well-founded on set of the form $[\![0, n]\!]$ (with $n \in \mathbb{N}$), so is $\trianglelefteq$ on subsets of $\mathscr{V}$ of finite depth. □

Lemma 20.2

Let V be a subset of $\mathscr{V}$ of finite depth, and $E := \mathbb{E}_V$ (see notation 19.1 "Expressions" (page 239)). The restriction to E of $\preccurlyeq$ is well-founded.

Proof. Since $\trianglelefteq$ is well-founded on V, $\trianglelefteq_{\mathrm{DM}}$ is well-founded on $\mathcal{M}(V)$, hence the result. □

Lemma 20.3

Let V be a subset of $\mathscr{V}$ of finite depth, and $E := \mathbb{E}_V$. The restriction to E of $\precsim$ is well-founded.

Proof. It is well-founded as a lexicographic order of two well-founded orders. □

Definition 20.11 – A terminating relation

Let us denote $\twoheadrightarrow$ the smallest relation on $\mathscr{U}$ that satisfies

a) $\forall (v, e) \in \mathscr{A}, \forall C^{/n}(f_1, \ldots, f_n) \in \mathscr{C}$,

$$\bigcup_{i=1}^{n} \operatorname{Var}(f_i) \subseteq \operatorname{Var}(e)^{\downarrow \trianglelefteq} \Rightarrow (v, e) \twoheadrightarrow C^{/n}(f_1, \ldots, f_n)$$

b) $\forall (C^{/m}(e_1, \ldots, e_m), D^{/n}(f_1, \ldots, f_n)) \in \mathscr{C}^2$,

$$\begin{pmatrix} m \geqslant n \land \\ \exists \sigma \in \mathfrak{S}_m : \forall i \in [\![1, n]\!], f_i \prec e_{\sigma(i)} \end{pmatrix} \Rightarrow C^{/m}(e_1, \ldots, e_m) \twoheadrightarrow D^{/n}(f_1, \ldots, f_n)$$

where $\mathfrak{S}_n$ is the set of permutation of $[\![1, n]\!]$ (see section A.6 "Standard Objects" (page 376)),

c) $\forall ((v, e), (v', e')) \in \mathscr{A}^2$,

$$v' \triangleleft v \land \begin{Bmatrix} e' \prec e \text{ if } v \text{ is freshly allocated} \\ e' \precsim e \text{ otherwise} \end{Bmatrix} \Rightarrow (v, e) \twoheadrightarrow (v', e')$$

Let us reword that relation in English. Item *a* explains that we can generate a condition from a directed constraint as long as all variables used are ghostlier. Item *b* allows triggering a condition from a condition when the new condition involves fewer or as many expressions, and that each of them is lower (with respect to $\prec$) than an expression of the old condition, which is used at most once. Item *c* is about generating a directed constraint from a directed constraint. The left-hand side must be strictly ghostlier but the condition on the right-hand side is more complicated. It must be lower with respect to $\preccurlyeq$ if the variable is newly allocated, otherwise, we need only the right-hand side to be lower or equal.

Theorem 20.2

Let V be a subset of $\mathscr{V}$ with finite depth, and $U := \mathbb{U}_V$ (see definition 19.5 "Elementary ghost constraints" (page 248)).
Relation $\rightarrowtriangle$ is well-founded on U.

Proof. Previous lemmas directly imply that $\rightarrowtriangle$ is well-founded on $\mathbb{A}_V$ and on $\mathbb{A}_V$. Moreover, since we can go at most once from $\mathbb{A}_V$ to $\mathbb{A}_V$, if there is an infinite decreasing sequence of elements of U, then it must either be an infinite decreasing sequence of elements of $\mathbb{A}_V$ or have an infinite decreasing suffix of elements of $\mathbb{A}_V$. By contraposition, $\rightarrowtriangle$ is well-founded on U. □

This does not directly provide termination, because this theorem assumes that we only work with a subset of $\mathscr{V}$ of finite depth. The trick is that order $\rightarrowtriangle$ consumes complexity on expressions each time we allocate a new ghost variable (this is item *c* of definition 20.11). We can only allocate a ghost variable under the left-hand side of a directed constraint, thus, no variable under the right-hand side. Indeed, we recall that we assumed that the left-hand side of assignments never appear in the right-hand side. Thus, we can consume complexity on the right-hand side of directed constraints only finitely many times. The consequence is that we can only allocate a finite depth of variables, from a finite support, thus, the last theorem applies. We just need to take V to contain the actual support and any variables that can be added in the maximum number of allowed allocations. Name M this number, we can only allocate variables that are at most M levels under a living variable: this set has finite depth.

This relation is the one all domains use. Once again, this order is good enough for all cases we met, but it has not particular status. We can use any terminating order, and it is likely we need to adapt $\rightarrowtriangle$ in the future.

20.5 Binary Operations

20.5.1 An Illustration of the Problem

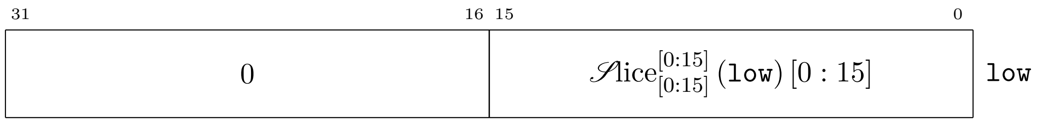
In our concrete framework, binary operations are only union. In practice, they also include widening, intersection and narrowing (for termination and refinement). But these operations are difficult to perform with operands that have different supports: if a ghost variable exists in only one operand, it is not clear what to do with it. We need to know the semantics of the ghost variable to decide, but only the domain that owns is aware of the semantics.

Thus, binary operations are split into two steps: we unify the supports, and we run the actual binary operation. Domain unification is the role of function $\mathcal{B}$ of a generic abstract domain. Before diving into the details, let us illustrate the solution and the problem on an example.

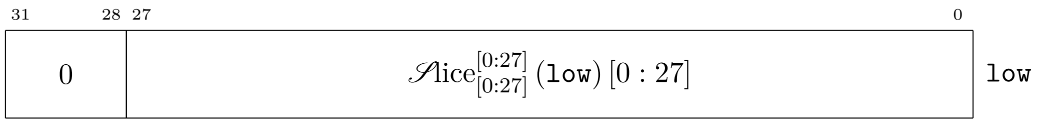
```
1  int x, low;
2  if(?) {
3      low = x & 0xffff;   // low = x & (2^16 − 1)
4  } else {
5      low = x & 0xfffffff;   // low = x & (2^28 − 1)
6  }
```

Listing 20.4: A join with different support

In Listing 20.4, the support always contains two real variables: `x` and `low`. After the third line, we need another ghost variable: $\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low})$ that stores the value of `x`, and 𝔖𝔩𝔦𝔠𝔢 knows that


31
16
15
0
0
$\mathscr{S}\text{lice}^{[0:15]}_{[0:15]}(\texttt{low})\,[0:15]$
low

After line 5, the support contains the ghost variable $\mathscr{S}\text{lice}^{[0:27]}_{[0:27]}(\texttt{low})$, which gets the value of `x` and the domain remembers that


31
28
27
0
0
$\mathscr{S}\text{lice}^{[0:27]}_{[0:27]}(\texttt{low})\,[0:27]$
low

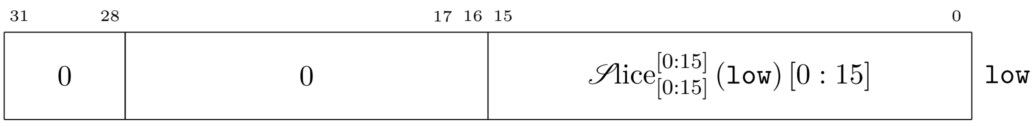
To compute the state after line 6, we need to compute the union of these states, but this is not possible directly. Since the supports are different.

We can see there are 3 regions:

– bits 0 to 15 of `low` are, in both branches, bits 0 to 15 of a ghost variable;
– bits 16 to 27 of `low` are zeroes in a branch, and known to be bits 16 to 27 of a ghost variable in the other branch;
– bits 28 to 31 of `low` are zeroes in both branches.

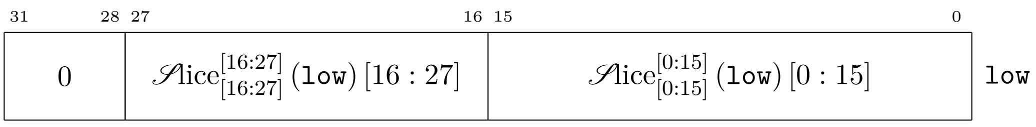
The union is trivial and precise for bits 0 to 15 and bits 28 to 31. But, the domain know that it will have nothing to say for the region between bits 16 and 27. Thus, to unify the support, it is allowed overapproximate the states; hopefully, unification can be done without loss of precision.

After line 3, the state becomes



31
28
17
16
15
0
0
0
$\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}\,(\texttt{low})\,[0:15]$
low

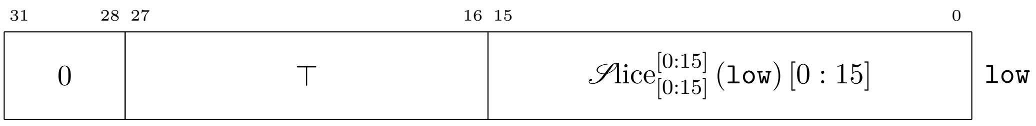
and it allocates a useless ghost variable $\mathscr{S}\mathrm{lice}^{[16:27]}_{[16:27]}\,(\texttt{low})$. After line 5, the new state is



31
28
27
16
15
0
0
$\mathscr{S}\mathrm{lice}^{[16:27]}_{[16:27]}\,(\texttt{low})\,[16:27]$
$\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}\,(\texttt{low})\,[0:15]$
low

Here, it needs to add the new ghost variables $\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}\,(\texttt{low})$ and $\mathscr{S}\mathrm{lice}^{[16:27]}_{[16:27]}\,(\texttt{low})$, assign them the value of $\mathscr{S}\mathrm{lice}^{[0:27]}_{[0:27]}\,(\texttt{low})$ and then delete the latter.

Now, in both state, the support is made of real variables, $\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}\,(\texttt{low})$ and $\mathscr{S}\mathrm{lice}^{[16:27]}_{[16:27]}\,(\texttt{low})$. We can perform the union, and we get



31
28
27
16
15
0
0
$\top$
$\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}\,(\texttt{low})\,[0:15]$
low

That is, after line 6, bits 0 to 15 of `low` are bits 0 to 15 of $\mathscr{S}\mathrm{lice}^{[0:15]}_{[0:15]}\,(\texttt{low})$, which can be remembered, for instance by $\mathfrak{Equality}$ domain, to be equal to `x`. Bits 16 to 27 are unknown, and bits 28 to 31 are zeroes. We can now delete $\mathscr{S}\mathrm{lice}^{[16:27]}_{[16:27]}\,(\texttt{low})$ that is useless. In practice, we can compress steps not to allocate $\mathscr{S}\mathrm{lice}^{[16:27]}_{[16:27]}\,(\texttt{low})$ at all, since it is easy to predict that it will be useless after the union. The implementation indeed avoids allocating such short-lived variables to save time, but it causes strange intermediate states. In this example, we would not allocate $\mathscr{S}\mathrm{lice}^{[16:27]}_{[16:27]}\,(\texttt{low})$ in the else branch. Consequently, after unification, bits 16 to 27 of `low` are $\top$. One may think it is a huge loss of precision, but the domain allows it only because it will eventually happen. Yet, it may be surprising to see such not-immediately-necessary overapproximation.

20.5.2 Formal Description

To perform support unification, domains have a function

$$\mathcal{B} : \mathbb{N}^* \times \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to (\mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathscr{G})^2$$

The function takes two states whose support should be unified and a positive integer. The integer is the layer of ghost variables we need to unify. For each state, the function return a set of variables to add, a set of variables to kill and a constraint-DAG.

For each level i, $\mathcal{B}$ should be called with states where ghost variables of level 0 to $i-1$ are already unified, and it unifies ghost variables of level i (i.e. variables in $\mathscr{V}_i$). These constraints are formally expressed in assumption 20.11 “Assumptions on support unification function” (page 262).

To unify the whole support, we need to successively call $\mathcal{B}$ for each level, increasingly. Returned constraint-graphs are evaluated just like for unary operations, and we should collect all constraints recursively to get the best precision.

Ensuring termination of this process is quite tricky. Since each round consists in applying unary operations on the two abstract states, they will all clearly terminate. But, we should still have a finite number of rounds.

Unlike regular assignments that can lead to a finite but arbitrarily high number of new ghost variables depending on the right-hand side expression, unification assignments are meant to make both states similar. The form of the forest of ghost variables can change, but its depth must stay the same. It is a reasonable constraint since adding a variable where there was none means guessing some information from nothing, which seems dubious. Thus, to ensure termination, we dictate that the depth of the ghost variable tree should not increase.

Definition 20.12 – Depth of an abstract state

Let $a \in \mathbb{D}^\sharp$ be an abstract state. We call depth of a the value

$$\operatorname{depth}(a) := \max\{\operatorname{depth}(v) \mid v \in \operatorname{supp}(a)\}$$

The depth of an abstract state is well-defined since the support is always finite.

After unification of a and b, the depth of resulting states a' and b' should not exceed $\max(\operatorname{depth}(a), \operatorname{depth}(b)) + 1$. It is not a natural property but a political one. It is enforced by ignoring creation of new variables too deep and constraints about them. Actually, with reasonable domains, especially domains described in following chapters, unification never allocate variables beyond the limit and this forced termination protocol is never triggered.

Chapter 21

Base-offset Pointer Domain

Pointer abstraction is often necessary in C, and is implemented in Astrée as explained in [Min06; Min13]. Though the old domain is perfectly compatible with the new framework for product of domains with ghost variables, there are reasons that make interesting to adapt it into the framework described in chapter 20 “Generic Abstract Domains and Reduced Product” (page 253).

In this chapter, we will see how this domain work from a mathematical point of view. We will compare the old implementation with what the new framework allows, and describe the design of the new version of this pointer domain, from a quite formal point of view. Implementation aspects will be treated later.

21.1 Abstracting Pointers

This section is about the formal foundations of pointer abstraction described in [Min06; Min13], and it can be safely skipped if the reader is comfortable enough with this abstract domain.

In most cases, to build pointer values, we take the address of a variable, and perform pointer arithmetic. A pointer can be seen as the address of the beginning of a variable, and an offset relative to this address. Let us denote $\mathcal{V}$ the set of variables. Pointers are elements of $\mathcal{V} \times \mathbb{Z}$. If the offset is too big (or negative), the pointer points after the end of the variable (resp. before the variable). It is not a problem as long as we do not dereference it. Thus, we allow even illegal pointers. For instance, in Listing 21.1 (where we assumed `sizeof(int) == 4`), after lines 3 and 4, the pointer is out of `x`, dereferencing it would be an error, but the pointer can still be modified and made valid before being dereferenced.

An imprecise pointer value is a set of such ordered pairs, that is an element of $\mathcal{P}(\mathcal{V} \times \mathbb{Z})$. We abstract it as an ordered pair of $\mathcal{P}(\mathcal{V}) \times \mathcal{P}(\mathbb{Z})$. The first component is bounded by the cardinal of $\mathcal{V}$, and can be represented explicitly. The second component is delegated to be abstracted by a numerical domain.

Pointer arithmetic is translated into operations on the offset. Adding or subtracting an integer directly performs the same computation on the offset, up to a constant factor

```
int x[4];
int* p = &x[0];  // p = (x, 0)
p--;             // p = (x, -1 × 4)
p += 5;          // p = (x,  4 × 4)
p--;             // p = (x,  3 × 4) = &x[3]
*p = 42;
```

Listing 21.1: A pointer that is valid, then out of bounds, then valid again

depending on the type. To compute the distance between two pointers, we need the two pointers to point to the same variable, and we compute the difference of offsets, divided by the size of the type. To check that two pointers points to the same variable, sets of pointed variables should be equal singletons.

We can represent well-formed pointers with this technique, but we need additional special values for non-regular cases. Especially, we add a special value `NULL`, for C's null pointer, and a special value `INVALID` for all values that have not been obtained from the address of a variable and some pointer arithmetic. For instance, pointers obtained with bitwise operations, like explained in previous chapters, will result in special value `INVALID`. We can hope to refine such a pointer into a precise pointer, using other domains.

21.2 Old Implementation vs. New Framework

As explained in subsection 13.2.3 "The Stack of Domains" (page 167), and especially shown on Figure 13.1 "Structure of the composite domain of ASTRÉE " (page 168), the current pointer domain is parametric: it takes a numerical domain as parameter that it uses to represent offsets and numerical values.

In the old implementation, variables are represented by identifiers chosen by the struct domain. Pointer domain uses the same identifier for the numerical values in its underlying domain, and thus can use only one numerical value for each variable. This has a serious consequence on the representation of variables that may hold an integer and pointer value. To know if the underlying numerical value should be interpreted as an offset or directly as the value of the variable, there is an additional special base. While pointer values are represented by a variable the pointer points to and an offset, for numerical values, this base variable is a special value `NUM`. An integer value n will be represented by the pair $(\texttt{NUM}, n)$. In a case where integer and pointer values are mixed, this leads to major imprecision. Let us illustrate on Listing 21.2. In the if-branch, `p` get a numerical value 42. The representation is $(\texttt{NUM}, 42)$. In the else-branch `p` points to variable `x`, with a byte-offset of 4; the representation is $(\texttt{x}, 4)$. After line 7, once the union is performed, the abstract value is $(\{\texttt{NUM}, \texttt{x}\}, \{4, 42\})$. At this point, the abstract value is very imprecise: the concretization is made of numerical values 4 and 42, and pointers to `x` with offset 4 and 42. Since pointers can have arbitrary numerical value, and numerical values can point anywhere, this abstract value is supremely imprecise. The new framework allows a much smarter representation. Numeric and pointer domains are at the same level, and share the same variables. Thus, the numerical domain can store directly the value of each variable. Pointer domain may allocate a ghost variable to store exclusively the offset.

```
int x[4];
int* p;
if(?) {
    p = 42;      // p = (NUM, 42)
} else {
    p = &x[1];  // p = (x, 4)
}                // p = ({NUM, x}, {4, 42})
```

Listing 21.2: A variable containing an integer and a pointer value

We can go even further and allocate multiple ghost variables to be the offset of each base variable, including for the `NUM` special base. This allows possible numerical values and pointer values not to be mixed. For instance, in the example of Listing 21.2. We would obtain the abstract value $\{(\texttt{NUM}, \{42\}), (\texttt{x}, \{4\})\}$. The effective value of `p` is still unknown, but we do not mix both values. Thus, if we test whether `p != 42`, then, we know that `p` points to `x[1]`.

The new framework allows several domains to represent pointers and to cooperate. In the old version of the pointer domain, the variable `y` of the example of Listing 20.1 "A simple example that needs ghost variables" (page 253) would receive the value `INVALID` as it is not constructed by taking the address of a variable and performing pointer arithmetic. Having a sibling domain that is more comfortable with this kind of operations can help to refine the state, by sharing the constraint $\texttt{x} = \texttt{y}$, which allows the pointer domain to refine `INVALID` into the value $(\texttt{t}, \texttt{idx})$.

Indeed, some special cases of bitwise operations are implemented in the pointer domain, with some assumptions on alignment. This used to be the only solution, but it is quite immoral from a software engineering point of view. In the new framework, it is easy to add a sibling domain that is able to handle another kind of properties.

Another benefit of the adaptation of the pointer domain is that it can use the top level numerical domain rather than having its own underlying one. Using a single instance of the numerical domain allows it to express relation between real numerical values and offsets.

21.3 Adaptation in the New Framework

Let us give some formal aspects of the adaptation of the pointer domain in the new framework. Abstract transfer functions are quite similar to the old version, we will not linger on that matter. We are mainly interested into ghost variable-related elements.

First, this domain defines a single role: $\mathscr{O}$ffset. If we want an offset for each base, we would need a role $\mathscr{O}\text{ffset}_v$ for each variable v. At first, we use only one role $\mathscr{O}$ffset to be as close as possible to the old implementation.

Generated constraints are simply translations on the offset of the effects of pointer arithmetic. For instance, the instruction `q = p + i`, where `p` is a pointer and `i` and integer, generates the directed constraint $\mathscr{O}\text{ffset}(\texttt{q}) \leftarrow \mathscr{O}\text{ffset}(\texttt{p}) + i \times \texttt{sizeof(t)}$ where t is the type pointed by `p`.

```c
int x[4]:
int* p = &x[1];
int* q = &x[3];
int diff = q - p;
```

Listing 21.3: Difference of two aliasing pointers

Though most reductions are quite straightforward, there is a quite natural case where the intuitive approach fails. Let us examine Listing 21.3. After line 3, `p` and `q` point to `x`, with offsets $\mathscr{O}\text{ffset}(p)$ and $\mathscr{O}\text{ffset}(q)$, which are respectively equal to 4 and 12 (assuming `sizeof(int) == 4`). The assignment of line 4 is correct since both `p` and `q` point to `x`. It is appealing to generate from this assignment the directed constraint $\texttt{diff} \leftarrow \frac{\mathscr{O}\text{ffset}(q)-\mathscr{O}\text{ffset}(p)}{4}$, but this would be illegal, since it does not comply the rule of increasing ghostliness (see item f of assumption 20.10 “Hypotheses on constraint triggering” (page 261)). Indeed, ghost variables have an owner, which is the only domain that can generate directed constraints about it (and thus, assign it). Here, `diff` is a real variable, thus it is not owned: all domain will perform the real assignment, and thus `diff` is not necessarily $\top$, and thus, the reduction is not equivalent to an assignment. There are two workarounds, but both are based on the same idea: not to use a directed constraint. The (undirected) constraint $\texttt{diff} = \frac{\mathscr{O}\text{ffset}(q)-\mathscr{O}\text{ffset}(p)}{4}$ is correct and respects all hypotheses. The two possibilities are either to issue it in ghost variables reduction process, or delay it until the ordinary reduction, *à la* [Cou+06b].

Due to its simple usage of ghost variables, support unification prior to binary operations is also pretty elementary. The only possible case where there is something to do is when a variable in an operand has an offset while the same variable in the other operand does not. The simplest strategy is to add the missing offset variable with a $\bot$ value. Since this variable is unused, the concretization of the whole state is not empty. This might be a problem with domains implementing $\bot$-coalescence that have been implemented in the old framework and is not aware that there are ghost variables, and $\bot$-coalescence does not apply if the ghost variable is unused. Fortunately, it is not necessary to initialize the new unused ghost variable with $\bot$; any overapproximation is correct. Providing an arbitrary value can lead to loss of precision, but we can initialize the new ghost variable to the value it has in the other operand, where it is defined.

This domain complies with the termination relation defined in section 20.4 “Termination” (page 266). Let us explain the intuition. The only relevant things this domain can deduce are about offsets. Translating a pointer operation in the offset world increase the ghostliness of constraints. There is still the problem of allocation of new ghost variables, that could make the depth of tree of ghost variables infinitely deep. This domain allocates a ghost variable only when a variable, which is newly allocated or used to hold a numerical value, gets a pointer value. In this case, the offset of the left-hand side receives the offset of the right-hand side, which is strictly ghostlier, ensuring termination.

21.4 Possible Improvements

Since a perfect analyzer does not exist, we should keep in mind that our objective is to be able to perform analysis without false alarms on software we are interested in, and make it fast enough not to exhaust user's patience. From that perspective, and given the current applications, this domain does not need improvements. Yet, we can envisage future applications that could highlight limitations of the current state of this domain. Let first note that the new implementation of this domain is more precise than the old one[*]. Every precision problem explained thereafter also applies to the old implementation, though the proposed solutions for the new implementation might not be feasible in the old one.

The most radical precision improvement, as mentioned before would be to define roles $\mathscr{O}\text{ffset}_v$ for all variable v. Offsets of different bases would not be mixed anymore. This would probably have a major impact on performance as we need to apply ghost constraints on each ghost variable, which can be many. A hybrid solution would be to keep the role $\mathscr{O}\text{ffset}$, and only use $\mathscr{O}\text{ffset}_v$ for variables v for which a distinct offset could increase the precision. This can either be decided by heuristics or provided as an analysis hint by the user. Indeed, in most cases, when a pointer can point to several variables, they are analogous, and the respective offsets are too.

There might still be interesting to add a role to define ghost variables that store the pure numerical value. For now, this is simply the value of the variable. But when a variable can be a pointer, its numerical value can be arbitrary, and thus we lose any purely numerical data. By adding a role $\mathscr{N}\text{umericValue}$[†], one could isolate the numerical values aside. The integer interpretation of the variable would still be imprecise, but we would know that it can be a precise integer value, or a precise pointer value. This would be useful for programs where variables can be used with two different types. This is idiomatic in PYTHON and other dynamically typed languages: since we can test the type of variables at run-time (e.g. using `isinstance`), we can distinguish cases. In statically typed languages, like C, there is no way to know if a variable contains a pointer value or an integer, thus it is not that common to mix both kinds of values. It can still happen as a tagged union, but then it would be relevant to partition according to the tag.

[*]The new implementation might be slightly less efficient. See section 25.3 "Performance Evaluation" (page 320) for a more detailed investigation about execution time.

[†]With the previous notation, we could have written $\mathscr{O}\text{ffset}_{\texttt{NUM}}$, but it may look weird since the numerical value is not an offset.

Chapter 22

Slices Domain

The main application that motivated the new framework is the precise abstraction of bitwise operations. This chapter details this domain. We will show our choice of roles, explain it and exhibit their semantics. Then we will describe the abstract properties handled by this domain, and the concretization. Finally, we will explain how to perform ghost reduction.

In the following, we will assume this domain work with a simple equality domain. We need it to decide whether two variables are definitely equal or not. It is required only to implement two cases: it has to deduce that two variables x and y are equal when it gets the assignment $y := x$ or the constraint $x = y$ (of course including the directed constraint $x \leftarrow y$). Without this very elementary domain, slices domain is much less interesting.

For the sake of simplicity, we will assume that variables are all 32-bit wide, and consequently the set of values $\mathbb{I}$ is $[\![0, 2^{32} - 1]\!]$.

22.1 Roles and Semantics of Ghost Variables

We already saw roles of this domains in examples. Formally, this domain defines roles

$$\mathscr{S}\text{lice}^{[a:a+s]}_{[b:b+s]} \text{ with } \begin{cases} 0 & \leqslant & a & \leqslant & a+s & \leqslant & 31 \\ 0 & \leqslant & b & \leqslant & b+s & \leqslant & 31 \\ 0 & \leqslant & s & \leqslant & 30 \end{cases}$$

Simple counting allows us to find that this defines $\sum\limits_{i=2}^{32} i^2 = \frac{32(32+1)(2\cdot 32+1)}{6} - 1 = 11439$ distinct roles. Informally, the semantics of a ghost variable $\mathscr{S}\text{lice}^{[a:a+s]}_{[b:b+s]}\, v$ is the set of states where its bits a to $a+s$ (both bounds included) are equal to bits b to $b+s$ (both bounds included) of v. We excluded $\mathscr{S}\text{lice}^{[0:31]}_{[0:31]}$ because it would simply state that both variables are equal.

More formally

$$\left(\mathscr{S}\text{lice}_{[b:b+s]}^{[a:a+s]}\left(v\right)\right) := \left\{ s : \mathscr{V} \rightharpoonup \mathbb{I} \;\middle|\; \begin{array}{c} v \in \operatorname{supp}\left(s\right) \\ \mathscr{S}\text{lice}_{[b:b+s]}^{[a:a+s]}\left(v\right) \in \operatorname{supp}\left(s\right) \\ \mathscr{S}\text{lice}_{[b:b+s]}^{[a:a+s]}\left(v\right)\left[a : a+s\right] = v\left[b : b+s\right] \end{array} \right\}$$

We see that we only allow the domain to express equality between contiguous sets of bits. This implies that the statement `y = x & 0xff00ff` requires the creation of two ghost variables: $\mathscr{S}\text{lice}_{[0:7]}^{[0:7]}\left(\mathtt{y}\right)$ and $\mathscr{S}\text{lice}_{[16:23]}^{[16:23]}\left(\mathtt{y}\right)$ that both get the value of `x`. More complex roles could allow stating the equality of arbitrary patterns, but this solution was not retained. Indeed, it is much more complex to handle in the implementation: contiguous slices are naturally orderable, but arbitrary patterns are not. Especially, it is much easier to check that parts of a sliced variable are disjoint when we only use contiguous slices.

Moreover, extracting non-contiguous bits is not a common operation in our context. Building segment descriptors (see subsubsubsection 2.3.2.2.1 “Segment Descriptor” (page 19)) or paging structures (see subsection 2.3.3 “Paging” (page 25)) only uses contiguous slices.

From a more theoretical point of view, any set of role that is able to express equality of two arbitrary bits of two variables is expressive enough. If the number of roles is an issue, we can reduce it to $32^2 = 1024$, we would only need the roles

$$\mathscr{B}\text{it}_b^a \text{ with } \left\{ \begin{array}{ccccc} 0 & \leqslant & a & \leqslant & 31 \\ 0 & \leqslant & b & \leqslant & 31 \end{array} \right.$$

Informally, the semantics of $\mathscr{B}\text{it}_b^a(v)$ is the set of states where the a^{th} bit of $\mathscr{B}\text{it}_b^a(v)$ is equal to the b^{th} bit of v. Reducing the number of roles can be useful for languages where types have a run-time cost (assuming we cannot share a single type for all roles of the same form). The drawback is obvious: this setup requires to allocate many ghost variables. Since ghost variable allocation is relatively slow, using slices seems to be a good trade-off between performance and maintainability.

22.2 Abstract Values and Concretization

This domain establishes relations between ghost variables it owns and their immediate parents, but from the point of view of each layer of variables, the abstraction is non-relational. In particular, there is no relation between ghost variables under distinct real variables. Variables are split in non-overlapping segments of bits. Each segment can have 4 states: zeroes, ones, unknown or a slice of an immediately ghostlier variable.

Formally, we define the set of values of a slice[*]:

$$\text{𐀡} := \left\{\text{ZERO}, \text{ONE}, \top\right\} \uplus \left\{ (a, b) \in [\![0, 31]\!]^2 \;\middle|\; 0 \leqslant b - a \leqslant 30 \right\}$$

[*]We recall that we use Linear B characters for short-lived variables (see page 238). The character 𐀡 stands for the syllable “po”, like “possibilities”, since it stands for the possible values of a slice.

The meaning of ZERO, ONE and $\top$ are quite intuitive. Pairs of integer stands for a slice of a ghostlier variable. We do not need to name it since its name is deducible from the bounds of the represented slice (slice of the parent variable) and the representing slice (slice of the ghost variable).

The set of slices is defined as[†]:

$$\text{𐀯} := \left\{ (i, j, s) \in [\![0, 31]\!] \times [\![0, 31]\!] \times \text{ᒉ} \,\middle|\, \begin{array}{l} 0 \leqslant j - i \\ s \in [\![0, 31]\!]^2 \Rightarrow j - i = \pi_2\left(s\right) - \pi_1\left(s\right) \end{array} \right\}$$

Slices are made of 3 components: the first index, the last index and the content of the slice. When the content of the slice is a slice of a ghostlier variable, sizes must be the same, and such a slice can be at most 31-bit long.

The state of a variable is a sequence of slices where indices form a partition of $[\![0, 31]\!]$, and is formally defined as[‡]:

$$\text{𐀞} := \left\{ \left(s_i\right)_{i \in [\![1,n]\!]} \in \text{𐀯}^\star \,\middle|\, \begin{array}{l} n \geqslant 1 \\ \pi_1\left(s_1\right) = 0 \\ \pi_2\left(s_n\right) = 31 \\ \forall i \in [\![1, n-1]\!], \pi_2\left(s_i\right) = \pi_1\left(s_{i+1}\right) + 1 \end{array} \right\}$$

Expression $\text{𐀯}^\star$ stands for the KLEENE star of 𐀯 (see section A.6 "Standard Objects" (page 376)).

An abstract state is a map from a support V to 𐀞, with some constraints. If a variable v has a slice $(i, j, (a, b))$, the variable $\mathscr{S}\text{lice}_{[i:j]}^{[a:b]}\left(v\right)$ should exist. Formally,

$$\mathbb{D}^\sharp := \left\{ f : \mathscr{V} \rightharpoonup \text{𐀞} \,\middle|\, \begin{array}{l} \forall v \in \operatorname{supp}\left(f\right), \\ \quad \textbf{let } \left(s_i\right)_{i \in [\![1,n]\!]} := f(v) \textbf{ in} \\ \quad \forall i \in [\![1, n]\!], \\ \qquad \textbf{let } (c, d, (a, b)) := s_i \textbf{ in} \\ \qquad \pi_3\left(s_i\right) \in [\![0, 31]\!]^2 \Rightarrow \mathscr{S}\text{lice}_{[c:d]}^{[a:b]}\left(v\right) \in \operatorname{supp}\left(f\right) \end{array} \right\}$$

In practice, we also want ghost variables belonging to the slice domain to be made of a unique $\top$ slice. Indeed, given a variable x and a ghost variable $\mathscr{S}\text{lice}_{[c:d]}^{[a:b]}\left(x\right)$, bits outside the segment a to b are useless. If bits from a to b are a `ONE` slice, we could simply say that bits c to d of x form a `ONE`-slice and not use a ghost variable. If they are a slice, we could transfer it to x as well. This is also true if bits a to b of x are made of several slices.

Now we have abstract states, it is time to define the concretization, which is what allow us to decide what is sound. We have already given a good insight on the con-

[†]Character 𐀯 stands for the syllable "si", which is the closest of "slice" available.

[‡]Symbol 𐀞 is the syllable "pa", for "partition", and because its shape looks like a sliced vertical line.

cretization. Let us state it formally

$$
\begin{aligned}
&\gamma : \mathbb{D}^\sharp \to \mathbb{D}\\
&f \mapsto \left\{ g : \operatorname{supp}(f) \to \mathbb{I} \;\middle|\; \begin{array}{l}
\forall v \in \operatorname{supp}(f),\\
\quad \textbf{let } ((a_i, b_i, s_i))_{i \in [\![1,n]\!]} := f(v) \textbf{ in}\\
\quad \forall i \in [\![1,n]\!],\\
\quad \left\{ \begin{array}{lll}
s_i \in [\![0,31]\!]^2 & \Rightarrow & g \in \left\{ \mathscr{S}\text{lice}_{[a_i:b_i]}^{[\pi_1(s_i):\pi_2(s_i)]}(v) \right\}\\
s_i = \texttt{ONE} & \Rightarrow & v[a_i : b_i] = 2^{b_i - a_i + 1} - 1\\
s_i = \texttt{ZERO} & \Rightarrow & v[a_i : b_i] = 0
\end{array} \right.
\end{array} \right\}
\end{aligned}
$$

In other words, if a slice is equal to a slice of a ghost variable, this relation must be true in the concretization. If a slice is `ONE`, then all bits of this slice are ones. If a slice is `ZERO`, then all bits of this slice are zeroes. If it is $\top$, the slice can have arbitrary value: there is no constraint. Using the semantics of ghost variables, the concretization is quite direct.

22.3 Ghost Constraints

As usual, there are two kinds of reductions:

- reductions that are useful to keep the abstract state precise, constraining only ghost variables belonging to the slices domain, which are mainly expressed as directed constraints;
- reductions that communicates interesting facts to other domains about variables that do not belong to the slices domain, which are actually what the domain is made for, e.g. notifying other domains when a value is rebuilt through bitwise operations.

We will detail these categories successively.

22.3.1 Directed Constraints

Directed constrains are generated to react to a directed constraint or to an assignment. Until now, expressions where arbitrary trees of constructors, but we need to open the structure of expressions to explain the reductions.

There are two categories of operators: bitwise operations that are precisely abstracted, and other operations (including arithmetic ones) that are handled very coarsely. Result of operations of the second kind are abstracted by $\top$. Operations that are precisely abstracted are divided in three categories:

- unary operators: bitwise negation (`~`);
- binary operators between two sliced values: bitwise and (`&`), bitwise or (`|`), bitwise xor (`^`), join ($\sqcup$) and meet ($\sqcap$);
- binary operators whose right operator is a literal integer: left shift (`<<`), and right shift (`>>`).

It is to emphasize that a shift with a non-literal second operand is considered as an computation that the domain cannot abstract precisely.

For operations of the second category (binary operations with two sliced operands), variables must be unified, that is we should transform them so that they share the same partition of their bits, though content can be different.

We will show how to evaluate a directed constraint (or an assignment) for a single slice (i, j). These rules should be applied on each slice, independently. Another step may follow to perform compression. Slices may not been known in advanced and simply cut again as needed.

Given a slice fine enough and an abstract state, we define the function

$$\langle\cdot\_\cdot\rangle : \mathscr{E} \to \{\texttt{ZERO}, \texttt{ONE}, \top\} \cup (\llbracket 0, 31 \rrbracket \times \llbracket 0, 31 \rrbracket \times \mathscr{E})$$

that gives the value of a single slice as one of the three constants `ZERO`, `ONE` and $\top$, or a 3-tuple (a, b, e) meaning that this slice is the equal to the slice a to b of expression e.

e	`ZERO`	`ONE`	(a, b, u)	$\top$
$\langle\cdot\texttt{~}e\cdot\rangle$	`ONE`	`ZERO`	$\top$	$\top$

These binary operations are commutative. Not to overcharge the following tables, we omit the lower half.

$\langle\cdot e \text{ \& } f\cdot\rangle$	`ZERO`	`ONE`	(c, d, v)	$\top$
`ZERO`	`ZERO`	`ZERO`	`ZERO`	`ZERO`
`ONE`	▨	`ONE`	(c, d, v)	$\top$
(a, b, u)	▨	▨	A	$\top$
$\top$	▨	▨	▨	$\top$

$$\text{where } A := \begin{cases} (a, b, u) & \text{if } \begin{cases} (a, b) = (c, d) \text{ and} \\ u \equiv v \end{cases} \\ \top & \text{otherwise} \end{cases}$$

where $\equiv$ means that these expressions are provably equal. We can see that, for A, we need to decide the equality of ghost expressions, and thus, we need the equality domain mentioned in the introduction of the chapter. The slices domain can ask the equality domain if two variables are equal[§] using the input communication channel (see section 12.3 “Partially Reduced product” (page 159)) about the precondition.

Likewise, we can give the rules for other binary operations.

[§]Of course, this is a sound approximation: two variables may be in fact equal, while the equality domain does not know it. Indeed, it has limited deduction power, this is why we design abstract domains (like the slices domain) to deduce more equality properties.

$\langle\!\cdot e \mid f \cdot\!\rangle$	ZERO	ONE	(c,d,v)	$\top$
ZERO	ZERO	ONE	(c,d,v)	$\top$
ONE	▨	ONE	ONE	ONE
(a,b,u)	▨	▨	A	$\top$
$\top$	▨	▨	▨	$\top$

$$\text{where } A := \begin{cases} (a,b,u) & \text{if } \begin{cases} (a,b)=(c,d) \text{ and} \\ u \equiv v \end{cases} \\ \top & \text{otherwise} \end{cases}$$

$\langle\!\cdot e \mathbin{\hat{}} f \cdot\!\rangle$	ZERO	ONE	(c,d,v)	$\top$
ZERO	ZERO	ONE	(c,d,v)	$\top$
ONE	▨	ZERO	$\top$	$\top$
(a,b,u)	▨	▨	A	$\top$
$\top$	▨	▨	▨	$\top$

$$\text{where } A := \begin{cases} 0 & \text{if } \begin{cases} (a,b)=(c,d) \text{ and} \\ u \equiv v \end{cases} \\ \top & \text{otherwise} \end{cases}$$

$\langle\!\cdot e \sqcup f \cdot\!\rangle$	ZERO	ONE	(c,d,v)	$\top$
ZERO	ZERO	$\top$	$\top$	$\top$
ONE	▨	ONE	$\top$	$\top$
(a,b,u)	▨	▨	A	$\top$
$\top$	▨	▨	▨	$\top$

$$\text{where } A := \begin{cases} (a,b,u) & \text{if } \begin{cases} (a,b)=(c,d) \text{ and} \\ u \equiv v \end{cases} \\ \top & \text{otherwise} \end{cases}$$

$\langle\!\cdot e \sqcap f \cdot\!\rangle$	ZERO	ONE	(c,d,v)	$\top$
ZERO	ZERO	$\bot$	ZERO	ZERO
ONE	▨	ONE	ONE	ONE
(a,b,u)	▨	▨	A	(a,b,u)
$\top$	▨	▨	▨	$\top$

$$\text{where } A := \begin{cases} (a,b,u) & \text{if } \begin{cases} (a,b)=(c,d) \text{ and} \\ u \equiv v \end{cases} \\ \bot & \text{otherwise} \end{cases}$$

Shifts do not modify the content of slices, but change their position in the result. Before stating the semantics of shifts, we will define functions that keep only bits 0 to 31 of modified sliced expressions, that is sliced variables that are not subjected to the usual bounds $[\![0,31]\!]$. Let us define modified sliced expressions. First, the state of a single slice, then a slice, the set of sliced expressions and modified sliced expressions

$$
\begin{aligned}
\text{ㄣ}' &:= \{\mathrm{ZERO}, \mathrm{ONE}, \top\} \uplus \{(a,b,u) \in [\![0,31]\!] \times [\![0,31]\!] \times \mathscr{E} \mid 0 \leqslant b-a \leqslant 30\} \\
\text{卅}' &:= \left\{ (i,j,s) \in [\![0,31]\!] \times [\![0,31]\!] \times \text{ㄣ}' \;\middle|\; \begin{array}{c} 0 \leqslant j-i \\ s \in [\![0,31]\!] \times [\![0,31]\!] \times \mathscr{E} \Rightarrow \\ j-i = \pi_2(s) - \pi_1(s) \end{array} \right\} \\
\text{ǂ}' &:= \left\{ (s_i)_{i\in[\![1,n]\!]} \in \text{卅}'^{\star} \;\middle|\; \begin{array}{l} n \geqslant 1 \\ \pi_1(s_1) = 0 \\ \pi_2(s_n) = 31 \\ \forall i \in [\![1,n-1]\!], \pi_2(s_i) = \pi_1(s_{i+1}) + 1 \end{array} \right\} \\
\widetilde{\text{ǂ}}' &:= \left\{ (s_i)_{i\in[\![1,n]\!]} \in \text{卅}'^{\star} \;\middle|\; \begin{array}{l} n \geqslant 1 \\ \pi_1(s_1) \leqslant 0 \\ \pi_2(s_n) \geqslant 31 \\ \forall i \in [\![1,n-1]\!], \pi_2(s_i) = \pi_1(s_{i+1}) + 1 \end{array} \right\}
\end{aligned}
$$

Modified sliced expressions are sliced expressions but which are can cover more than 32 bits. Their primary purpose is to be an intermediary in shift operations, before cutting the surplus. More precisely:

- expression-slices ($\text{ㄣ}'$) are intermediary values in computations, slices refer to an expression (and not a ghost variable);
- positioned expression-slices use expression-slices;
- sliced expression of $\text{ǂ}'$ use expression-slices, but respect the $[\![0,31]\!]$ bounds;
- modified sliced expressions of $\widetilde{\text{ǂ}}'$ are like expressions of $\text{ǂ}'$, but they may overflow the bounds $[\![0,31]\!]$. They will be cut later to get a value of $\text{ǂ}'$.

Now, let us define the cut-left function, that removes everything below the bit 0:

$$
\begin{aligned}
\wr\_| : \widetilde{\text{ǂ}}' &\to \widetilde{\text{ǂ}}' \\
((a_i, b_i, s_i))_{i\in[\![1,n]\!]} &\mapsto \left((a'_i, b'_i, s'_i)\right)_{i\in[\![l,n]\!]} \\
&\quad \text{where } \begin{cases} l \in [\![1,n]\!] \\ 0 \in [\![a_l, b_l]\!] \\ \forall i \in [\![l+1,n]\!], (a_i, b_i, s_i) = (a'_i, b'_i, s'_i) \\ (a'_l, b'_l) = (0, b_l) \\ s_l \in [\![0,31]\!]^2 \Rightarrow s'_l = (\pi_1(s_l) - a_l, \pi_2(s_l)) \\ s_l \notin [\![0,31]\!]^2 \Rightarrow s'_l = s_l \end{cases}
\end{aligned}
$$

the cut-right function, that cuts everything beyond bit 31:

$$\begin{aligned} \lfloor\_\wr : \widetilde{\ddagger}' &\to \widetilde{\ddagger}' \\ ((a_i, b_i, s_i))_{i\in[\![1,n]\!]} &\mapsto \left((a'_i, b'_i, s'_i)\right)_{i\in[\![1,h]\!]} \\ \text{where} &\begin{cases} h \in [\![1, n]\!] \\ 31 \in [\![a_h, b_h]\!] \\ \forall i \in [\![1, h-1]\!], (a_i, b_i, s_i) = (a'_i, b'_i, s'_i) \\ (a'_h, b'_h) = (31, b_l) \\ s_h \in [\![0, 31]\!]^2 \Rightarrow s'_h = (\pi_1(s_h), \pi_2(s_h) - b_h + 31) \\ s_h \notin [\![0, 31]\!]^2 \Rightarrow s'_h = s_h \end{cases} \end{aligned}$$

and the composition, that keeps only slices for existing bits (i.e. bits from 0 to 31):

$$\begin{aligned} \wr\_\wr : \widetilde{\ddagger}' &\to \ddagger' \\ x &\mapsto \wr \lfloor x \wr \rfloor \end{aligned}$$

We can now define the result of shifts. Let $((a_i, b_i, s_i))_{i\in[\![1,n]\!]} \in \ddagger$ be the sliced expression of variable e and p be an integer. The result of e << p is

$$\wr \left((a'_i, b'_i, s_i)\right)_{i\in[\![0,n]\!]} \wr$$

where

$$\begin{aligned} \forall i \in [\![1, n]\!], (a'_i, b'_i) &= (a_i + p, b_i + p) \\ (a'_0, b'_0) &= (0, p-1) \\ s_0 &= \texttt{ZERO} \end{aligned}$$

Similarly, the result of e >> p is

$$\wr \left((a'_i, b'_i, s_i)\right)_{i\in[\![1,n+1]\!]} \wr$$

where

$$\begin{aligned} \forall i \in [\![1, n]\!], (a'_i, b'_i) &= (a_i - p, b_i - p) \\ (a'_{n+1}, b'_{n+1}) &= (32 - p, 31) \\ s_{n+1} &= \texttt{ZERO} \end{aligned}$$

For an instruction e of any other kind, including a variable, we have

$$\langle e \rangle = (0, 31, e)$$

The second step is compression. We define the function

$$
\begin{aligned}
\llparenthesis\_\rrparenthesis &: \ddagger' \to \ddagger' \\
((a_i, b_i, s_i))_{i\in[\![1,n]\!]} &\mapsto \left((a'_i, b'_i, s'_i)\right)_{i\in[\![1,m]\!]} \\
&\quad \text{such that } \exists f : [\![1,n]\!] \to [\![1,m]\!] : \\
&\quad f(1) = 1 \land f(n) = m \land \forall i \in [\![1, n-1]\!], f(i+1) \in \{f(i), f(i)+1\} \\
&\quad \land\ \forall i \in [\![1, n-1]\!], s_i \in \{\texttt{ZERO}, \texttt{ONE}, \top\} \land s_i = s_{i+1} \Rightarrow f(i) = f(i+1) \\
&\quad \land\ \forall i \in [\![1, n-1]\!], s_i \in \{\texttt{ZERO}, \texttt{ONE}, \top\} \land s_i \neq s_{i+1} \Rightarrow f(i)+1 = f(i+1) \\
&\quad \land\ \forall i \in [\![1, n-1]\!], (s_i, s_{i+1}) \in ([\![0,31]\!] \times [\![0,31]\!] \times \mathscr{E})^2 \Rightarrow \\
&\qquad \textbf{let } ((\alpha_1, \beta_1, \eta_1), (\alpha_2, \beta_2, \eta_2)) := (s_i, s_{i+1}) \textbf{ in} \\
&\qquad \beta_1 + 1 = \alpha_2 \land \eta_1 \equiv \eta_2 \Rightarrow f(i) = f(i+1) \\
&\quad \land\ \forall i \in [\![1, n-1]\!], (s_i, s_{i+1}) \in ([\![0,31]\!] \times [\![0,31]\!] \times \mathscr{E})^2 \Rightarrow \\
&\qquad \textbf{let } ((\alpha_1, \beta_1, \eta_1), (\alpha_2, \beta_2, \eta_2)) := (s_i, s_{i+1}) \textbf{ in} \\
&\qquad \beta_1 + 1 \neq \alpha_2 \lor \eta_1 \not\equiv \eta_2 \Rightarrow f(i)+1 = f(i+1) \\
&\quad \land\ \forall i \in [\![1, n-1]\!], (s_i, s_{i+1}) \in ([\![0,31]\!] \times [\![0,31]\!] \times \mathscr{E}) \times \{\texttt{ZERO}, \texttt{ONE}, \top\}, \\
&\qquad f(i)+1 = f(i+1) \\
&\quad \land\ \forall i \in [\![1, n-1]\!], (s_i, s_{i+1}) \in \{\texttt{ZERO}, \texttt{ONE}, \top\} \times ([\![0,31]\!] \times [\![0,31]\!] \times \mathscr{E}), \\
&\qquad f(i)+1 = f(i+1) \\
&\quad \land\ \forall i \in [\![1, m]\!], a'_i = a_{\min\{j\in[\![1,n]\!] \mid f(j)=i\}} \\
&\quad \land\ \forall i \in [\![1, m]\!], b'_i = b_{\max\{j\in[\![1,n]\!] \mid f(j)=i\}} \\
&\quad \land\ \forall i \in [\![1, m]\!], \textbf{let } j := \min\{j \in [\![1,n]\!] \mid f(j) = i\} \textbf{ in} \\
&\qquad \textbf{let } j' := \max\{j \in [\![1,n]\!] \mid f(j) = i\} \textbf{ in} \\
&\qquad s_j \in \{\texttt{ZERO}, \texttt{ONE}, \top\} \Rightarrow s'_i = s_j \\
&\qquad s_j \in ([\![0,31]\!] \times [\![0,31]\!] \times \mathscr{E}) \Rightarrow s'_i = \left(\pi_1(s_j), \pi_2(s_{j'}), \pi_3(s_j)\right)
\end{aligned}
$$

This function is well-defined since f is completely constraint: we know that $f(1) = 1$, and step by step, we can build it entirely. In natural language, $\llparenthesis\_\rrparenthesis$ merges consecutive slices whose values are contiguous. For instance, the expression

```
(x & 0xf) | (x & 0xf0) | 0xf00 | 0xf000 | (0xf0000 & 0) | (0xf00000 & 0)
```

gives the uncompressed sliced expression

$$
s := \begin{pmatrix} (0, 3, (0, 3, \texttt{x})), (4, 7, (4, 7, \texttt{x})), (8, 11, \texttt{ONE}), \\ (12, 15, \texttt{ONE}), (16, 19, \texttt{ZERO}), (20, 23, \texttt{ZERO}), (24, 31, \texttt{ZERO}) \end{pmatrix}
$$

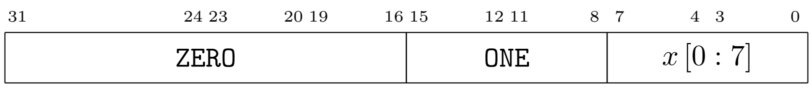
or more graphically

31 — 24	23 — 20	19 — 16	15 — 12	11 — 8	7 — 4	3 — 0
ZERO	ZERO	ZERO	ONE	ONE	$x[4:7]$	$x[0:3]$

But, after compression, we have

$$\llparenthesis s \rrparenthesis = ((0,7,(0,7,\texttt{x})),(8,15,\texttt{ONE}),(16,31,\texttt{ZERO}))$$

that is



Once we got a compressed sliced expression, we can generate directed constraints. Let $((a_i,b_i,s_i))_{i\in[\![1,n]\!]}$ be a sliced expression that is assigned to a variable x. There are two possibilities: the sliced expression may be trivial are not. It is trivial if it is made of a single slice. For instance `(k | 0xffff) | (k | 0xffff0000)` gives the sliced expression $(0,31,\texttt{ONE})$, which is not trivial since it is not the original expression. But, `(k & 0xffff) | (k & 0xffff0000)` gives the sliced expression $(0,31,(0,31,\texttt{k}))$, which is trivial.

If the sliced expression is not trivial, we can emit interesting directed constraints. For all $i\in[\![1,n]\!]$, if s_i has the form (α_i,β_i,e_i), we shall generate the directed constraint

$$\mathscr{S}\text{lice}^{[\alpha_i:\beta_i]}_{[a_i:b_i]}(x) \leftarrow e_i$$

after having allocated $\mathscr{S}\text{lice}^{[\alpha_i:\beta_i]}_{[a_i:b_i]}(x)$ if needed. Unused ghost variables under x can be removed.

22.3.2 Other Reductions

If the sliced expression is trivial, we can again distinguish two cases. Either the slice is the original expression or not. We have given two examples of the latter case above. Let us give an example of the former: the expression `(k & 0xf) + 1` simply gives the sliced expression $(0,31,(0,31,\texttt{(k \& 0xf) + 1}))$. In this case, the domain deduced nothing interesting. When the expression e result in the trivial sliced expression $(0,31,(0,31,f))$ with $e\neq f$ the sliced expression is not uninteresting, and thus we should emit a constraint. But the constraint $x\leftarrow f$, when x is the assigned variable, does not match the requirements of directed constraints. But we can generate the regular constraint $x=f$, which can be helpful. In particular if the f is a single variable, when the equality domain receives this constraint, it merges equivalence classes of x and f.

This is what happens when a variable is reconstructed bitwise, like in the fourth line of Listing 20.1 “A simple example that needs ghost variables” (page 253).

This reduction is the primary purpose of this domain, but it can emit other reductions. For instance, if, in the assignment `x = e`, `e` results in the sliced expression $((0,i,(0,i,f)),(i+1,31,\texttt{ZERO}))$, with $i\in[\![0,30]\!]$, the domain can deduce the constraint $\texttt{x}\equiv f\ \left[2^{i+1}\right]$. This may be useful for a numerical domain.

22.4 Possible Improvements

A few improvements have been considered. The first one is to generalize the pattern of ghost variables, to allow more than contiguous slices. For now, a bit-segment of a ghost variable can be used to represent a bit-segment of its parent variable. We may want to generalize this to any pattern. This matter has been discussed previously and was judged difficult to implement and useless, as we only cut contiguous slices in our application. In the future, for other use cases, it might be useful to represent more complicated patterns using a single ghost variable, if performance is an issue. It would require $2^{32} - 2$ roles (since we exclude a variable that represent no bit, or all bits).

A more complicated but interesting question is the sharing of ghost variables. Ghost variables are useful to keep copies of expressions at a point of the execution. For instance, ghost variables in Listing 20.1 “A simple example that needs ghost variables” (page 253) all store the same value: the value of `x` (which is not modified). It could be a major improvement to allow sharing ghost variables with a copy-on-write strategy. We could go even further: when a ghost variable is a simple copy of a variable, we could just use the ghost variable, still with a copy-on-write strategy. When a variable is used instead of a ghost variable, it is watched for modifications. If it is modified (or killed), we allocate a ghost variable to store its former value, and we use it instead. Unfortunately, it works only when the expression cut by bitwise operations is a simple variable. Moreover, in most cases, this variable will indeed be killed, thus, we probably should allocate at least one ghost variable in the first place. Sharing ghost variables is only an implementation issue.

One can also add new reductions according to the needs. For instance, any variable whose lowest bit is in a `ZERO` slice is even; and if the lowest bit is in a `ONE` slice, the variable is odd. There are many such cases that must be discovered and implemented when they become necessary for future use cases.

Chapter 23

Other Domains

Other domains in this new framework are interesting enough to be mentioned, but each of them does not deserve a separate chapter. So, they will all share the same.

We will first look at adapting existing numerical domains. We will see an example of numerical domain that may use ghost variables, and a domain to handle linear combinations of pointers, that is not implemented for now, as it needs an additional feature in the framework of domain cooperation with ghost variables to be efficient.

23.1 Plain Old Numeric Domain

Any domain that existed in the old framework can be adapted for the new one. There are two cases: either it uses ghost variables, and we need to adapt this usage (see chapter 21 "Base-offset Pointer Domain" (page 275)), or it is a simple domain without ghost variables. We want here to explain the latter as it is the case of the composite numerical domain of Astrée. More generally, any domain that does not use ghost variables can be adapted in the same way.

Intuitively, such a domain needs no ghost variable and thus has no role. Ghost variable-related features are trivial, and do nothing. Let us explain what to do with each component of a generic abstract domain, as shown in definition 20.3 "Generic abstract domain" (page 258).

The domain name $n \in \mathfrak{N}$ is arbitrary, but it is useless to give it any roles. Most components are unchanged:

- $\mathbb{D}^\sharp$
- $\gamma : \mathbb{D}^\sharp \to \mathbb{D}$
- $\sqcup : \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$
- $\mathrm{lfp}^\sharp : (\mathbb{D}^\sharp \to \mathbb{D}^\sharp) \to (\mathbb{D}^\sharp \to \mathbb{D}^\sharp)$
- $\textsc{Assign} : \mathscr{A} \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ (natural extension of the assignment of real variables)
- $\textsc{Guard} : \mathscr{C} \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ (idem)
- $\textsc{Alloc} : \mathscr{V} \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ (idem)
- $\textsc{Kill} : \mathscr{V} \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ (idem)

- EXTRACT : $IO^\sharp \times \mathbb{D}^\sharp \to IO^\sharp$ (adapted as explained in section 20.2 "Product of Domains" (page 263))
- REFINE : $IO^\sharp \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ (idem)

But ghost variable-related objects are yet to be defined:

- $\lessdot \subseteq \mathscr{U} \times \mathscr{U}$: this relation is useful to ensure termination, and should be well-founded. When domains generate ghost constraints, new constraints must match the relation defined by all involved domain. Here, a domain without ghost variable has no such requirement, and it would be a shame to choose an order that uselessly restrict the set of allowed constraints. There are two ways to avoid this problem:
  - If all domains share a common order, as explained in section 20.4 "Termination" (page 266), we just use the same here. In this case, this relation will be just as restrictive as the one in other domains.
  - We allow this relation not to be well-founded for domains that does not use ghost variables. Since such a domain does not generate ghost constraints, there is no termination problem. Moreover, other domains are still subjected to a well-founded relation, as long as there is at least one domain that use ghost variables. If there is no such domain, termination is ensured because no domain generates constraints. With this formalism, we choose $\lessdot = \mathscr{U} \times \mathscr{U}$ as it accepts everything.
- $$\begin{aligned} \mathcal{U} : \mathscr{U} \times \mathcal{P}(\mathscr{U}) \times \mathbb{D}^\sharp &\to \mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathscr{G} \times \mathcal{P}(\mathscr{U}) \\ (u, U, a) &\mapsto (\varnothing, \varnothing, g_0, L) \end{aligned}$$

  where g_0 is the constraint-DAG that contains no constraint, and L is a set of constraints allowed of the next step, according to the choice made for $\lessdot$. Formally,

  $$g_0 := (\{n_0, n_1\}, \{(n_0, n_1)\}, u_0)$$

  where n_0 and n_1 are two arbitrary nodes and u_0 is a function with empty support. Formally

  $$u_0 := (\{n_0, n_1\}, \mathscr{U}, \varnothing)$$

  according to the definition of functions (see section A.5 "Function Theory" (page 374)). This DAG has two nodes, one edge, with no constraint; its source is n_0 and its sink is n_1. Due to the definition of constraint-DAG, we need at least one edge, thus, we cannot have a DAG made of a single node that is both the source and the sink. This is to reject more easily disconnected constraint-graphs. This remark and such trivial graphs have already been mentioned in section 19.4 "Ghost Constraints" (page 247).

  This $\mathcal{U}$ generates no new constraint, does not allocate or kill new variables.
- $$\begin{aligned} \mathcal{B} : \mathbb{N}^* \times \mathbb{D}^\sharp \times \mathbb{D}^\sharp &\to (\mathcal{P}(\mathscr{V}) \times \mathcal{P}(\mathscr{V}) \times \mathscr{G})^2 \\ (i, a, b) &\mapsto ((\varnothing, \varnothing, g_0), (\varnothing, \varnothing, g_0)) \end{aligned}$$

Like $\mathcal{U}$, since the domain defines no role, there is nothing to do for ghost variables. Here, we can see that supports are always unified since the set of ghost variables belonging to the domain is always empty.

23.2 Compute Through Overflow

Though existing numerical domains can be adapted to the new framework by making it ignores ghost variables, the converse is not true: there are numerical domains that can be interested in ghost variables.

An example of such a domain is a domain that handles compute through overflow (CTO). To work properly, it needs both signed and unsigned versions of considered variables. But often, only one of them is a real variable, the other is a ghost variable[*]. We will not define precisely this domain as it is outside the scope of our work, but we will give some aspects related to ghost variables.

This domain needs two roles: $\mathscr{S}$ignedOfUnsigned and $\mathscr{U}$nsignedOfSigned. In the old framework, ghost variables where simply variables in an underlying domain, just as the base-offset domain. To adapt a CTO domain to the new framework, directives given to the underlying domain must be concerted into constraints. This is very similar to the adaptation of the base-offset domain (see chapter 21 “Base-offset Pointer Domain” (page 275)).

Though this adaptation is less systematic as the one for domain without ghost variable (see section 23.1 “Plain Old Numeric Domain” (page 293)), the method used for base-offset is a very didactic example whose pattern can be followed with few adaptations.

23.3 Linear Combinations

In subsection 18.1.3 “Linear Combinations of Pointers” (page 234), we have explained how linear combinations of pointers may be useful. This could be solved using linear combinations of ghost variables, that store the values of terms of the linear combination. This domain is not implemented for now as it is not useful yet, but it has a major difficulty. Let us show an example to identify it.

Let us study Listing 23.1. In lines 6 and 9, we assign `r` a pointer to array `x`, with offset 1 and 3 respectively, to which one adds an arbitrary value `p` (or the equal value `q`). At line 11, `p` is removed from `r`, so that `y` is only `&x[1]` or `&x[3]`. At line 6, `r` is the symbolic sum of two ghost variables, containing values of `&x[1]` and `p`. These ghost variables can be written $\mathscr{T}\mathrm{erm}_1(\mathtt{r})$ and $\mathscr{T}\mathrm{erm}_2(\mathtt{r})$. Indices are arbitrary, since there is no meaningful way to discriminate both variables terms: they have the same coefficient, addition is commutative.... For the sake of the example, we simply follow the syntactic order. At line 9, we have the same support: two ghost variables under `r` $\mathscr{T}\mathrm{erm}_1(\mathtt{r})$ and

[*]There are some exceptions. Typically, when we use a type like `union {int s; unsigned int u;};`. Here, both the signed and unsigned values are real variables.

```
1  int x[4];
2  int* p = ?;
3  int* q = p;
4  int* r;
5  int* y;
6  if(?) {
7      r = &x[1] + p;
8  }
9  else {
10     r = q + &x[3];
11 }
12 y = r - p;
```

Listing 23.1: A simple code with a difficult join

$\mathscr{T}\text{erm}_2(\mathtt{r})$, but here, the former contains the value of `q`, while the latter contains the value of `&x[3]`.

After line 10, we need to perform a join. Supports are already unified, but a direct join would be very imprecise. Let us sum up in a table:

	$\mathscr{T}\text{erm}_1(\mathtt{r})$	$\mathscr{T}\text{erm}_2(\mathtt{r})$
if branch	`&x[1]`	`p`
else branch	`q` (= `p`)	`&x[3]`

But if variables are renamed in one branch, we would obtain a much nicer state, for instance, if we rename variables in the else branch:

	$\mathscr{T}\text{erm}_1(\mathtt{r})$	$\mathscr{T}\text{erm}_2(\mathtt{r})$
if branch	`&x[1]`	`p`
else branch	`&x[3]`	`q` (= `p`)

In this case, the join is much more precise. After the union, `r` is the sum of two ghost variables: $\mathscr{T}\text{erm}_1(\mathtt{r})$ that points to `x` with byte-offset 4 or 12, and $\mathscr{T}\text{erm}_2(\mathtt{r})$ that stores the value of `p` (which is `q`). Line 12 can be precisely abstracted using a simple equality domain and noticing that $\mathscr{T}\text{erm}_2(\mathtt{r})$ and `p` are equal.

In such a domain where ghost variables have a symmetrical meaning, abstracting precisely binary operations may require renaming ghost variables. This matter will be developed in section 27.2 “Ghost Variable Renaming” (page 333).

Chapter 24

Implementation

THE IMPLEMENTATION of the proposed framework is not a direct conversion into code of the formal description. For maintainability, debugging and performance reasons, the way it operates is very different from the mathematical point of view.

In this chapter, we explain how the framework is implemented, what are the differences between the mathematics and the implementation, and it explains why the implementation was designed this way.

The question of the results obtained by this implementation will be exposed in chapter 25 "Discussion" (page 315); in particular, execution time will be discussed in section 25.3 "Performance Evaluation" (page 320).

24.1 Overview

This new product of domains with shared ghost variables has been implemented in ASTRÉE, thus, using OCAML language [Ler+20]. The programming language is very important as it guides architecture choices. They would have been very different in C++ or PYTHON, for instance. In particular, in OCAML, it is common to make extensive use of modules and functors*. The implementation of this new framework implied an estimated† addition of 30 kloc and the deletion of 14 kloc, among the 200 kloc of ASTRÉE. Moreover, it resulted into external libraries and tools for a total of 17 kloc of OCAML and 11 kloc of PYTHON. Overall, integrating the new framework can be easily qualified as a major change in ASTRÉE.

The general architecture is shown on Figure 24.1, where each node is a module. Module `key dispenser` belongs to `translator`, and `translator` belongs to the `driver`; they are presented on the side since they are not abstract domains. Their purpose will be explained later (respectively in section 24.2 "Roles" (page 299) and section 24.3 "Trans-

*In C++, we would probably have used template classes.

†This is an estimation because it is difficult to distinguish changes directly related to the new framework from other changes. Indeed, this new framework is closely connected to the support of inline assembly. Moreover, it is difficult to take changes in internal libraries into account, as they were motivated by the new framework, but also globally useful.

lator" (page 301)). Everything in red has been implemented for the new framework. The struct domain is paced here for reference, so that it is easy to place this composite domain in the global structure of ASTRÉE (see Figure 14.1 "New structure of the composite domain of ASTRÉE" (page 179)). We can also remark that we reused numerical domains thanks to an `adapter`.

There are two important facts to know to understand the implementation. First, the evaluation strategy is very different from the one explained in the formalization. Here, it is a dialog between domains and the `driver` that manages the execution of ghost constraints: they are still generated and enforced in a distributed way, but constraints all go through the `driver` that acts as the conductor. Moreover, theoretical aspects are mainly absent from the implementation. Typically $\lessdot$, the relation enforcing the termination as seen in definition 20.3 "Generic abstract domain" (page 258), has no place in the code. It should be satisfied, but the order is not implemented by itself. Likewise, roles are totally implicit: constraints linked to roles should still hold, but they are not made explicit as the variable identifier. There are roles in the implementation, but they are a different notion from the mathematical one, though they often overlap.

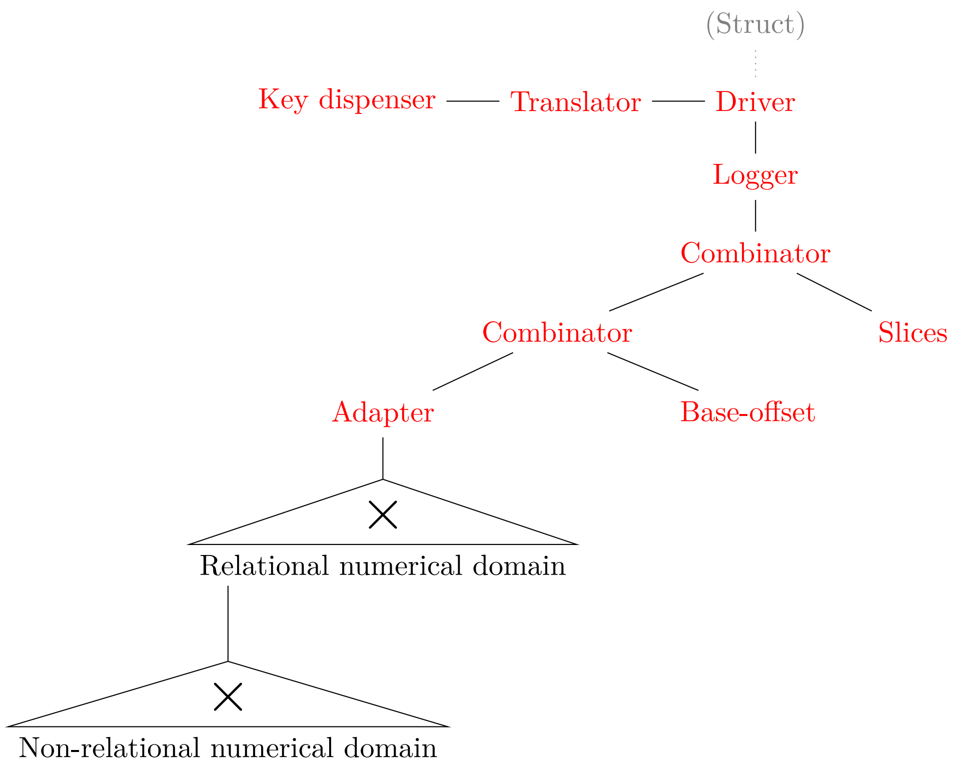


Figure 24.1: New structure of the composite pointer domain of ASTRÉE

24.2 Roles

In Astrée, under struct domain, variables represent memory cells and are identified by a tag and an integer. Such a pair is called a “key”. Tags encode some information about the type of the variable, but it is a very technical concern we are not interesting in. In the following, the key will be reduced to the single integer field. Keys are generated by the struct domain, as a function of the type, the memory block and the position of the cell in its memory block. To allocate new variables, we need new identifiers, but struct domain can use any integer, thus, we need to dynamically build a correspondence between outer keys, allocated by struct domain, and inner keys, allocated by the `driver`, that include keys of ghost variables. Everything coming from the struct domain should be translated to use inner keys. Likewise, when something is communicated back to the struct domain, concrete types should be translated to use outer keys. Quite often, values given to the struct domain are not really used, but only provided back to the pointer domain at the next instruction[‡]. To avoid two useless translations, we made an intensive use of abstract types, so that the struct domain cannot introspect these values, and thus they do not need to use outer keys. The old framework was also adapted in the same way to keep compatibility. Indeed, the new and the old framework share the same signature and are still both usable[§]. In fact, when we do not need the slices domain (or any new domain), we should rather use the old framework since it provides better execution time (see section 25.3 “Performance Evaluation” (page 320)), and it is probably more reliable, because it was already heavily used.

Inner keys are generated by the `key dispenser`. In OCaml, it is usual to choose functional style. Each module defines an immutable type `t`, that stores the state of the module. Such types usually include fields for underlying modules’ `t` types. Transfer functions does not modify the value in place, but return a new object for the result. Since data are persistent, sharing of substructure allows this style to be efficient.

But for the `key dispenser`, we need a mutable (imperative) state. This is closely linked to the notion of roles from the implementation point of view. Let us explain what happen when analyzing Listing 24.1. At line 4, we need the creation of a ghost variable to represent the offset of `p`; at line 7, we also need such a variable (in the else-branch, it was not already allocated). At the end of the conditional branching, we shall join both states. The offset of `p` of each branch was represented by a separate value. In particular, the state of the else-branch is not aware that a ghost variable allocation occurred in the if-branch. But we cannot use twice the same key for different variable, as the join would merge variables that have nothing in common, thus the state of `key dispenser` must be shared globally. Moreover, we would like to use the same key for the offset in both branches, so that, after line 8, no renaming is needed, and we can perform the join directly.

For this reason, the `key dispenser` has a mutable state that is shared between all

[‡]For instance, communication channels: the deduced constraint about the postcondition is the constraint provided for the precondition of the next instruction.

[§]It only takes a command line flag to switch between the two frameworks.

```c
int x[4];
int* p;
if(?) {
    p = &x[1];
}
else {
    p = &x[3];
}
```

Listing 24.1: Creating a ghost variable in two branches

values of `driver`'s type `t`. This is also why roles are useful in the implementation: as they encode the semantics of the ghost variable, two variables with the same role get the same key. Roles in the code can be less precise than the one in the formalization. Indeed, we are only interested in identifying variables for binary operations. For instance, we defined the role $\mathscr{S}\text{lice}_{[c:d]}^{[a:b]}$ to express that bits a to b of the ghost variable are equal to bits c to d of the overlaying variable. Here, we do not need to know which slice of the ghost variable is useful, we only need to know which slice of the overlaying variable is represented. If bounds (a, b) are different between the two states, it is the responsibility of the unification process (see section 20.5 "Binary Operations" (page 271)) to act accordingly. During the unification process, ghost variables representing a slice can be shifted so that the same segment is used; we can also give up and default to $\top$ (as the theoretical framework does).

This has another consequence: ghost variables may not have roles. This is unadvised since a new key will be generated each time, and the unification is more complicated since two variables without roles (so with different keys) may need to be unified: that is we allocate a third key, and we rename both old values into this new key.

`Key dispenser` state is purely incremental: it starts with an empty state. When a new key is requested, there are two cases:

– if a role is provided, the `key dispenser` looks for a key that would have been already provided for this role and returns it if so, otherwise it generates a new key and add the correspondence between the role and the key to its registry;
– if no role is provided, a new key is returned and will never be returned again.

Roles are defined inductively from domains: each domain provides a `module Role` implementing a common `module type ROLE` signature, shown in Listing 24.2. Function `val get_keys: t -> int list` returns the list of keys of overlaying (more real) variables. Unlike the implementation, a role is not always a unary constructor. In particular, an example of zeroary role will be given later. Roles of arbitrary arity have been also dreamed of, but not used for now. There is a subtle terminology point: in the formalization, roles where unary constructors, here, the role is the constructor and overlaying variable.

`Combinator` functors define the `type t` of their `module Role` a sum type of `type t` of

```
module type ROLE =
  (sig
    type t
    val compare: t -> t -> int
    val pp: Format.formatter -> t -> unit
    val get_keys: t -> int list
  end)
```

Listing 24.2: ROLE signature

underlying domains' `module Role`. Finally, the `key dispenser` knows two kinds roles: real and virtual[¶]. Ghost variables have a role of the form `Virtual r` where `r` is a value of the role defined inductively from domains. Roles of real variables have the form `Real k` where `k` is the outer key. It agrees with the notion of role (what the variable is useful for): a real variable represents the value of the variable identified by the outer key `k`.

This way of defining roles enforce a useful property: domain cannot see roles that they do not define. For instance, in Figure 24.1 "New structure of the composite pointer domain of ASTRÉE" (page 298), the combinator on the top define their roles as the disjunction of roles of the bottom combinator and roles defined by `slice` domain. The bottom combinator builds its roles from roles of the `adapter` and the `base-offset` domain. But the `slice` domain cannot see roles defined by `base-offset` domain. This way, domains are allowed to allocate only ghost variables they own, since they (usually) need to provide the role of the variable (and its overlaying variable). This allows building the owning relation, which can be checked when ghost constraints are generated or when the ghost variable is killed.

The `driver` also defines and can own ghost variables. This will be explained later.

24.3 Translator

Translating directives and constraints between inner and outer keys is the job of the `translator`. This module should keep correspondence between inner and outer keys of real variable. It is also in charge of the allocation of new keys. There are two mechanisms.

- For real variables, allocation is ordered by a directive emanating from the struct domain. When the `translator` is asked to translate an allocation directive for a variable with outer key `k`, it gets a new inner key from the `key dispenser` (for the full role `Real k`) and adds it to the correspondence.
- For ghost variables, new keys are requested by domains. To do so, they are given a proxy to the `translator`, which gets a new key from the `key dispenser`. The new key is added to the correspondence later, when ghost variables are allocated (just before enforcing ghost constraints).

[¶]This was the first name of ghost variables, before realizing the concept of virtual variables if in fact the same as the well-known notion of ghost variables.

`Key translator` does more than requesting new identifiers and keeping the correspondence between real variables' inner and outer keys. It also keeps track of the number of uses of ghost variables. Since abstract states can be very complicated, it might be difficult or expansive to check is a variable is still used somewhere. The `translator` provides a reference counting feature. Domains declare how many times it uses each variable, or the variation in the number of uses. When the number of uses comes down to 0, the variable is marked to be deleted at the next round of ghost variable deletion. By then, it is still alive, but unused.

The `translator` is owned by the `driver` (which owns pretty much everything, more or less directly) because it needs to know the inductive definition of roles. Consequently, its state is opaque to domains since they are not aware of roles except their own. But domains need to operate on the `translator` to allocate new variable, or change the number of uses of a variable (including deleting it). They may also need to access data, for instance, the number of uses of a ghost variable or its owner.

To solve this issue, domains are provided a view of the state of the `translator` with some modifications, and returns new modifications to apply, like a ghost variable allocation or reference counter update. Each domain receives the same view, but the set of updates is the union of all updates generated by domain executed before. When accessing the data through the view, one should take care to also take updates into account, since at this point, `translator`'s state is unchanged. When the domains return the control to the `driver`, modifications are applied on the `translator`.

Of course, utility functions are provided to manipulate view and updates transparently, without having to dive into the concrete structure.

In detail, the view is a record (a product type) containing function values. Let us enumerate the 3 types of modifications with their simplified representation:

- a ghost variable creation `NewKey(key, role, uses)`;
- an assignment to the number of uses of a ghost variable `SetUse(key, uses)`;
- an offset in the number of uses of ghost variables `AddUse(key, delta_uses)`.

To delete a ghost variable, one should just set the number of uses to 0. In all cases, modifications are about a single ghost variable, thus, they are stored in a map, whose keys are variable keys. For each ghost variable, there can be at most one modification. Indeed, updates can be merged according to the following (non-commutative) rules:

- `NewKey(key, _, _)` + `NewKey(key, _, _)` ⇒ absurd: one cannot create a variable twice;
- `NewKey(key, role, _)` + `SetUse(key, uses)` ⇒ `NewKey(key, role, uses)`
- `NewKey(key, role, uses)` + `AddUse(key, delta_uses)` ⇒ `NewKey(key, role, uses + delta_uses)`
- `SetUse(key, _)` + `NewKey(key, _, _, _)` ⇒ absurd: one cannot create a variable that already exists, or update the number of uses of a variable that does not exist yet;
- `SetUse(key, _)` + `SetUse(key, uses)` ⇒ `SetUse(key, uses)`
- `SetUse(key, uses)` + `AddUse(key, delta_uses)` ⇒ `SetUse(key, uses + delta_uses)`

– `AddUse(key, delta_uses)` + `NewKey(key, _, _)` ⇒ absurd: one cannot create a variable that already exists, or update the number of uses of a variable that does not exist yet;
– `AddUse(key, _)` + `SetUse(key, uses)` ⇒ `SetUse(key, uses)`
– `AddUse(key, delta_uses1)` + `AddUse(key, delta_uses2)` ⇒
`AddUse(key, delta_uses1 + delta_uses2)`

Thanks to the map structure, and since there is at most one modification for each variable, access to the view with updates is really quick, even if adding a modification requires to perform the merge: the cost of the merge is quite small, and accessing properties on variables is far more current than modifying them.

These modifications allow domains to order allocation and deletion of ghost variables. But this mechanism is not almighty, in particular, it is unable to perform ghost reduction. The `translator` is only interested in the support.

24.4 Driver and Combinator

Just as in [Cou+06b], to glue domains, we use a binary functor, the `combinator`, that takes two abstracts domains in arguments and forms a module that implements the same signature as abstract domains. Since it returns a module with the same signature, instances of `combinator` can be nested, so that, we have a binary tree of abstract domains. The `combinator` runs the given directive on both domains, starting with the left-hand side domain, and performs the exchange of information for reduction.

But, as explained in [Cou+06b], some constraints must be provided to every domain, called broadcast output channel. Thus, the tree is not self-sufficient: we need an additional module at the root to control the iteration in the tree. To broadcast a message, it climbs up to the root, which distributes it everywhere. In the old structure of ASTRÉE, there were no such module, but its job was done by the pointer domain. Here, as we will explain later, iteration is much more complicated and deserve a separate module. This is the `driver`. It is in charge of performing the reduction as before, but also of enforcing ghost constraints, and acts as an adapter between the struct domain and the composite domain with ghost variables.

As mentioned in section 24.1 "Overview" (page 297), the way ghost constraints are handled is very different from the theoretical description. In the formalization, partial applications of functions $\mathcal{U}$ are exchanged between domains, the transitive closure of constraints is computed in each domain and these constraints are run in each domain independently. This strategy has a lot of drawbacks in practice. Firstly, exchanging information in an all-to-all fashion is uselessly costly. Then, each domain computes the same transitive closure of constraints. These two first steps can be centralized: a single module receives all ghost constraint triggering functions, computes the transitive closure and returns it to the domains, which enforce these constraints separately. But, it is still improvable. This strategy does not allow domain cooperation during the evaluation of ghost constraints. Moreover, exchanging partial applications is, to say the least, risky. Though it is often considered as something nice to do with functional languages, it is usually not a good idea for maintainability and debugging. Indeed, when partial applications are buggy, since functions values cannot be introspected, we only know there is a problem when the function is fully applied, which may be a long time after the partial application. Then, when the bug appears, i.e. when the function is fully applied, the context in which the function was partially applied is already out of scope and deleted, preventing us to inspect the state at this point. Moreover, the last partial application is not necessarily the one that prepared the buggy partial application, thus, it is usually difficult and long to come back to the right partial application using the debugger. Overall, in large codebases, long-lived and complex function values are harmful.

In the implementation, we use a very different strategy. When computing an abstract transfer function, typically the application of a unary function (like an assignment or a guard), domains return a new state with a value encoding the progress. There are two kinds of progress values:

- the computation is finished and the progress value holds messages for the output channels (see [Cou+06b]);
- the computation is suspended, and two additional values are returned:
  - a context, a value of a domain-specific type, that allows the domain to be able to resume its computation where it was stopped;
  - an enriched constraint-DAG to apply.

If the computation is finished, the abstract transfer function is called on the following domain, naturally.

If the computation is suspended, the context and constraint-DAG are brought back to the `driver`, the context is saved on a stack, and constraints in the DAG are run using transfer functions of domains. During this process, computation can also be suspended; in this case, the new context is pushed on the stack and the new constraint-DAG is executed. We proceed recursively, and termination is ensured by the argument explained in section 20.4 "Termination" (page 266). When a constraint-DAG is fully executed, the parent instruction is resumed, using the context on the top of the stack. The domain that suspended its computation use its context to remember its state. It may carry its execution until the end, or generate a new constraint-DAG with a new context. Each domain is responsible to generate finitely many constraint-DAGs.

With this execution strategy, enforcement of ghost constraints is synchronized across all domains, which allows domains to cooperate between each step of ghost reduction, improving the precision. We avoid the exchange of function values and computing several times the same transitive closure. Moreover, this is more powerful than the formalization since it allows domains to emit successive constraint-graphs, by inspecting the state resulting from the application of the previous ghost constraints.

As explained before, for better performance, directed constraints are run as assignments. Constraint-DAGs also include guards, and several other instruction linked to the handling and collection of errors, hints communicated across domains and the abstraction of weak updates. For convenience, a special kind of constraint-DAG is implemented as lists of constraints, to encode chain DAGs. They are sufficient in most cases.

Though execution of ghost constraints is synchronized between domains, all previous hypotheses are still important to ensure soundness. Indeed, domains may not see the constraints in the same order. Let us remember the architecture shown on Figure 24.1 "New structure of the composite pointer domain of ASTRÉE" (page 298). Let a be a constraint being executed in each domain. The topmost combinator run it first on its left domain. The bottom-most combinator does the same. We assume that the numerical domain terminates its computation directly. Then, a is run by the base-offset domain, and let us assume it yields a ghost constraint b, with some context. They are forwarded to the `driver` that saves the context and run b. It is executed from left to right, and we assume no domain return additional ghost constraint. The evaluation of b is finished. Now, we resume the evaluation of a in the base-offset domain, thanks to the context it yielded. We assume it carries its execution without new constraints, and so does slices domain. At the end, a and b have been both enforced on all domains, but the numerical domain saw a before b, while the slices domain enforced b before a[**]. Thus, even with this execution strategy, it is important to allow constraints to be run in an arbitrary order. Moreover, the order of ghost constraint is very depend on the position of each domain in the tree, and the correction should not depend on these positions[††].

There is another optimization that is done in the implementation. We remember that an assignment $x := e$ assigns $\top$ to all variables in $x^{\downarrow\triangleleft}$ (the set of variables ghostlier than x), and then may generate directed constraints of the form $y \leftarrow f$ where y is ghostlier than x (that is $y \in x^{\downarrow\triangleleft}$). These two steps are reduced to a single one: rather than assigning $\top$ and then applying a directed constraint (thus, an assignment) on the same variable, we directly run the second assignment, skipping the reset to $\top$. This is correct since x does not appear in e, thus the tree of variables under x is disjoint from the forest of variables under variables in e.

This last hypothesis is central in a lot of arguments we made, but of course, it is not true in regular software. Most C codebases contain many statements like `i++;`, which transgresses this sacred law. The solution is very straightforward, that already as been mentioned in section 19.2 "Semantics" (page 241). We simply need an intermediate variable `tmp` and perform `tmp = i + 1; i = tmp;`. If needed, this translation is dynamically performed by the `driver` using a special ghost variable that is not under any real variable: its role is a nullary constructor. The driver is also able to perform this sanitation on ghost constraints if needed, even if it is not useful for now, it may become useful with more complex ghost structures.

The `driver` is also in charge of support unification: as explained in section 20.5 "Binary Operations" (page 271), it unifies layer by layer the set of living variables. More generally, all operations that must be performed on all domains are ordered by the `driver`.

[**]And the base-offset domain enforced the constraint b while the computation of a was suspended. Domains have the responsibility to be able to enforce ghost constraints (using ghostlier variables) when they suspend their execution. A simple strategy is to delay the modification until the end, by saving useful values in the context. With this strategy, the suspended computation is in fact applied after ghost constraints. Partially applying a may lead to an inconsistent state, but it might not be a problem to enforce ghost constraints, depending on the domain.

[††]Nevertheless, the order might change the execution time and the precision.

Thanks to its predominant situation, the `driver` may also check dynamically that theoretical constraints that are still applicable in the implementation, indeed hold. This is useful to detect errors as early as possible, rather than pursuing the execution with unexpected data, and resulting in a late error that may be nightmarish to investigate. Though, these checks are very time-consuming (see section 25.3 "Performance Evaluation" (page 320)), and should be disabled for non-development usage.

24.5 Abstract Order

To reach an inductive invariant, we need to stabilize loops. We threw the modest veil of $\mathrm{lfp}^\sharp$ on this problem. But, in practice, $\mathrm{lfp}^\sharp$ is not elementary: we need to build it from simpler operators. Usually, we need:

– widening, to ensure termination;
– union, intersection and narrowing, to improve precision;
– a sound approximation of the concrete order (usually, the inclusion), to know when we reach a correct approximation of a concrete post fixpoint.

The two first points are all binary operators, and the way we handle them has been explained in section 20.5 "Binary Operations" (page 271). But checking the inclusion is a slightly different problem. We have to return a truth value, tt or ff, not compute a new abstract state. The common point is that states we compare may have unequal support.

Though we need to perform support unification, we do not need to use a sound approximation of the identity. We simply need to reduce the problem to an equivalent (and hopefully simpler) problem. From states a and b, we generate respectively states a' and b' such that $a' \sqsubseteq^\sharp b' \Leftrightarrow a \sqsubseteq^\sharp b$.

The idea is that there are two steps to check the inclusion:

– abstract states must have the same layout;
– values of ghost variables must be included.

For instance, let us consider the base-offset domain. Let a and b, two abstract states and x a variable. To test whether $a \sqsubseteq^\sharp b$, we need to test the inclusion between the two values of x in a and b. So, we first need to compare the sets of pointed variables (bases). If the set of variables pointed by x in a is not included in the set of variables pointed by x in b, then $a \not\sqsubseteq^\sharp b$. This is an internal operation: we do not need to compare ghost variables, the single domain can already decide that the inclusion does not hold. If the set of bases in a is indeed included in the set of bases in b, we need to compare the offsets: the possible offsets in a should be a subset of possible offsets in b. And thus, we need to ask every domain to compare these ghost variables.

In practice, we proceed in these exact two steps. Firstly, we collect a set of variables to compare, layer by layer. It is initialized by with all real variables (i.e. variables of $\mathscr{V}_0$), then we ask for the set of ghost variables the domains need to compare, and we get a subset of $\mathscr{V}_1$. We iterate until we reach the depth of the support. During this process, a domain can already claims that the state inclusion does not hold, and shortcut the process. Once we get the set of all ghost variables we need to compare, it is provided to all domains, and they should compare only these variables. There are two reasons. Firstly, this improves performance by restricting the set of variable to examine. But, most importantly, this is a precision issue. Even if it exists in both a and b, if a ghost variable is not to compare, it should not be compared, since it might be unused. And if we add extra (and useless) hypotheses, we will reach an inductive invariant more slowly, and the result will be less precise since it required more widening steps.

This process is coordinated by the `driver` and relies on the *bona fides* of domains.

24.6 Domains

Domains in this framework should implement a larger signature than usual domains. For instance, support unification functions are specific to the use of ghost variables. Moreover, some standard functions may behave differently. For instance, the function that applies unary directives (like assignment or guards) may returned an unfinished computation.

Because of these characteristics, writing a domain from scratch in the new framework is not a trivial matter. To simplify this task, we provide directions to guide the developer, and utility libraries to ease its work.

24.6.1 Resumable Transfer Functions

When unary transfer functions emit ghost constraints, they return a context to be able to resume their execution. Moreover, we should ensure they do that finitely many times.

Many transfer functions do not need to emit constraints, but when they do, The general strategy is to use an acyclic transition system with finite paths. In practice, we use acyclic finite-state automata (FSAs) with finite memory to be able to keep results between each computation step. The state, with its memory, is a constructor of a sum type, and each computation step perform a transition. If the state is final, then the computation is completed, otherwise, a constraint-DAG is emitted.

We did not manage to produce a usable framework to easily encode transitions, as we did with the microcode for CPU instructions (see subsection 16.1.3 “Microcode” (page 197)). This is due to the large variety of transitions used depending on the domain or the kind of instruction. Developer libraries general enough to take this variability into account did not make things easier to write.

While transitions are difficult to generalize, we still provide an iterator for finite state automata. Indeed, sometimes, it is more natural to build the graph incrementally, for instance, when working on sliced variables, we generate a chain of sub-constraint-DAGs, each about a slice. But when no slice requires ghost constraints, the global constraint-DAG is empty. The provided iterator can detect empty constraint-DAGs (under its different implementations), and will directly run the next transition without suspending the computation, since there is in fact no constraint to emit.

24.6.2 Numerical Domain Adapter

Writing a domain that is able to handle ghost variables is not a walk in the park, either. There are many things to implement and some of them may be quite tricky. It would be a shame to be subjected to such trouble when writing a domain that has nothing to do with ghost variables. For this reason, we have written an adapter that lift an ordinary domain that needs no ghost variables (like domains written for the old framework), to a domain which handle ghost variables (but still does not use any). This adapter allows all existing numerical domains to be reused for free.

The theoretical aspects have been exposed in section 23.1 "Plain Old Numeric Domain" (page 293). But there is an important point to precise. In the comparison process, the `driver` collects ghost variables to compare, and communicates this set to all domains. But numerical domains have not been written to compare only a subset of variables: they compare all existing variables, and supports must match. To make comparisons works, the adapter deletes all variables that are not to be compared, which solve both problems at once.

Otherwise, the implementation is a quite direct adaptation of the theoretical explanation.

24.6.3 Domain Developer Libraries

To ease the developer work, many internal libraries have been developed to handle various kind of tasks.

Some developer libraries are very specific to ASTRÉE implementation. For instance, `translator`'s views and modifications can be used as a single value from a module implementing a signature very close to relevant parts of the signature of the `translator`. Likewise, abstract states, communication channels and `translator`'s views are all about a single concrete state and are thus always handled together not to risk mixing them. To handle these compound objects, we also provide a rich library to reduce as much as possible the need to look inside. For debugging purpose, we also provide pretty printers for all defined types.

We also defined a lot of more general internal libraries from pieces of code that was recurrent in domains or believed to be useful for the future. They cover a large variety of categories, e.g. memoization, data structures, algebraic computations.... External libraries and tools will be presented in Appendix B "By-products" (page 381).

24.7 Capabilities

Though we use a combination of very heterogeneous domains, we cannot use arbitrary combinations: some domains are required to make ASTRÉE work properly. For instance, while the interval domain is often very imprecise, it has an opinion on everything and is used as based of all abstractions. Another example is the need of base-offset domain. To resolve dereferences, the struct domain should know variables pointed by an expression. Moreover, there is no $\top$ special value in this case: set of bases are handled explicitly (since the set of all bases is finite and reasonably small), thus even to return the most imprecise result, the domain should be aware about existing bases. Thus, we need such a domain (base-offset, in our case), otherwise, some operations are not implemented.

In fact, requirements should not been expressed in terms of domains, but from the point of view of features: a domain can be able to fulfill several tasks, and conversely, a task can be achieved by several domains. To take these requirements into account, a global set of capabilities is defined, and domains specify capabilities they have.

This way, when domains are instantiated, the `driver` can check that all required features are fulfilled. Otherwise, the `driver` immediately raises an error at instantiation-time, thus, before the beginning of the analysis. In addition to required capabilities, we define a set of optional capabilities. They can be useful for domains to enable advanced features that are useless if some featured are not provided by another domain.

Lastly, this is useful for the `driver` to know if there are domains that use ghost variables without roles. Indeed, if all ghost variables have roles, some optimizations hold. Otherwise, they should be disabled as they would make the algorithm incorrect. For example, if all ghost variables have roles, there is no need to sanitize ghost constraints: as long as the real assignment does not use the left-hand side in its right-hand side, operands of directed constraints are always disjoint. But when a domain allows ghost variables without roles, ghost constraints may need sanitation. In this case, we need to use a stack of temporary ghost variables (for each level of ghost constraints that needs sanitation).

24.8 Communication Channels

As explained in [Cou+06b], input channels are implemented as record types of functions: we need a record type since there are multiple values to hold together, and we need functions fields, rather than explicit maps, since they allow lazy computation. We can go even further and memoize these functions to improve performance.

```ocaml
let memoize (type a b) (f: a -> b) : a -> b =
  let t = Hashtbl.create 0 in
  fun a ->
    match Hashtbl.find_opt t a with
    | None -> let b = f a in Hashtbl.add t a b; b
    | Some b -> b
```

Listing 24.3: Mainstream memoization in OCaml

Memoization is usually done with functions as shown in Listing 24.3. But that hides entirely the underlying `f` function and `t` table. When functions are enriched gradually by successive compositions, like in the case of input channels, we end up with many intermediate tables that may not be useful anymore. For this reason, we implemented an incremental memoization library that keeps `f` and `t` visible. It supports several memoization strategies to keep caches reasonably small while minimizing cache misses. This library also makes easy to memoize an arbitrary number of parameters, with different strategy for each of them.

Since we work with sound abstract properties, when we enrich input channels, we obtain a more precise sound abstract property by computing the intersection. Defining a function as the intersection of several functions is very common method to get better precision. Our memoizer is well-suited for this application as a memoized function value can contain a list of functions, and the cache is updated incrementally, so that even partial memoized computations, from the time when the list of functions where shorter, can be completed and updated, without redoing everything. To do so, the memoized function should know the default value (that is $\top$) to be returned when the list is empty, and the intersection function to refine results. Moreover, in some cases, we know when computation cannot be improved any more (or does not need to). For instance, for the question "may pointers x and y be aliased?", default (most imprecise) value is "yes" (or tt), the intersection function is the logical conjunction ($\wedge$). But if a domain certifies that no, x and y may not be aliased, then it is definitive, and nothing will change the result. Symmetrically, for the question "are we sure pointers x and y are aliases?", the default value is "no" (or ff), and the intersection is the logical disjunction ($\vee$). Indeed, if a domain can certify that two pointers are indeed aliases, it is definitively true. In these two cases, once we got the most precise value (ff and tt, respectively), it is useless to try to refine the state anymore. Our memoizer also support this optimization that allows short-circuiting a suffix of the list, if the optimal value was reached.

Sadly, some components of input channels may not have a clear $\top$ value. It was already mentioned in section 24.7 "Capabilities" (page 308) that the set of bases pointed by a pointer expression is given explicitly and thus can only come from a domain aware of such considerations. For such a function, the memoized function object can have a special value that means it is undefined yet: calling it is an error. We can guarantee that a regular value is eventually provided by requiring an adequate capability. It is indeed important to use a special value and not a dummy function (that will always raise an exception, for instance), because the behavior of the enrichment is only a substitution if the field is the dummy value, while it is an intersection otherwise.

Capabilities do not solve the whole problem: to an input channel, a single enrichment request to domains might not be sufficient. Indeed, some functions may have the special dummy value by default, and they can depend on each other (in an acyclic way). Thus, functions that depend on still-dummy function cannot be initialized properly: we need to iterate over all domains until all fields are filled. This is conceptually easy, but requires some implementation tricks to work well with function of various types[‡‡]. Because this kind of iterations may be dangerous, along with capabilities, this process is watched to ensure it terminates: the number of unfilled fields should decrease at each step, otherwise there is a bug. In practice, the number of necessary iteration is only 2, but it depends on the order of domains in the reduced product. A not as smart order may need more steps to get the same result: the point is to minimize the worst-case number of edges in

[‡‡]Currently, it is solved wrapping first class modules in a GADT.

a path of the dependence graph that go from a domain to the right to a domain to the left.

24.9 Logger

With ghost variables, bugs are more common, and they often have visible consequences a long time after the actual error. To help debugging domains, we provide a logger. It is a unary functor that take a domain in parameter and return a module of the same signature. It simply forwards all directives to the underlying domain, and returns its results, but with rich printing of parameters and results. This is achievable since we defined printers for all types that appears in signatures (concrete and abstract types), and printers are part of the signature of domains.

Because it preserves the signature, the logger can be put anywhere. By default, there is only one instance, under the `driver` (see Figure 24.1 "New structure of the composite pointer domain of ASTRÉE" (page 298)). In desperate times, we can make an intensive use of loggers, as shown in Figure 24.2, where we use 5 instances of the logger. This is quite easy since the logger can detect its number of instances and identify themselves, so that the log file is quite readable.

Even with a single instance of the logger, and when analyzing example of a few C lines, it can produce a log file of hundreds of MiB; any small but non-trivial example logs several GiB. To reduce log file to a human-readable size, the logger can be enabled and disabled using annotation in the source code, so that we can log only the section of code that we desire. The other benefit is execution time: I/O are slow; with logging activated everywhere, a very short analysis (less than a second) takes several minutes.

Thousands of hundreds of lines of log can be boring to read, and it is difficult to see at first glance important information, separation between steps. . . . For this reason, we developed an external library: OCOLOR. It makes use of semantic tags of OCAML's `Format` standard module to produce smart and nice pretty printing using ANSI[§§] escape sequences. See section B.1 "OCOLOR" (page 381) for more details about OCOLOR.

[§§]American National Standards Institute

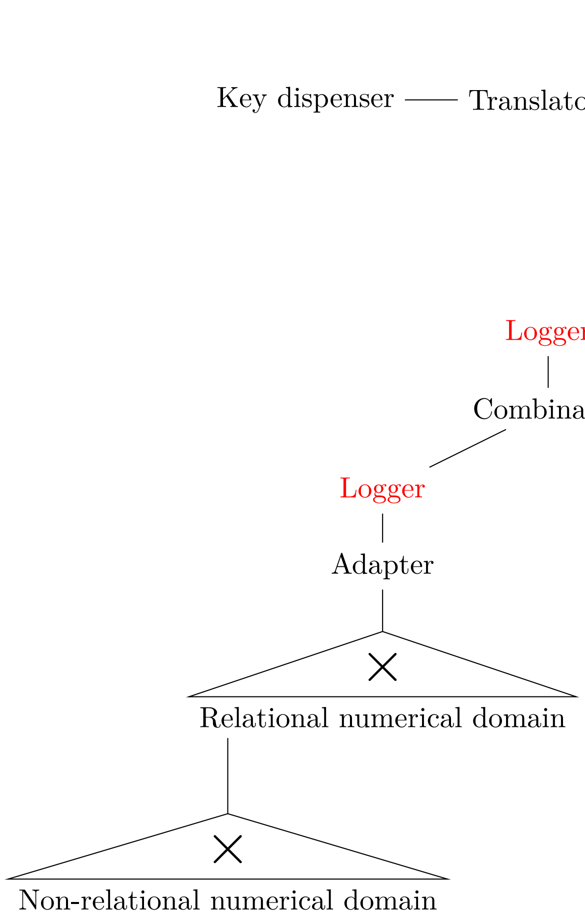


Figure 24.2: An extreme usage of loggers

24.10 Comparison Theory vs. Implementation

As we explained, the implementation is not a direct adaptation of the formal description. Though they match with respect to many aspects, there are some differences. As a rule of thumb, when there are some profound differences, the implementation is more expressive or precise than the formal framework. The implementation of the new framework has been designed to be easily improved, for instance with ideas explained in chapter 27 “Improving Product with Ghost Variables” (page 331).

The first difference we emphasize does not change the expressive power, but can be confusing when working with ghost variables. In the formal framework, an assignment reset underlying ghost variables to $\top$, then directed reductions refine this sound, yet imprecise, state. In the implementation, this is compacted into a single operation. Even if it is clearly equivalent, one should keep this difference in mind when implementing.

The evaluation strategy is also synchronized, while there is no such hypothesis in the formal framework. This is not only because mathematics describes the computation in a more abstract way; synchronization of ghost constraints allows domain cooperation between each step, which is not allowed in the formal framework. One should be careful with these reductions: they should only be about ghostlier variables than those involved in the last constraint. Indeed, overlaying constraints are not yet run by every domain, and the state is consistent only with respect to ghostly enough variables. As we empty the stack of recursive ghost constraints, eventually the abstract state is consistent with respect to all variables.

The current implementation of the reduced product with ghost variables allows domain to emit successive constraint-DAGs. Each constraint-DAG can depend on the state obtained from the previous constraint-DAGs. In the formal framework, domains are only allowed to emit a single constraint graph, that only depends on the constraint and the immediate postcondition of the transfer function (without refinement). This feature is marginally used as we write these lines, but it was judged to be probably interesting for more symbolic domains, for which the transformations induced by a constraint-DAGs may be difficult to predict and thus, new reductions can be useful.

Roles have a slightly different meaning in the implementation and in the mathematical description. Especially, semantics of unary roles is less precise in the implementation, but roles of arbitrary arity are allowed. While, some non-unary roles can be useful (see sanitation in section 24.4 “Driver and Combinator” (page 303)), they have no direct equivalent in the formal world. Each use of such roles should have its own translation in the mathematical framework, needing some tweakings of the semantics to add artificial machinery. Roles with two or more parameters need extra care because they break the forest structure the set of ghost variables has. We need to reinforce the hypotheses ensuring that the left-hand size of assignments does not appear in the right-hand side: since we work with DAGs and not only trees, we need the separation of underlying DAGs, which may be more tricky. Moreover, one should not join both DAGs during such an assignment. A natural property would be to enforce that DAGs under distinct real variables are disjoints; or, even stronger, to require that roles with multiple parameters can only use sibling ghost variables, since it would keep a rigid notion of depth for each ghost variable.

Chapter 25

Discussion

Abstraction quality depends on two main criteria: the precision and the performance. Indeed, there are no ultimate abstraction, they all have limitation, thus we can only judge if it is good enough to treat families of problems we are interested in (or we might consider in the future). Moreover, if the analysis is too slow, it might be conceptually interesting, but, in the end, it is useless as it cannot be used to prove properties on software. Without performance constraints, abstract domains are much easier to design.

We will explore these two points here: first the precision ("Which analyses can be successfully performed?"), then the question of performance ("How patient do I need to be?").

25.1 Successful Analyses

We recall we are interesting at proving safety properties. As usual, we want to prove the absence of RTEs. Moreover, our more original goal is to prove security property of a host platform, like memory isolation. We argued that these properties could be overapproximated by safety properties about low-level memory protection features. Moreover, in practice, the way these features are used is such that these approximate safety properties are quite satisfactory.

In particular, we were interested in proofs of the correctness of segmentation implementation (see subsubsection 2.3.2.2 "Protected Mode Segmentation" (page 19)). This is the original motivation for the slices domain.

25.1.1 Segmentation

We recall that segmentation setup implies cutting pointers using bitwise instructions to fill a segment descriptor (see subsubsubsection 2.3.2.2.1 "Segment Descriptor" (page 19)). Then, when accessing the descriptor, the processor internally rebuild the original pointer. Since we want to prove properties on segment we use, we need to be able to cut and rebuild pointers. The same holds about the size of the segment (“limit” field). This is exactly for what the slices domain was designed.

All segmentation policies are not equally reasonable, some may not work at all (see subsubsubsection 2.3.2.2.2 “Segmenting the Linear Address Space” (page 21) and subsubsection 2.3.3.4 “Translation Lookaside Buffers” (page 28)). In fact, we restricted even further the kind of segmentation our analysis accept, because we know what we expect. This is possible since segmentation is a very static strategy: segments are set up at startup and never modified afterwards[*]. This method would not work with paging which is much more dynamic and proceed to a much more complex address translation. We explore some ideas to prove correctness of paging in section 26.1 “Paging” (page 327).

Slices domain was indeed designed to handle bitwise operations involved in segmentation, and perfectly fulfilled its purpose. It managed to prove that segments are indeed set up according to a variant of the flat model, explained in subsubsubsection 2.3.2.2.2 “Segmenting the Linear Address Space” (page 21) and subsubsection 2.3.3.4 “Translation Lookaside Buffers” (page 28). Shortly said, this method uses trivial segments (starting at 0 and 4 GiB long) for system tasks, and shorter segments for user tasks. Thus, the space at the end of the linear address space is accessible only through system segments. Consequently, we can make these pages persistent in the TLBs to speedup translation, while ensuring no collision with user pages.

A very adjacent problem is the question of GDT entries unlinked to segmentation, like call gates (see subsubsection 2.4.2.3 “Inter-Privilege Calls” (page 31)), and IDT entries (see subsection 2.4.3 “Interrupts and Exceptions” (page 32)). These entries store function pointers in pieces along with various flags (like access rights). Though they are also set at startup, these entries are of a slightly more dynamic nature than segmentation. Their content may be changed after initialization, but, more importantly, we do not exactly know what to expect: fields are not constant values, but slices of functions pointers we do not exactly know. Nevertheless, the process is quite the same. Thanks to the slices domain, abstracting the IDT is basically reduced to the abstraction of an array containing function pointers.

Overall, entries of the GDT are precisely abstracted, and we succeed to retrieve interesting information when needed.

25.1.2 Pointer Alignment

Though the slices domain was made specifically for operations linked to memory protection features, it handles bitwise operations quite generally. Indeed, it was shown to be useful for other tasks that require precise abstraction of bitwise operations. The most notable one is pointer alignment.

Pointer alignment is the propriety of pointer to be a multiple of the word size. In some architecture, like old version of ARM and most RISC (reduced instruction set computer) architecture, pointer alignment is crucial as unaligned access are forbidden.

[*]In old systems, before paging, segments used to be more dynamics. Since it is almost subsumed by paging, segmentation is reduced to a quite trivial use.

In other architectures, like x86 or more recent ARM, there have hardware support of unaligned access, though they often require two memory accesses, and thus are slower. However, even in the very lenient x86 architecture, some structures need alignment[†]. For these reasons, pointer alignment is often an important matter; this is mostly handled by the compiler in usual cases, but for low-lever software, it is usual to bother about it, and having to enforce alignment manually.

In all reasonable architectures[‡], word sizes are a power of 2. Typically, 32-bit values on 32-bit architecture need to be aligned on 4 B-boundaries. The interesting consequence is that alignment can be checked and enforced using bitwise operations on pointers. An address aligned on 4 B-boundaries is a multiple of 4, thus it ends with two 0. Given a pointer `p`, we can check if it is aligned by testing whether `!(p & 0x3)`, and we can align it to the previous boundary using `(p & ~0x3)`.

There are two kinds of code we are interested in: those that require inter-domain cooperation, and those that do not. About the latter, let us look at Listing 25.1. At line 2, slices domain manages to remember that the 2 last bits of `p` are zeroes, which is exactly enough to prove the condition of line 4. In this case, there is no need for domain cooperation.

```
1  int* p = ?;
2  p = p & 0xfffffffc; // Enforcing alignment
3  // ... where p is not modified
4  if(!(p & 0x3)) { // Testing alignment
5      // ...
6  }
```

Listing 25.1: Enforcing and testing alignment

This is interesting, as it was useful in real source code, but very limited as it does not allow arithmetic operations on `p`, or dereference. To do better, we need the slices domain to emit a constraint to improve the state of another domain. In this case, when we clear the k lower bits of a pointer `p`, then the new value is in the interval $\llbracket \mathtt{p} - 2^k + 1, \mathtt{p} \rrbracket$ and $\mathtt{p} \equiv 0 \left[2^k\right]$. Thanks to these pieces of information, numerical and base-offset domains can deduce interesting facts, that allow using the aligned pointer interestingly.

These reductions have been tested in a temporary quick-and-dirty implementation to handle such pointer alignment-related statements in an industrial low-level source code[§] running on a RISC architecture. These instructions are useful for processor's caches management.

[†]Especially, structures involved in memory protection need to be aligned.

[‡]We are never safe from an exotic architecture made by an eccentric electronics engineer.

[§]This was not from the same application as the host platform we use as common thread from the beginning.

25.1.3 Retrying Old Analyses

Since the new framework for domain cooperation is supposed to subsume the old one, and domains have been adapted, it is reasonable to expect that analyses that used to work this the old domain are reproducible with the new domain. In particular, analyses that used to be successful should still produce no alarm, but also, analyses that used to raise alarms should still produce analogous alarms.

After a thorough test-and-debug process on an extensive collection of tailor-made examples, designed to test every feature gradually (since it cannot be made independently), we indeed tried to rerun old analyses. This has two purposes: improve our confidence in the new implementation and evaluate the performance. For both reasons, we used examples of significant size. The performance aspect will be detailed later, in section 25.3 “Performance Evaluation” (page 320).

Rerunning these analyses exhibited a few unknown bugs at the beginning, hard to reproduce on smaller examples. The logger was most useful with such big analyses, especially the activation through annotation feature. Moreover, most bugs were not subtle soundness issues, but important unverified invariant; thanks to sanity checks, they were quite easy to locate.

Once these bugs were fixed, we were glad to notice that the analysis reports indeed matched: we get similar alarms and invariants. Invariants are quite hard to compare: since we use ghost variables, the same property is expressed differently and comparing them requires the expertness of a human eye. Alarms also may not exactly match, but not only because there are ghost variables involved. Indeed, in the old stack architecture, the base-offset used to have a single ghost variable to represent the numerical value and the pointer offset. Now, with the new architecture, the numerical value do not need ghost variables, and the ghost variable is used only to store the offset. The analysis is thus a bit more precise. In practice, it does not allow removing false alarms since this is only useful when performing pointer specific operations (like dereference) on variables that may contains a pointer value or an integer, which is anyway buggy. But alarms with the new version of the base-offset domain are more informative, as we see the offset separately, making easier to track the source of the bug.

Overall, even if the new framework was not used as intensively as the old one (which is in the commercial version of ASTRÉE), we managed to reproduced known results, increasing greatly our confidence in the new framework.

25.2 On the Study Case

With this new framework, and thanks to the support of assembly as described in Part III “Abstraction of Mixed C and x86 Assembly” (page 173), we were able to analyze some parts of the study case. We recall that it is an OS hosting several programs. Some are provided by the manufacturer (our industrial partner); they undergo a strict verification process[¶], and are considered trustworthy. The other applications are installed by the

[¶]Probably as strict as the OS itself, maybe using ASTRÉE as well.

customer, and the manufacturer has very little control over them; we cannot trust them *a priori*. We expect the OS to ensure memory isolation so that untrusted tasks cannot interfere with trusted ones. We are also interested in other similar properties, all about isolation, but many of them are not directly performed by the operating system. For instance, we do not want an untrusted task to use the "`chown`" system call to get the ownership of a file belonging to the system. The implementation of `chown` is out of our scope, but we still want to prove that the system call mechanism is correct: it is protected against illegal usages, it forwards correctly the parameters to the correct function. . . .

We did not directly perform this analysis: for confidentiality reasons, we did not have access to the real code. Our industrial partner gave us representative snippet of codes from which we build the solutions presented before, and performed itself the analysis using a binary of our development version of ASTRÉES that we provided. These snippets may be simple excerpts, or forged examples that illustrate a specific difficulty. Moreover, we did not aim to analyze the whole OS. As explained before and detailed in chapter 26 "Proving Low-Level Properties" (page 327), there are several aspects that we cannot prove for now (typically correctness of paging).

The OS is separated in a few major parts. The first is the initialization: it begins when the boot loader[**] launches the OS, and ends when the OS launches its *init* program. After the boot, this part is never executed again. Other major parts are about the handling of interrupt and exceptions. We can highlight a few particular handlers:

- the handler of the interrupt triggered by the timer that calls the scheduler;
- the handler of the page fault exception that calls the page manager;
- the handler of the interrupt used for system calls, which may also calls the page manager for memory allocation and deallocation.

We were mainly interested in the initialization phase. We manage to analyze it almost entirely: everything, but the very beginning and the end. Indeed, the beginning depends on the boot loader, on which we know very little. Moreover, just after, paging-related setup is performed, which is not handled for now. At the very end, the OS launches its *init* process. This was not analyzed either for want of hypotheses about the loader, and file system primitives.

On this part, we proved the absence of RTEs, and the correctness of segmentation according to a modified flat model (see subsection 25.1.1 "Segmentation" (page 315)). As explained in subsection 16.1.2 "System Registers and CPU State" (page 196), we check that segments match expected requirements when segment registers are loaded.

Paging-related code is not analyzable soon as it requires a precise memory domain to abstract page directories and page tables; this is explained in section 26.1 "Paging" (page 327). Indeed, wanted properties (mainly memory isolation) needs a specialized domain, probably from the arts of shape analysis. Nevertheless, we believe that the other aspects of paging-related code can already be abstracted precisely enough. Likewise, the scheduler needs a specific domain, but otherwise, current abstractions should be satisfactory.

We tried to analyze other parts of the OS, mainly the part that handles syscalls.

[**]A GRUB-like program.

We were at first interested in the absence of RTEs, but we failed because of a single infamous trick, worthy of anathema: a memory access to a deep call frame, across a C call frame[††]. Some ideas to solve this problem are explained in section 26.2 "Access Across Call Frames" (page 329). Apart from this abomination, it seems that the current state of the analyzer is good enough to perform an interesting analysis, but it is yet to be confirmed, since the remaining functions may hide other surprise, and piece-wise analyses are not always representative of the difficulties of a monolithic analysis, and never as satisfactory.

Overall, we expect to be able to analyze much wider parts of the OS, by developing relatively few domains or extensions.

We were told that the analysis of the initialization of the OS takes about 20 min to complete, using the improved version described in section 25.3 "Performance Evaluation" (page 320). This is considered to be an acceptable duration. We trust that the optimizations described in chapter 27 "Improving Product with Ghost Variables" (page 331) would allow performing the same analysis in less than 10 min.

25.3 Performance Evaluation

Apart from precision, the second master criterion is performance. For an analysis to be useful, it should be runnable in moderate time and with reasonable resources. We cannot expect the user to have the goodness to wait for weeks for the analysis of a not-so-big program, or to use a supercomputer. The only way to measure performance is to test on realistic software. Indeed, some approaches may have a satisfactory theoretical cost while being way too long in practice, and on the contrary, some algorithms with prohibitive complexity may perform nicely on real-world problems, maybe up to some heuristics. Testing on arbitrary software is not representative. It is indeed easy to write unrealistic source code for which the analysis behaves much worse than in real world cases. For instance, on the new framework we try to evaluate, it is quite easy to undermine the heuristic of ghost variable deletion: for now they are deleted as soon as they become useless, thus it is easy to force domain to spend a lot of time allocating and killing variables. If we kept useless ghost variables, it would be easy to trick them into creating huge, unmanageable supports. In-between solutions can also be trapped[‡‡]. Moreover, even without considering adversarial examples, if evaluation cases are not big enough cost linked to front-end (like parsing, typing, translations...) are predominant or, at least, non-negligible. Snippets of code written as non-regression and functional tests are in this category and are thus unsuitable for performance evaluation.

[††]Sizes of C call frames are not supposed to be known.

[‡‡]The Bélády's anomaly is a good example of how we can make this kind of algorithm arbitrarily bad [FI10], though we are aware the question is slightly different, this is quite an ill omen for our case.

25.3.1 Results

Unsurprisingly, the new framework is slower than the old one (on code that both succeed to analyze). We will not give raw results as they are impossible to interpret without context, and even with context, it is very difficult and heuristic. Moreover, the general behavior is quite clear and there are very few variations: it is not uncommon to get twice the same timing to the second, for a several minutes analysis. As we do not try to outperform the current state, but understand why it is slower, a general idea is sufficient. In the following, we will denote T the time used for the analysis by old framework, that serve as a reference.

Our first experiment was quite discouraging, but mildly surprising: we get a timing around $70T$. Such a slow down is completely prohibitive for an industrial use: a formerly 1-minute analysis takes now more than an hour. A few obvious fixes were applied. First, disabling sanity checks allowed saving most of the time: we went down to $6T$ by just flipping a few Boolean variables. We went even further in this way, be deactivating at compile time sanity checks and debug printed rather than using a Boolean variable. Even with immutable variables, OCaml compiler seemed not to manage to remove never-satisfied tests. We saved some time and went down to $5T$.

The next round of optimizations were less obvious and more specific to this software: it is about roles of ghost variables. If ghost variables have roles, some algorithms are more straightforward, and some precautions are useless. For instance, support unification become much simpler, and sanitation of directed ghost constrains is useless, as long as real assignment are indeed sanitized. Since we only use ghost variables with roles, it would be a shame not to benefit from such optimizations. To do so, there are two solutions: we can dynamically detect variables without role or ask domains at instantiation whether they might create such ghost variables. The latter is faster, as it has a constant (and very small) cost, but the former allows optimizations to be used as much as possible, even if domains may allocate a ghost variable without role in the future. A hybrid approach was chosen, so as to minimize the impact in the code, and not to enforce choices we could regret. To ask if domains may allocate variable without role, we use capabilities (see section 24.7 “Capabilities” (page 308)). These modifications improved running time down to $3T$.

From then, there were no simple optimizations with non-negligible outcomes: the current slowdown is still 3. Only precise investigation can allow us to hope for further optimizations.

25.3.2 Investigation

To profile time usage, we used an internal profiling tool, so it can be aware of particularities of the code we are working on, is easier to control and provide more relevant statistics. In particular, we rejected the idea of a global instrumentation since it has been proven to have a major impact on the relative distribution of processing time: adding a constant overhead is prejudicial to very fast functions called many times. Moreover, we are only interested in effective computation time, since it is obviously the bottleneck:

there are neither insane memory usage (requiring slow swapping) or other wait on I/O.

As we are performing the same analysis, the exact same transfer functions are called with the same arguments. This simplifies the investigation: we shall find which transfer functions are more expensive than before, by how much, and then find why. It may sound trivial, but this is already a very low-level basis to start investigation: we can ignore anything from the interpreter to the struct domain.

However, while it is simple to find which abstract transfer functions are involved in the slow down, since the internal working may have very few in common with the old implementation, it might be difficult to know what could be done faster.

Given that basis, finding inefficiency sources is not conceptually difficult but is a thankless job, requiring method and patience, with a heuristic part to try to compensate the time spent by using the profiler, which is not negligible for frequently called functions.

25.3.3 Explanation

It appears that there are two major sources of slowdown. The most obvious (and not the biggest) is linked to the increased number of variables. This is part of the big trade-off performance vs. precision. As it is not a net loss (we trade it for precision), and it is not the worse, we judge it to be an acceptable cost.

The other source of inefficiency is binary operations, and more precisely, the preparation. We recall that there are two types of preparation: support unification for operations that return a new abstract state, like union, intersection, widening and narrowing (see section 20.5 “Binary Operations” (page 271) and section 24.4 “Driver and Combinator” (page 303)), and collection of comparable keys for inclusion or equality checks (see section 24.5 “Abstract Order” (page 306)).

Support unification may be inefficient when at each iteration of a loop, we allocate and delete the same ghost variables: allocation may be required by the body of a loop, but the union (or widening) at the top of the loop yield a much less precise abstract state, where the ghost variable is not useful anymore, and thus deleted. At the next iteration, the variable is created again, then deleted, and so on. Doing so lead to an awful waste of time. A solution to this problem is suggested in section 27.8 “Support Heuristics” (page 339). Simply said, the point is to keep useless variables when we believe we will need them later.

The collection of keys to compare is, by itself, quite fast, but implies a subsequent cost. We explained earlier that not all ghost variables are to be compared to check inclusion. This is usually a nice thing, since it allows domains to do less work: they compare only variables explicitly specified. But numerical domains do not work that way. They have not been written for the new framework, but not to lose the quite important collection of domains, we wrote an adapter that promotes a simple domain to a domain able to handle ghost variables (see subsection 24.6.2 “Numerical Domain Adapter” (page 308)). Since simple domains were not made to receive a subset of variables to actually compare, we need to delete all other variables (variables to ignore): the operation that should be faster becomes even slower. Fixes are exposed in section 27.7 “Adapting Numerical Domains” (page 338). The point is to make them to take a set of variables to compare while keeping compatibility, and using another adapter when domains are difficult to adapt. This is not conceptually difficult, the main problem is a software engineering one, to limit the amount of work, and still being compatible with the old framework. Nevertheless, it takes a lot of time to adapt all numerical domains.

Part V

Perspectives and Conclusion

Chapter 26

Proving Low-Level Properties

Analyzing mixed C and assembly allowed us to prove some properties on segmentation, but otherwise, we only proved the absence of RTEs in analyzed sections. Handling mixed code is already quite a challenge, and we did not have time to prove other low-level properties.

Here, we will see two points that we need to handle properly for our target application. The first is paging: proving that paging is correct would be a huge step to memory protection. The second is a hack more specific to our application, though we believe it might be useful in many other codebases: it is accesses to a deeper call frame.

26.1 Paging

Paging is explained in subsection 2.3.3 "Paging" (page 25). It is an important feature of memory management. Unlike segmentation, paging is a very dynamic and shared between all processes.

In real world strategies of segmentation, the problem could be simplified since we use segments starting at 0 and spanning over the whole space (or almost). The address translation is thus the identity, which can be safely ignored: we just check the set up segments imply a trivial translation, then there is no need to distinguish the logical address space and the linear address space.

Paging does not use identity mapping in general, but when it does, this is for a very good reason (see subsubsection 2.3.3.3 "Identity Paging" (page 27)). Thus, we need to distinguish the linear address space and the physical address space. In fact, most variables are only in one of both spaces: variables used only before activation of paging can be represented only in physical address space. Variables used only after activation of paging can be represented only in linear address space. There is a problem for variables that are alive during the activation of paging.

Let us break down this problem into more elementary ones. First, we shall prove that variables alive when paging is enabled are in a portion of memory that is identity mapped. This is not a formal requirement of the processor, but this is the usual solution since it might be awfully complicated otherwise. To do so, we need to know the address of these variables, but also the addresses of functions. This requires to get external information usually contained in linker scripts. The most pleasant solution would be to be able to interpret these scripts, but another solution, that already exists in Astrée, is to write annotations on variables and function declarations. Moreover, we need to prove that pages containing all these addresses are identity mapped. Formally, let $a \in [\![0, 2^{32} - 1]\!]$ be the 32-bit address of a variable. We need to prove that

$$\left(* \left(* \left(\mathrm{CR3} + 4 \times a\,[22:31]\right)[12:32] \times 2^{12} + 4 \times a\,[12:21]\right)\right)[12:31] = a\,[12:31]$$

where CR3 is the page directory base register (see subsubsection 2.3.3.1 “Translation Mechanism” (page 25)). Of course, we need an extra property: accessed entries are marked as present (P flag is set). That is, the $a\,[22:31]^{\text{th}}$ entry of the page directory points to a page table whose $a\,[12:21]^{\text{th}}$ entry points the page containing a.

This is what we need to do when activating paging, but it is not enough to prove that paging works correctly: we also have to check that pages do not designate the same physical memory (except explicitly specified, for shared memory). This is more difficult since each process has its own page directory, but only the currently executing process has loaded its page directory address in CR3: we need to keep track of page directory of suspended processes.

Once again, from a theoretical point of view, suspended processes are not a distinct feature: given the state of a machine, there is no general way to identify suspended processes without knowing conventions and strategies used by the system. The only interesting criterion is that the memory of a process should not be changed while it is asleep, keeping track of page directories is an overapproximation that allows replacing a complicated security property (expressible only on set of traces), by a safety property.

To find page directories, it might be enough to remember which values have been in CR3 register. Though, it does not take into account that processes can be terminated, and thus its pages can be reused. A solution might be to consider pages as unused, until proven otherwise. A page that belongs to another process, currently suspended or terminated, can be given to the current process. The error will be emitted the next time the suspended process is scheduled again, that is, when it loads its page directory base address into CR3: this is the proof that it was not terminated and that the page should not have been given to another process.

This method does not need annotation or heuristic to guess where are suspended processes and when they are terminated or just asleep.

At last, there is the problem of abstracting paging structures: page directories and page tables. They form almost a forest from page directory bases addresses to pages, but some converging edges may appear when sharing memory. Abstracting such structure will probably require shape analysis methods.

Overall, abstracting paging is a huge work. There are plenty of difficulties, and we probably just skimmed the surface of the underwater peak of problems. Likewise, solutions that we have mentioned are probably a very small part of the work. Also, a possible future implementation in Astrée would be one of the biggest changes. Nevertheless, it is almost sure that the new framework for domain cooperation with ghost variables and partial support of assembly will be helpful.

26.2 Access Across Call Frames

In our target application, in some parts of the code, there are unsupported operations. Though there are few of them, they prevent to analyze these parts. One of these unsupported operations are access to a deeper call frame in the stack. Handling such accesses would allow us to analyze crucial parts of the code.

In our handling of mixed C and assembly, we only allow assembly blocks to access global and current local variables. However, sometimes, assembly is used to dig deeper into the stack, and to read and write local variables of a caller (not necessarily the closest). Our abstraction of call frames indeed reject such accesses (see section 15.2 "Abstracting Successions of Call Frames" (page 188)) but it would be useful to relax this restriction.

The solution seems to be conceptually easy, but difficult to implement. Moreover, we need much more hypotheses beyond the C and assembly standards or the calling convention in use: we need to know how the compiler manages call frames. The goal is to be able to predict the size of each call frame, so that we know how the offset needed to skip them. This is sufficient to handle access to an assembly stack segment, but if we try to access a C local variable, we also need to know where local variables are inside each call frame. These are very strong hypotheses on the compiler but there are not superfluous: without them, a code that accesses deep call frames is illegal. When we use such hypotheses, the analysis becomes specific to the compiler, and a priori, it would not work with another compiler. This is not a weakness of the analysis: it is the same in the concrete world: compiled with a compiler that manages call frame in another way, such a code will not work. Short, we cannot expect an analysis to be successful with fewer hypotheses than we made when writing the code, and thus, hypotheses we need to compile it.

Though the idea is simple, there are a few difficulties to implement it properly. First, from a very technical point of view, as ASTRÉE was designed to analyze C, it assumes no relative position of variables, even local ones. Moreover, it is not clear how to encode these hypotheses. They are very specific to a given compiler, and thus, they should not be hard coded in ASTRÉE. Rather, it would be better to write them in configuration files interpreted by ASTRÉE. Such files should contain hypotheses related to call frame management in a general enough format. Then, we would be able to write the call frame management policy of any (reasonable) compiler, or at least a relevant fragment, to test if the code is still correct when compiled with these hypotheses.

Chapter 27

Improving Product with Ghost Variables

New domain cooperation framework is already a big change that provides a great expressive power, but it still can be improved in many ways. Obviously, we may add new domains, but also we can improve the framework with new features (useful for new domains), and try to improve the concerning performance issue presented in section 25.3 "Performance Evaluation" (page 320).

We give some improvement possibilities, which are not light to implement or formalize. We start by idea about new features or domains, and we give after the suggestions to improve performance. Most of them are very general ideas that require much more work.

27.1 Abstracting a Different Concrete World

Current version of the product with ghost variables is designed to be very general with respect to the semantics of the language, but requires memory states to be maps from variables to values, without aliasing or dereference. This was useful to justify that right-hand side expression are not changed when assigning the left-hand side with directed constraints. If modifying the abstract state of the left-hand side impacts the value of the right-hand side, strange interference may happen.

Nevertheless, most languages, especially C, allow pointers and dereference, even if sometimes it is only implicit (most high-level languages use pointers implicitly). It might be interesting to handle ghost variables in a world where aliasing exists. In the stack of domains of Astrée, that would mean to lift it before memory accesses are resolved into physical cells (above struct domain or at its place).

Since we cannot have a syntactical guarantee that assignments will not interfere, we might crave for a more subtle criterion: a semantic argument. It should be possible to adapt the ghost variable framework for any state that may be split into independent parts. For instance, it could be a way to implement cofibered domains [Ven96]. Typically, we could imagine using several domains of shape analysis, side to side: node of each domain can be a substructure of another one. With a domain that handles trees, and one that handles lists (with a domain to store numerical values), we could represent lists of trees as well as trees of lists, and any recursive combination. We rely on domains to prove that there is no harmful aliasing dynamically. We could still default to $\top$ if something goes wrong.

A similar architecture could be use at the supreme level: iterators. We could imagine an adaptation of the framework where iterators are at the same place as ordinary domains, and not above, as usual. The product is fed the whole program (for instance, as an AST); while most domains have nothing to do with directives more complicated than a comparison or an assignment, iterator-domains can interpret the program and translate each syntactical element into abstract directives sent to all domains. This may be interesting to implement gradually a language: each syntactic or semantic feature can be handled by a different domain. If there is no domain to provide constraints for a given feature, the default $\top$ value remains, which is correct. We may also raise an informative alarm to explain that a non-leaf node of the AST were not handled. Making each language feature handled by a different domain may allow simplifying the maintenance, but also to reuse elements common to several languages. From a theoretical point of view, we believe it requires not that many adaptations to formalize. Indeed, we could see iterator-domains as translator of constraints: constraints are initially written as a part of an AST (which is a constraint thanks to the language semantics), and the iterator generates an equation on abstract states. This is close to the intention of [JMO20], which simplify gradually the programs and expressions. The solution we propose do not enforce an ordering in the simplification; our hierarchy is completely flat.

Moreover, we are not limited to have a single domain for each syntactic feature. For instance, we may have several translators for loops. For range-based for-loops (`for in` loops), several translations are possible. In C++, such loops may be desugared using iterators, but if we have a higher level description of the data structure on which the loop iterates, we may generate inductive predicates that would work very well with a specialized domain, like a shape analysis domain. Even standard while-loops can be translated into internal directives naively using widening at loop's head, or using a finer widening strategy, or even unrolling. Correctness criterion is simple: each translation must preserve (or overapproximate) the semantics. But there are other problems, we name two.

– How to ensure termination when some translations can be applied arbitrarily many times? For instance, loop unrolling generates new unrolled statements, but a modified loop (for non-unrolled iterations) is also emitted. It is still possible to unroll this new loop.
– How to choose the next translation when there are multiple possibilities?

For the first problem, we can imagine there are two (or arbitrarily many) instances of loop-unrolling domain. Indeed, a single domain can be smart enough not to unroll its own result. A way to ensure termination is to annotate each statement[*] with its history.

[*]Here, it can be a complex compound statement, like a whole function body, a simple statement, or even an abstract statement involving abstract transfer functions.

In this case, the history is a tuple where each component is managed by a domain: each time the domain transforms a statement, new statements inherits the parent history, but where the component corresponding to the current domain is modified. Each component should follow a well-founded order defined by each domain, which ensures termination. In the case of loop unrolling, the component can be an integer representing the "fuel" of the domain: how many times it still allowed to unroll the loop. Another interesting example is the (rather trivial) domain that transforms a sequence of statements into the composition of the semantics of each element of the sequence. Since syntactic construct (such as branching or loops) may be arbitrarily nested, the sequence-domain can be called arbitrarily many times to obtain elementary statements. A solution is to set its fuel to the depth of the AST, and each time it splits a sequence, the fuel is decremented in newly generated statements. The fuel can never be negative, ensuring termination. Moreover, if the AST contains non-sequence nodes on each path to the deepest leaves, the fuel will not even reach 0.

The question of choosing the transformation is more heuristic. Each translation is sound and a single translation can be precise enough. To favor one over another, we may rely on heuristics to guess which one will be the better one. But we may still be wrong, and performing the analysis again, but with another translation will give another sound result. The intersection of both results will be more precise and still sound, but it might be unnecessarily long. An idea could be to annotate the analyzed program with important points for each domain: domains can say where they can be useful (the loop node, for loop-related domains) and where we will get the result (after the loop). If we choose a translation and the remaining of the analysis succeed, we do not need additional work. But if it fails, a backward analysis (such as a causal analysis) may guess that the loop's postcondition is too imprecise, and try to improve it using another transformation and reanalyzing the loop. Hopefully, it will not be necessary often, since we believe that in many cases, the right translation is not difficult to guess (depending on the traversed structure, the language, the kind of the loop...).

Of course, these are only general ideas, and an actual implementation would probably raise many other problems. However, we trust this is a possible solution to build a supremely modular analyzer supporting many languages. The question of performance is probably the biggest problem: such a solution requires much more machinery than the architecture of ASTRÉE, and it would probably have a large impact on analysis time. On the other hand, a flat architecture offers other optimization opportunities, e.g. it makes distributed computations much simpler. Slow domains may instantiated many times and the workload distributed.

27.2 Ghost Variable Renaming

Utility of handling linear combination of ghost variables has been explained in subsection 18.1.3 "Linear Combinations of Pointers" (page 234). Yet, as shown in section 23.3 "Linear Combinations" (page 295), ghost variables representing terms of linear combination are not meaningfully ordered. A smart ordering can help to preserve as much precision as possible. Though, it is only possible to find the good ordering once we have both states, to compare how to lose the least precision. Then, we need to rename variables so that the following binary operation will be as precise as possible.

Exchanging roles of variables requires variables to have a symmetric semantics, so that, when roles are swapped, the concretization do not change. To do so, we would need an internal operation to change variable roles. This can be made using a temporary variables and assignments, but it would be faster as a simple relabeling. Such feature would allow implementing a smart linear combination domain.

We could generalize this feature to any rewriting rule that describes a sound transformation. Such a case will be exposed in a following section.

27.3 Partial Partitioning

Partitioning is a general and systematic strategy to improve precision, though it is usually not clever and may be expensive. Indeed, most of the time, partitioning is only useful for a few variables, but we duplicate the abstract state in memory. One may argue that persistent data structures allow common substructures to be shared, but it is often a negligible improvement for long-lived objects. Indeed, maps (since they are the common way to store the variable-to-values correspondence) are balanced and need sometime to be rearranged. When it happens, a lot of shared substructures are broken into smaller shareable substructures, at the end, the benefit may not be clear.

Ghost variables may provide a nice solution by implementation partitioning as a part of the abstract state, and not above. Given a partitioning condition c and a variable x, we create variables $\mathscr{T}\text{rue}_c(x)$ and $\mathscr{F}\text{alse}_c(x)$ to store values of x in cases where c is true and false respectively. This allows the analyzer to partition only variables we are interested in.

For instance, initially, we may only partition variables that appear in condition c. After each assignment, the left-hand side variable should be partitioned if there is at least a partitioned variable in the right-hand side. This is the most comprehensive partitioning since other variables are not impacted by the value of c.

Smarter heuristics can be designed depending on the application. We can also implement many subtle improvements. For instance, delete ghost variables if they are equal, or if one of them is strictly greater than the other, deleting the greatest one will not lose precision, since x should be equal to this variable. It may also be useful to end partitioning through annotations: we can manually reduce the lifetime of ghost variables, improving performances.

$$\begin{tikzcd}
 & & \text{x} \arrow[dll, dash] \arrow[dl, dash] \arrow[dr, dash] \arrow[drr, dash] & & \\
\mathscr{F}\text{alse}_c(x) & \mathscr{T}\text{rue}_c(x) & & \mathscr{F}\text{alse}_d(x) & \mathscr{T}\text{rue}_d(x)
\end{tikzcd}$$

Figure 27.1: Flat partitioning

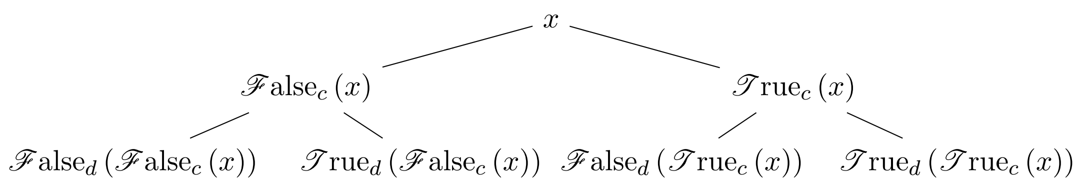

x
$\mathscr{F}\text{alse}_c\,(x)$
$\mathscr{T}\text{rue}_c\,(x)$
$\mathscr{F}\text{alse}_d\,(\mathscr{F}\text{alse}_c\,(x))$
$\mathscr{T}\text{rue}_d\,(\mathscr{F}\text{alse}_c\,(x))$
$\mathscr{F}\text{alse}_d\,(\mathscr{T}\text{rue}_c\,(x))$
$\mathscr{T}\text{rue}_d\,(\mathscr{T}\text{rue}_c\,(x))$

Figure 27.2: Nested partitioning

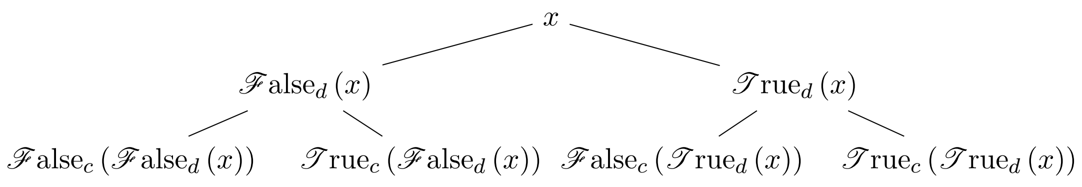

x
$\mathscr{F}\text{alse}_d\,(x)$
$\mathscr{T}\text{rue}_d\,(x)$
$\mathscr{F}\text{alse}_c\,(\mathscr{F}\text{alse}_d\,(x))$
$\mathscr{T}\text{rue}_c\,(\mathscr{F}\text{alse}_d\,(x))$
$\mathscr{F}\text{alse}_c\,(\mathscr{T}\text{rue}_d\,(x))$
$\mathscr{T}\text{rue}_c\,(\mathscr{T}\text{rue}_d\,(x))$

Figure 27.3: Inverted nested partitioning

Enhanced rewriting rules can be useful when we partition according to several conditions. Let c and d two conditions and x a variable. Let us assume that we partition x with respect to c and d. There are several possibilities: partition could be flat, as depicted on Figure 27.1, or nested, as on Figure 27.2. Flat partitioning avoid the following problem we are about to explain, but it has less precision, thus it cannot be used in every case, and sometimes, nested partitioning must be used. We assume the current state is as shown on Figure 27.2, but, e.g. thanks to an annotation, we want to end the partitioning with respect to d; this is very easy: we just need to delete variables on the leaves. If we want to end partitioning with respect to c instead, we need to transform the tree to obtain the tree Figure 27.3. This can be done using rules

$$\begin{gathered}
\forall(c,d,x), \mathscr{F}\text{alse}_d\,(\mathscr{F}\text{alse}_c\,(x)) \to \mathscr{F}\text{alse}_c\,(\mathscr{F}\text{alse}_d\,(x))\\
\forall(c,d,x), \mathscr{T}\text{rue}_d\,(\mathscr{F}\text{alse}_c\,(x)) \to \mathscr{F}\text{alse}_c\,(\mathscr{T}\text{rue}_d\,(x))\\
\forall(c,d,x), \mathscr{F}\text{alse}_d\,(\mathscr{T}\text{rue}_c\,(x)) \to \mathscr{T}\text{rue}_c\,(\mathscr{F}\text{alse}_d\,(x))\\
\forall(c,d,x), \mathscr{T}\text{rue}_d\,(\mathscr{T}\text{rue}_c\,(x)) \to \mathscr{T}\text{rue}_c\,(\mathscr{T}\text{rue}_d\,(x))
\end{gathered}$$

which hold since each rule involves two variables that abstract the same concrete state, but we also need additional computations to create $\mathscr{F}\text{alse}_d\,(x)$ and $\mathscr{T}\text{rue}_d\,(x)$. Typically,

$$\begin{gathered}
\mathscr{F}\text{alse}_d\,(\mathscr{F}\text{alse}_c\,(x)) \sqcup \mathscr{F}\text{alse}_d\,(\mathscr{T}\text{rue}_c\,(x)) \to \mathscr{F}\text{alse}_d\,(x)\\
\mathscr{T}\text{rue}_d\,(\mathscr{F}\text{alse}_c\,(x)) \sqcup \mathscr{T}\text{rue}_d\,(\mathscr{T}\text{rue}_c\,(x)) \to \mathscr{T}\text{rue}_d\,(x)
\end{gathered}$$

This can also be expressed as a single rule as tree rewriting. From the new tree, we can easily end the partitioning with respect to c. We can remark that ending a partitioning can also be expressed as a sound tree rewriting rule. Both rules can be composed to get a single, more efficient rule.

This partitioning domain is limited as it can only work at the same level as the new framework, but it could be an optimization by limiting the size of the abstract states in memory and the number of abstract transfer functions to call. We shall also take care not to partition ghost variables under a partitioned variable; this requires domains to be able to look in the stack of roles.

27.4 Sparse Support

Thanks to the previous domain, we might restrict the number of partitioned variables, improving the performance. But each partitioned variable need the original one x and two ghost variables $\mathscr{T}\mathrm{rue}_c\,(x)$ and $\mathscr{F}\mathrm{alse}_c\,(x)$, while simply duplicating the state needs only 2 variables. When we partition many variables, this may be slower than the basic implementation.

A solution might be to remove unnecessary variables. Values of some variables may entirely be determined by its ghost variables only. This is typically the case of partitioning: the value of x can be found from the value of $\mathscr{T}\mathrm{rue}_c\,(x)$ and $\mathscr{F}\mathrm{alse}_c\,(x)$. This case is very easy as it requires a simple abstract join.

Even if x is required in the enriched concrete semantics, it can be missing in the abstract state. We need to adapt the concretization to generate all variables above a ghost variable, even if it is missing. Missing variables can be rebuilt on demand. One should find a good heuristic so that the deletion and reconstruction of variables do not occur too frequently to ruin the potential performance gain. This can only be designed from a real use case.

27.5 Activation on Demand

Since the new framework is slower, if the analysis only uses domains that existed in the old framework, it would be foolish not to use it. In many cases, slices domain is absolutely necessary, but on short sections of code. Based on this observation, we could use the old framework by default, until we reach a point where additional domains with ghost variables are needed. There, the state is exported to a state of the new framework and the analysis can carry on. Once the new framework completed its deed, we perform the reciprocal translation; we can lose information, but nothing still useful.

Adopting such a strategy doubles the maintenance cost as both the new and the old framework are still in use. Moreover, abstract state translation is probably very expensive: it may only be suitable for code where there are few translations to do, and where sections that may be analyzed with the old framework are big enough.

This idea is mentioned to get a quite systematic list of solutions, based on the criterion we may be able or willing to relax, but it has very little chance to indeed work. Moreover, due to the maintenance cost, it may not be desirable anyway. A nicer approach would be to activate domains on demand, while using only the new framework. Sadly, automatically detecting if a domain is required is not that different to actually use it. We could also rely on annotations, but that would mean to lose automation.

27.6 Parallel Computations

When we cannot improve the algorithm or the implementation, a classical strategy is to try to make it parallel. Let us explore how it works in our case. This implies two questions: how to divide the work into parallel tasks and how to make these tasks thread-safe.

The first, simplest, strategy is to parallelize computations from a higher level, for instance, from the partitioning domain (see Figure 14.1 “New structure of the composite domain of ASTRÉE” (page 179)). This is easier as we do not need to modify the way the new framework operates, but it still needs a few adaptations. Most of the product with ghost variables is thread-safe as we use mainly immutable data types and pure functions. But, there are few mutable values that should be taken into account. First, we can mention maps used in memoized functions. These functions are thread-local and thus, their hidden maps are thread-local as well. The other mutable data is the state of the `key dispenser`. It is explained in section 24.2 “Roles” (page 299) why it needs a global mutable state. For these reasons, we understand that the state should in fact be globally shared across all threads. This implies the use of locks. This basic exclusion method would probably not impact the performance in a great extent since modification to the state of the `key dispenser` are quite rare in average, and very fast. This kind of high-level parallelism is already implemented in ASTRÉE (before the work presented here) but cannot be used with the new framework for the reason explained. If parallelization is eventually supported, we would need to adapt the logger as well: it should not write two lines at the same time, and each instance must identify its thread. Moreover, it would be nice to keep lines about a single abstract directive consecutive. Each logger can use a separate file or use a cache to delay lines until the computation of the current statement is finished and print everything at once. This is more readable but produce no output when a computation does not terminate.

We may also wonder if we can introduce parallel computations in the new framework itself. As a rule of thumb, since we used synchronization to perform more reduction, we cannot expect that much parallelism without having a prohibitive cost in inter-thread communication. A possible improvement would be to make each domain run in a separate thread. It would require to adapt the iteration strategy, in particular, all domains are run at once, and they may all yield constraint-DAGs. A simple implementation wait the slowest domain to complete before running each DAG (and so on, recursively). A more complex implementation may deliver constraint-DAGs asynchronously. In both cases, the possible performance gain is not clear, but it may be worth to try when working with a high number of domains.

Another, possibility is to parallelize only the computation of parallel edges in constraint-DAGs. This would be much easier to implement, but it would require extremely long parallel branches (or a lot of recursively collected constraint-DAGs) to compensate for thread creation and communication time. Once again, this is a very heuristic estimate, and implementation and testing is the only way to have an idea of the real efficiency. Though, it might be heavy to implement for a discouraging result, we do not recommend it as being the first option. Especially, following sections give ideas that would almost surely be not disappointing.

27.7 Adapting Numerical Domains

It was said in subsection 25.3.3 "Explanation" (page 322) that a major source of inefficiency is the preparation of comparisons in numerical domains. Indeed, the adapter does the best it can, but since numerical domains are unable to receive a set of variables to compare, the adapter must remove all other variables.

The solution is simple: we have to adapt all numerical domains to be able to receive a set of variables to compare, and ignore the other ones. In most domain, the transformation is easy. For instance, the composite non-relational domain stores the state of each variable in a map from variable keys to abstract values; for these domains, we shall simply iterate over the set of the variables to compare rather than the whole map. Many relational domains would require very few transformations as well. Yet, some domains, especially those that are implemented in C or packed domains, are more difficult to adapt. A temporary solution may be to apply the adapter only on domains that are difficult to adapt. There would be several instances of the adapter, tied together by the combinator of domain with ghost variables.

The performance gain is almost sure, but there are several drawbacks. First, adapted domains would not be able to work in the old framework, which is for now the faster and more reliable one. Moreover, apart from the comparison functions, there are many other functions to adapt to get the right signature, even without using ghost variables. Finally, using the combinator of domains with ghost variables is less efficient than the combinator of numerical domains that does not take care of ghost variables; since we know this subtree of domains does not use ghost variables, the old combinator is sufficient.

We believe there is a common solution to all these problems. Firstly, we adapt numerical domain by *adding* a new function that allows partial comparison, given a set of variables. This way, the traditional signature of numerical domains is preserved. The old combinator of numerical domains needs the same adaptation, so that we get a composite domain compatible with the old framework, with an additional function for partial comparison. We can write a small adapter for domains for which it is difficult to implement partial comparison. It will work in the same way as the current adapter, by deleting variable we do not want to compare. This small adapter only generate the partial comparison function and can be used on each domain that needs it, and not at the root of the tree of domains. Finally, we change the current adapter (the one at the root) to use this new partial comparison function instead, so that it does not need to remove extra variables anymore.

This solution is probably the best since ghost variable management is entirely performed by the adapter. Underlying combinators are good old numerical domain combinators, which do not work with ghost variables, and thus are more efficient. Domains are still compatible with the old framework, and the change is minimal. This solution has not been implemented because of time constraint and the number of domains difficult to adapt; while there are relatively few of them, the absolute number is quite important, since ASTRÉE has many domains.

There is another way in which old domains may be adapted that allows to improve the precision of other domains. During support unification, one may need to add an unused ghost variable in an operand if it exists in the other (for instance, see section 21.3 "Adaptation in the New Framework" (page 277)). For a join, since this newly created variable is unused, it would be extremely precise to assign it the value $\perp$: during the join, only the value of the existing variable is kept. Though, as mentioned in section 21.3 "Adaptation in the New Framework" (page 277), numerical domains implement $\perp$-coalescence: when a variable is $\perp$, the whole property is $\perp$. This used to be relevant when there were only real variables, but this is not true with ghost variables: if an unused ghost variable gets the value $\perp$, the concretization of the state is not necessarily $\perp$. For now, we store in the new ghost variable the value of the existing one in the other property; while being optimal for non-relational properties, it destroys relations that involve this ghost variable. Adding a mechanism to prevent unwanted $\perp$-coalescence, would allow assigning $\perp$ to unused ghost variables and, hopefully, keep relations. A solution may be to have a Boolean value for each ghost variable that tells whether it is useful with respect to its immediate parent. When $\perp$ appears in a used variable, it propagates to the parent, and recursively until we reach a real variable or an unused variable. If we reach a real variable, $\perp$-coalescence indeed applies. This "is used" status is not the same as the reference counting used for ghost variable collection (see section 24.3 "Translator" (page 301)). Indeed, a freshly added variable is not used in a relation (preventing $\perp$-coalescence), but is still useful for the domain (not to be deleted). Using the same counter for both notions ("ununsed" and "to be deleted") would imply to delay variable deletion to specific steps, in case a variable may be in fact useful; this prevents deleting actually unused variables between support unification and the following binary operation, whereas variable deletion is an essential part of support unification. Overall, it is better to allow domains to allocate and keep unused variables. Keeping unused variables is also the key idea of the optimization explained in the next section.

27.8 Support Heuristics

Another major source of inefficiency is support unification in preparation for binary operations. Many variables are added and removed, though most of them are not new. Indeed, we can check that the `key dispenser` already provided a key for these roles. In practice, they are mainly variables that exist after the analysis of the body of a loop, but not in the cumulative state at the top of this loop.

Indeed, variables are added by precise computations, then removed when merging with a less precise state. This process is very expensive. A solution is not to remove ghost variables, even if they are unused. This way, we would only add new variables, which will likely converge to a stable support. Yet, this method is not advisable since it would result in huge supports with important memory usage. We must be able to remove variables, but not too soon.

Once again, there is a world of strategies, and once again, there are two main possibilities: user-written annotations or automatic heuristics. Annotations transfer the responsibility of the problem to the user, which makes the analyzer even more difficult to use. Of course, as good as heuristics can be, we can still write code where they fail. That's why we always advocate to use both methods: smart heuristics allow the everyday user to make most analyses successful, and annotations allow the expert user to guide the analyzer when heuristics fail. An even more powerful solution is to be able to describe heuristics in a configuration file using a domain-specific language (DSL). This allows the expert user to automate its work, but within the limitations of the DSL.

We will explain a heuristic that seems both reasonable and efficient. Repeated union, widening and other binary operations happen mainly in loops, as explained before. Of course, support unification is also performed at the end of if-then-else (see section 20.5 "Binary Operations" (page 271)), but only once. The proposed heuristic is to simply not remove unused variables while analysis the loop, and remove unused ghost variables only after the loop analysis. Of course, as any heuristic, its efficiency can only be proven by actual experiments. Real world applications can also help to design new and more adequate heuristics.

Chapter 28

Conclusion

In this thesis we examined the problems raised by the analysis of mixed C and assembly code, motivated by the formal verification of an industrial study case, a host platform, with ASTRÉE. We are interested in the absence of run-time errors, and some security properties (like memory isolation) that are overapproximated by low-level safety properties. We saw there are mainly two parts.

The first problem is the interaction of C and assembly, though they use very different memory model and control flow structures. But before being able to analyze such code, we need to define its semantics. We provided simplified versions of C and assembly, to build a model of mixed C and assembly code. The main point is the memory model that allows C and assembly operations to work correctly together. This semantics requires some hypotheses that are not part of C standard, but which are necessary to reason on mixed code, otherwise, we cannot conclude anything interesting. Parts of our study case need even more hypotheses and a more precise memory model, but we have ignored these parts for now, as the assumptions involved are very specific to this case (and compiler-depend), and are only useful for a few artful, but essential, snippets.

Once the semantics is set up, we can use any method to analyze programs. Since we focused on abstract interpretation, we showed some abstractions that are sound and are precise enough for our study case. The domains that we designed depend on the software we analyze, and even when a program is correct with the semantics we have defined, domains may be not precise enough, and wrongfully find errors. Though, we explained why we believe that the abstractions that we presented are both efficient and precise enough in many cases. Another benefit is that our abstractions are parametric: they abstract an aspect of assembly (e.g. registers) using more generic abstractions (e.g. integer domains). This allows the developer to fully reuse existing domains, and improve simultaneously general and assembly-specific abstractions.

The second problem is not directly related to the presence of assembly, but comes from what we do when assembly is involved. Indeed, if assembly parts could be done in C, we would not bother writing mixed code. Assembly blocks exist for specific purposes (like using CPU low-level features), that imply operations that are pointless in pure C. We detailed two examples: chopping bit slices of values (including pointers) and rebuilding them (typically to fill a GDT, or manage paging), and computing linear combinations of values (e.g. for dynamic code). We saw that these problems could be solved easily using ghost variables, but we also explained why it is not easy to completely use the power of ghost variables in a reduced product of abstract domains. In this thesis we proposed a framework that solves this problem. It allows domains to cooperate (that is, to perform partial reduction), and share ghost variables in a very decentralized manner. With this framework, new domains using ghost variables can be added without modifying anything in existing domains. Conversely, new domains can also improve the precision of existing domains, since the precision of each domain depends on the precision of communication channels. We can develop a rich collection of abstract domains and use only the required subset for each problem to get optimal performance. Thanks to adapters, we are able to reuse existing domains that does not handle ghost variables, which allows the new framework to be at least as precise as the former one.

All these solutions have been implemented in AstréeS, a development version of Astrée, showing that they achieve required precision on our industrial study case, and we trust they are precise enough to be used on many reasonable source codes. Moreover, when a better precision is needed, it might be sufficient to improve the underlying abstractions, without changing the parametric abstractions that are specific to assembly handling. In terms of performance, these solutions are satisfactory and can be used on real-life software, though there is a clear, but not prohibitive, slowdown associated with the product of domains with ghost variables.

Of course, there are some possible improvements. On the first problem, we mentioned that sometimes, additional hypotheses are ensured and required. If we want to take them into account, we need to make the semantics more precise accordingly and find more precise abstractions. Abstract domains can also be improved, like any domain, but it is likely to be a problem not limited to the assembly. For instance, a proof of paging would need an adequate memory model to represent the structures involved (page directories and page tables).

Many improvements are still possible on the product of domains with ghost variables. First, we can always add new domains, according to the needs. The modularity is very satisfactory, and for now, it provides much more features that are used by domains (like ghost variables without roles). Nevertheless, some domains can benefit from even more advanced features (like role rewriting for partitioning or linear combinations). We also explained briefly how this product of domains could be lifted to build a whole analyzer with a supremely flat, thus very modular, architecture. Currently, the main problem of the product of domains with ghost variables is the execution time: while the slowdown may be considered an acceptable trade-off for improved precision and expressive power, it is still significant when the old framework is good enough. We gave several ideas to speed it up; some are internal (like better adaptation of old domains), but other options imply transferring some tasks to the new framework when it can perform them more efficiently, improving global execution time (like ghost variables-based partitioning). We think these optimizations may reduce the slowdown to an acceptable value.

Our ultimate motivation is the analysis of our study case, which is an operating system. The analysis of OSes is very complicated, but necessary since they are a fundamental layer in many software systems. We expect that OSes match very specific properties, like memory protection (which was studied in this thesis) or fairness of the scheduler. Some questions, like the latter, are usually studied separately as they are rich enough *per se*. Specific analysis tools may eventually be integrated in a general analyzer like ASTRÉES, to make it able to check entire OSes at once.

Bibliography

[Abs20] [SOFTWARE] AbsInt, *StackAnalyzer* version 19.10, Apr. 27, 2020. URL: `https://www.absint.com/stackanalyzer/index.htm`.

[AČW09] Rajeev Alur, Pavol Černý, and Scott Weinstein. "Algorithmic Analysis of Array-Accessing Programs". In: *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*. 2009, pages 86–101. DOI: `10.1007/978-3-642-04027-6_9`. URL: `https://cernyp.github.io/publications/csl09/csl09.pdf`.

[Ama+16] Gianluca Amato et al. "Efficiently intertwining widening and narrowing". In: *Science of Computer Programming* 120 (May 2016), pages 1–24. ISSN: 01676423. DOI: `10.1016/j.scico.2015.12.005`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0167642315004165`.

[Bla+03] Bruno Blanchet et al. "A static analyzer for large safety-critical software". In: *ACM SIGPLAN Notices*. Volume 38. ACM, June 9, 2003, pages 196–207. ISBN: 1-58113-662-5. DOI: `10.1145/781131.781153`. URL: `https://hal.archives-ouvertes.fr/hal-00128135`.

[Bou92] François Bourdoncle. "Abstract interpretation by dynamic partitioning". In: *Journal of Functional Programming* 2.4 (Oct. 1992), pages 407–435. ISSN: 1469-7653. DOI: `10.1017/S0956796800000496`. URL: `https://www.cambridge.org/core/product/identifier/S0956796800000496/type/journal_article` (visited on 09/25/2019).

[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*. ACM, 1977, pages 238–252. DOI: `10.1145/512950.512973`. URL: `https://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml` (visited on 02/08/2017).

[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, San Antonio, Texas, USA, January 1979*. POPL. https://dblp.org/rec/bib/conf/popl/CousotC79. San Antonio, Texas, USA, Jan. 1979. DOI: `10.1145/567752.567778`. URL: `htt` ps://www.di.ens.fr/~cousot/COUSOTpapers/publications.www/CousotCousot-POPL-79-ACM-p269--282-1979.pdf (visited on 09/25/2019).

[CCF13] Agostino Cortesi, Giulia Costantini, and Pietro Ferrara. "A Survey on Product Operators in Abstract Interpretation". In: *Electronic Proceedings in Theoretical Computer Science* 129 (Sept. 19, 2013), pages 325–336. ISSN: 2075-2180. DOI: 10.4204/EPTCS.129.19. URL: https://arxiv.org/abs/1309.5146v1.

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. "A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: Association for Computing Machinery, Jan. 26, 2011, pages 105–118. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1925844.1926399. URL: https://hal.archives-ouvertes.fr/inria-00543874/.

[Čer03] Pavol Černý. "Verification by Abstract Interpretation of Parameterized Predicates". DEA Thesis. Paris, France: Ecole normale superieure (ENS), 2003.

[CF20] Marc Chevalier and Jérôme Feret. "Sharing Ghost Variables in a Collection of Abstract Domains". In: *Verification, Model Checking, and Abstract Interpretation*. VMCAI 2020. Edited by Dirk Beyer and Damien Zufferey. Volume 11990. Lecture Notes in Computer Science. New Orleans, LA, USA, Springer International Publishing, Jan. 2020, pages 158–179. ISBN: 978-3-030-39322-9. DOI: 10.1007/978-3-030-39322-9_8. URL: https://hal.archives-ouvertes.fr/hal-02378809.

[Che20a] [SOFTWARE] Marc Chevalier, *Epictetus* version 3.0.0, 2020. LIC: MIT. URL: https://github.com/marc-chevalier/epictetus(visited on 05/20/2020), VCS: https://github.com/marc-chevalier/epictetus.git.

[Che20b] [SOFTWARE] Marc Chevalier, *OColor* version 1.2.1, 2020. LIC: MIT. URL: https://github.com/marc-chevalier/ocolor(visited on 05/20/2020), VCS: https://github.com/marc-chevalier/ocolor.git.

[Che20c] [SOFTWARE] Marc Chevalier, *Plato* version 1.1.0, 2020. LIC: MIT. URL: https://github.com/marc-chevalier/plato(visited on 05/20/2020), VCS: https://github.com/marc-chevalier/plato.git.

[CIE19] CIE. *Colorimetry — Part 4: CIE 1976 L*a*b* colour space*. ISO/CIE standard 11664-4:2019(E). International Commission on Illumination (Commission internationale de l'éclairage), 2019. URL: http://cie.co.at/publications/colorimetry-part-4-cie-1976-lab-colour-space-1.

[CL05] Bor-Yuh Evan Chang and K. Rustan M. Leino. "Abstract Interpretation with Alien Expressions and Heap Structures". In: *Verification, Model Checking, and Abstract Interpretation.* Volume 3385. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pages 147–163. ISBN: 978-3-540-30579-8. DOI: `10.1007/978-3-540-30579-8_11`. URL: `https://link.springer.com/10.1007/978-3-540-30579-8_11`.

[Col98] Robert R. Collins. *The Pentium F00F Bug.* Dr. Dobb's. May 1, 1998. URL: `https://www.drdobbs.com/embedded-systems/the-pentium-f00f-bug/184410555` (visited on 07/06/2020).

[Coq20] [SOFTWARE] The Coq development team, *Coq* version 8.11.2, 2020. LIC: LGPL 2.1. URL: `https://coq.inria.fr/`, VCS: `https://github.com/coq/coq`.

[Cou+01] [SOFTWARE] Patrick Cousot et al., *Astrée*, 2001. URL: `https://www.astree.ens.fr/`.

[Cou+06a] [SOFTWARE] Patrick Cousot et al., *AstréeA*, 2006. URL: `http://www.astreea.ens.fr/`.

[Cou+06b] Patrick Cousot et al. "Combination of abstractions in the ASTRÉE static analyzer". In: *Annual Asian Computing Science Conference.* ASIAN. Tokyo, Japan: Springer, 2006, pages 272–300. ISBN: 978-3-540-77505-8. DOI: `10.1007/978-3-540-77505-8_23`. URL: `https://hal.archives-ouvertes.fr/inria-00528571/` (visited on 09/13/2016).

[Cou+20] Patrick Cousot et al. *The Astrée Static Analyzer.* CNRS/ENS/INRIA. 2020. URL: `https://www.astree.ens.fr/`.

[Cou78] Patrick Cousot. "Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes (Iterative methods for construction and approximation of fixpoints of monotone operators on lattices, program static analysis)." Thèse ès Sciences Mathématiques. Grenoble, France: Université Joseph Fourier, Mar. 21, 1978. 278 pages. URL: `https://www.di.ens.fr/~cousot/publications.www/CousotTheseEsSciences1978.pdf` (visited on 04/01/2020).

[Cou99] Patrick Cousot. "The Calculational Design of a Generic Abstract Interpreter". In: *Calculational System Design.* Edited by M. Broy and R. Steinbrüggen. 1999, page 88. URL: `https://www.di.ens.fr/~cousot/COUSOTpapers/Marktoberdorf98.shtml` (visited on 05/26/2020).

[Das+19] Sandeep Dasgupta et al. "A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019.* ACM. Phoenix, AZ, USA: Kathryn S. McKinley and Kathleen Fisher, June 22, 2019, page 16. ISBN: 978-1-4503-6712-7. DOI: `10.1145/3314221.3314601`. URL: `http://fsl.cs.illinois.edu/index.php/A_Comp` `lete_Formal_Semantics_of_x86-64_User-Level_Instruction_Set_Architecture`.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Communications of the ACM* 5.7 (July 1, 1962), pages 394–397. ISSN: 0001-0782. DOI: `10.1145/368273.368557`. URL: `https://homepage.cs.uiowa.edu/~tinelli/classes/295/Spring03/papers/Dav62.pdf` (visited on 07/20/2020).

[DM79] Nachum Dershowitz and Zohar Manna. "Proving termination with multiset orderings". In: *Communications of the ACM* 22.8 (Aug. 1, 1979), pages 465–476. ISSN: 00010782. DOI: `10.1145/359138.359142`. URL: `https://portal.acm.org/citation.cfm?doid=359138.359142`.

[Dol13] Stephen Dolan. *mov is Turing-complete*. July 19, 2013. URL: `http://stedolan.net/research/mov.pdf` (visited on 03/11/2020).

[Dom15] [SOFTWARE] Chris Domas, *movfuscator*, 2015. URL: `https://github.com/xoreaxeaxeax/movfuscator`(visited on 03/11/2020).

[Dow97] Mark Dowson. "The Ariane 5 software failure". In: *ACM SIGSOFT Software Engineering Notes* 22.2 (Mar. 1, 1997), page 84. ISSN: 0163-5948. DOI: `10.1145/251880.251992`. URL: `https://dl.acm.org/doi/10.1145/251880.251992`.

[DSB86] Michel Dubois, Chrlstoph Scheurich, and Faye Briggs. "Memory Access Buffering in Multiprocessors". In: *Proceedings of the 13th Annual International Symposium on Computer Architecture*. ICSA '86. Tokyo, Japan: IEEE Computer Society Press, May 1986, pages 434–442. ISBN: 0-8186-0719-X. DOI: `10.1145/17356.17406`. URL: `https://dl.acm.org/doi/10.5555/17407.17406`.

[ECM91] ECMA. *Control Functions for Coded Character Sets*. ECMA standard 48. Ecma International, 1991. URL: `https://www.ecma-international.org/publications/standards/Ecma-048.htm`.

[Fer00] Jérôme Feret. "Confidentiality Analysis of Mobile Systems". In: *Static Analysis*. Redacted by Jan van Leeuwen, Gerhard Goos, and Juris Hartmanis. Volume 1824. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pages 135–154. ISBN: 978-3-540-45099-3. DOI: `10.1007/978-3-540-45099-3_8`. URL: `https://link.springer.com/10.1007/978-3-540-45099-3_8`.

[FI10] Peter Fornai and Antal Ivanyi. "FIFO anomaly is unbounded". In: *arXiv:1003.1336 [cs]* (Mar. 9, 2010). arXiv: `1003.1336`. URL: `https://arxiv.org/abs/1003.1336`.

[GAO92] U. S. Government Accountability Office. *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia.* Report to the Chairman, Subcommittee on Investigations and Oversight, Committee on Science, Space, and Technology, House of Representatives IMTEC-92-26. GAO, Feb. 27, 1992. URL: `https://www.gao.gov/products/IMTEC-92-26` (visited on 06/26/2020).

[GSS07] Walter Greiner, Stefan Schramm, and Eckart Stein. *Quantum Chromodynamics.* 3rd edition. Berlin Heidelberg: Springer-Verlag, 2007. ISBN: 978-3-540-48534-6. DOI: `10.1007/978-3-540-48535-3`. URL: `https://www.springer.com/gp/book/9783540485346`.

[HO80] Gérard Huet and Derek C. Oppen. "Equations and Rewrite Rules: A Survey". In: *Formal Language Theory.* Edited by Ronald Vernon Book. Academic Press, 1980, pages 349–405. ISBN: 978-0-12-115350-2. DOI: `10.1016/B978-0-12-115350-2.50017-8`. URL: `http://rewriting.loria.fr/documents/CS-TR-80-785.pdf` (visited on 05/04/2020).

[HP08] Nicolas Halbwachs and Mathias Peron. "Discovering Properties about Arrays in Simple Programs". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008.* June 2008, page 10. DOI: `10.1145/1375581.1375623`. URL: `https://hal.archives-ouvertes.fr/hal-00288274/fr/`.

[Int20] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4.* 325462-072US. May 2020. URL: `https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html` (visited on 06/29/2020).

[Int99] Intel. *Pentium ® Processor Specification Update.* en. 242480-041. Jan. 1999. Chapter 81, pages 59–60. 101 pages. URL: `https://web.archive.org/web/20160304060905/http://download.intel.com/support/processors/pentium/sb/242480.pdf`.

[ISO02] ISO. *Control Functions for Coded Character Sets.* ISO standard 6429:1992. International Organization for Standardization, Dec. 2002. URL: `https://www.iso.org/standard/12782.html`.

[JMO20] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. "Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer". In: *Verified Software. Theories, Tools, and Experiments.* Edited by Jorge A. Navas and Supratik Chakraborty. Volume 12031. Cham: Springer International Publishing, 2020, pages 1–18. ISBN: 978-3-030-41600-3. DOI: `10.1007/978-3-030-41600-3_1`. URL: `https://link.springer.com/10.1007/978-3-030-41600-3_1`.

[Jou+15] Jacques-Henri Jourdan et al. “A Formally-Verified C Static Analyzer”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL ’15*. the 42nd Annual ACM SIGPLAN-SIGACT Symposium. Mumbai, India: ACM Press, 2015, pages 247–259. ISBN: 978-1-4503-3300-9. DOI: `10.1145/2676726.2676966`. URL: `https://dl.acm.org/citation.cfm?doid=2676726.2676966`.

[Ler+20] [SOFTWARE] Xavier Leroy et al., *OCaml* version 4.10.0, 2020. INRIA. LIC: LGPL 2.1. URL: `https://ocaml.org/`(visited on 05/11/2020), VCS: `https://github.com/ocaml/ocaml`.

[LT93] N.G. Leveson and C.S. Turner. “An investigation of the Therac-25 accidents”. In: *Computer* 26.7 (July 1993). Conference Name: Computer, pages 18–41. ISSN: 1558-0814. DOI: `10.1109/MC.1993.274940`. URL: `https://web.stanford.edu/class/cs240/old/sp2014/readings/therac-25.pdf` (visited on 06/26/2020).

[Mau99] Laurent Mauborgne. “Representation of Sets of Trees for Abstract Interpretation”. PhD thesis. École Polytechnique, Nov. 25, 1999. 197 pages. URL: `https://www.di.ens.fr/~mauborgn/publi/t.pdf` (visited on 04/05/2020).

[Min06] Antoine Miné. “Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics”. In: *ACM SIGPLAN Notices*. Volume 41. ACM, 2006, pages 54–63. ISBN: 1-59593-362-X. DOI: `10.1145/1159974.1134659`. URL: `https://hal.archives-ouvertes.fr/hal-00136650`.

[Min13] Antoine Miné. “Static analysis by abstract interpretation of concurrent programs”. HDR. Ecole Normale Supérieure de Paris-ENS Paris, 2013. URL: `https://hal.archives-ouvertes.fr/tel-00903447/`.

[Min14] Antoine Miné. “Relational thread-modular static value analysis by abstract interpretation”. In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI. San Diego, CA, USA: Springer, 2014, pages 39–58. ISBN: 978-3-642-54013-4. DOI: `10.1007/978-3-642-54013-4_3`. URL: `https://hal.archives-ouvertes.fr/hal-00925713/`.

[Nav+12] Jorge A. Navas et al. “Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code”. In: *APLAS*. APLAS. Kyoto, Japan: Springer, 2012, pages 115–130. ISBN: 978-3-642-35182-2. DOI: `10.1007/978-3-642-35182-2_9`. URL: `https://link.springer.com/chapter/10.1007/978-3-642-35182-2_9`.

[Nec+02] George C. Necula et al. “CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs”. In: *Compiler Construction*. Edited by R. Nigel Horspool. Redacted by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Volume 2304. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pages 213–

228. ISBN: 978-3-540-45937-8. DOI: `10.1007/3-540-45937-5_16`. URL: `https://link.springer.com/10.1007/3-540-45937-5_16`.

[NIST02] National Institute of Standards & Technology. *The Economic Impacts of Inadequate Infrastructure for Software Testing.* Planning Report 02-3. NIST, May 2002. URL: `https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf` (visited on 06/26/2020).

[OSD20] OSDev community. *OSDev.* 2020. URL: `https://wiki.osdev.org/`.

[Pér10] Mathias Péron. "Contributions à l'analyse statique de programmes manipulant des tableaux". PhD thesis. Grenoble, France: Grenoble Alpes University, France, Sept. 22, 2010. 231 pages. URL: `https://hal.archives-ouvertes.fr/tel-00623697`.

[Ric53] H. G. Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical Society* 74.2 (1953), pages 358–366. ISSN: 1088-6850. DOI: `10.1090/S0002-9947-1953-0053041-6`. URL: `https://www.ams.org/journals/tran/1953-074-02/S0002-9947-1953-0053041-6/S0002-9947-1953-0053041-6.pdf`.

[Sha06] William Shakespeare. *The Tragedy of Macbeth.* Act V, scene 1. 1606. URL: `https://en.wikisource.org/wiki/Macbeth_(1918)_Yale/Text/Act_V` (visited on 07/16/2020).

[Ven96] Arnaud Venet. "Abstract cofibered domains: Application to the alias analysis of untyped programs". In: *Static Analysis.* Redacted by Juris Hartmanis, Jan van Leeuwen, and Gerhard Goos. Volume 1145. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pages 366–382. ISBN: 978-3-540-61739-6. DOI: `10.1007/3-540-61739-6_53`. URL: `https://link.springer.com/10.1007/3-540-61739-6_53`.

[Ven98] Arnaud Venet. "Automatic Determination of Communication Topologies in Mobile Systems". In: *Static Analysis.* Redacted by Juris Hartmanis, Jan van Leeuwen, and Gerhard Goos. Volume 1503. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pages 152–167. ISBN: 978-3-540-49727-1. DOI: `10.1007/3-540-49727-7_9`. URL: `https://link.springer.com/10.1007/3-540-49727-7_9`.

[Ven99] A. Venet. "Automatic analysis of pointer aliasing for untyped programs". In: *Science of Computer Programming* 35.2 (Nov. 1999), pages 223–248. ISSN: 01676423. DOI: `10.1016/S0167-6423(99)00012-X`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S016764239900012X`.

Index

List of Acronyms

ABI	application binary interface	87
ANSI	American National Standards Institute	311
AST	abstract syntax tree	164
ATV	Automated Transfer Vehicle	163
BCD	binary-coded decimal	68
CFG	control flow graph	94
CIE	International Commission on Illumination (*French*: Commission internationale de l’éclairage)	382
CIL	C Intermediate Language	164
CISC	complex instruction set computer	8
CPL	current privilege level	14
CPO	complete partial order	145
CPU	central processing unit	173
CTO	compute through overflow	295
DAG	directed acyclic graph	94
DPL	descriptor privilege level	14
DSL	domain-specific language	339
ESA	European Space Agency	163
FFI	foreign function interface	110
FPU	floating-point unit	201
FSA	finite-state automaton	307
GDT	global descriptor table	19
GPR	general-purpose register	10
IDT	interrupt descriptor table	24
ISQ	International System of Quantities	370
ISS	International Space Station	163
JSON	JavaScript Object Notation	384
LDT	local descriptor table	19
NIST	National Institute of Standards and Technology	1
OS	operating system	3
RAM	random-access memory	9
RGB	red-green-blue	382
RISC	reduced instruction set computer	207

List of Figures

List of Listings

List of Tables

Appendices

Appendix A

Notations and Basic Definitions

THIS CHAPTER sums up notations and goes beyond by explaining some choices. It covers most topics, from architecture to fundamental mathematics, and recalls classical notations as well as it introduces more original ones.

A.1 Architecture

31 24	23 16	15 8	7 0	
`arr[7]`	`arr[6]`	`arr[5]`	`arr[4]`	4
`arr[3]`	`arr[2]`	`arr[1]`	`arr[0]`	0

Figure A.1: Layout of a 8-element array of 1-byte values

31 24	23 16	15 8	7 0
`n`[7:0]	`n`[15:8]	`n`[23:16]	`n`[31:24]

Figure A.2: Switching between big- and little-endian representation

When talking with computers about integers, it is often nice to use a non-decimal radix. Especially, binary and hexadecimal are good natural choices. Most programming languages use respectively the prefixes `0b` and `0x`. But, to follow INTEL's manual's style, we also use suffixes B and H, in particular in a mathematical (non-code) context. For instance, $10000\text{B} = 10\text{H} = 16$.

We also follow INTEL's manual's notation of memory structures: bits are placed from right to left (so that number are written in natural order) and bytes of lower address to the bottom. Bit offset is written on the top and byte offset on the right. An array `arr` of 1-byte values will be arranged as on Figure A.1, using C notation for array accesses.

We also need to cut bit sequences (like integers or addresses) and to rearrange them. To extract a slice of bits, the syntax is $s\,[a : b]$ where s is the sequence, and a and b are respectively the indices of the beginning and the end of the extracted sequence. In the context of a graphical representation, we may put the end before, so that $a \geqslant b$, to imitate the bit ordering. For instance, given a 32-bit integer `n`, Figure A.2 shows how to use this notation to switch between big- and little-endian representation. For example, if `n` is 12345678H (colored by packs of 8 bits) then `n`$[7:0]$ is 78H, `n`$[15:8]$ is 56H, `n`$[23:16]$ is 34H and `n`$[31:24]$ is 12H. After the transformation of Figure A.2, we would get 78563412H. Overall, we write indistinctly $s\,[a:b]$ and $s\,[b:a]$.

Units and prefixes are a common source of confusion when counting bits. We follow the specification of the International System of Quantities (ISQ) (ISO/IEC 80000) that defines, among other things, the prefix and quantity used in the SI*. The SI defines a proper subset of prefixes with respect to ISQ, but they agree the ones they have in common. However, the SI does not define units and prefixes specific to computer science.

The ISQ defines 3 units we are interested in: bit (bit), byte (B) and octet (o). A bit is a single binary digit, a very fundamental piece of information. An octet is by definition a sequence of 8 bits. A byte is the smallest addressable unit of memory. Once upon a time, bytes were not always 8-bit long. But that time is over. It is now the age of 8-bit bytes. And (not so) recent standards go in this direction. The x86 architecture plays it safe: a byte is indeed an octet. So we use both without worrying too much.

The other problematic point is the correct usage of prefixes. The SI prefixes are not questioned: k (kilo) is 10^3, M (mega) is 10^6, and so on. Thus, a kilobyte (kB) is 1000 B. For binary-friendly prefixes we used dedicated ones, as defined in the ISQ. These prefixes are listed in Table A.1. In practice, we use only the 3 lower ones.

Value		Symbol	Name
2^{10}	1024	Ki	kibi
2^{20}	1024^2	Mi	mebi
2^{30}	1024^3	Gi	gibi
2^{40}	1024^4	Ti	tebi
2^{50}	1024^5	Pi	pebi
2^{60}	1024^6	Ei	exbi
2^{70}	1024^7	Zi	zebi
2^{80}	1024^8	Yi	yobi

Table A.1: Binary prefixes

A.2 Logic

We write $a := b$ to define a as equal to b. We prefer this notation over $\triangleq$, or $\stackrel{\text{def}}{=}$ because it specifies which is the new symbol. One could as well write unambiguously $b =: a$. In the same way, we write $A :\Leftrightarrow B$ and $B \Leftrightarrow: A$ to define A as equivalent to B.

*French: Système international, International System of Units

We also use the IVERSON's bracket notation. Given a proposition P

$$[P] = \begin{cases} 1 & \text{if } P \\ 0 & \text{if } \neg P \end{cases}$$

This notation subsumes other notations like the indicator function ($\mathbb{1}_A(x) = [x \in A]$) or the KRONECKER delta ($\delta_{i,j} = [i = j]$).

We denote tt (resp. ff) the truth value "true" (resp. "false").

Notation	**Meaning**
$a := b$	defines a as equal to b
$b =: a$	defines a as equal to b
$A :\Leftrightarrow B$	defines A as equivalent to B
$B \Leftrightarrow: A$	defines A as equivalent to B
$[P]$	IVERSON's bracket
tt	logical true
ff	logical false

Table A.2: Logic notations

A.3 Set Theory

Notation	**Meaning**
$x \in A$	x belongs to A
$A \subseteq B$	A is a subset of B
$A \subsetneq B$	A is a proper subset of B
$\mathcal{P}(A)$	the power set of A
$A \cup B$	union of A and B
$A \cap B$	intersection of A and B
$A \setminus B$	A without B
$\varnothing$	the empty set
$\mathbb{N}$	set of natural integers (including 0)
$\mathbb{Z}$	set of integers
$\mathbb{N}^*$	set of positive integers (so excluding 0)
A^*	$A \setminus \{0\}$ (where A is a numerical set)
$[\![a, b]\!]$	the set of integers between a and b
$A^{\uparrow\mathcal{R}}$	set of elements greater than an element of A according to $\mathcal{R}$
$A^{\downarrow\mathcal{R}}$	set of elements lower than an element of A according to $\mathcal{R}$

Table A.3: Set theory notations

Classical notations are recalled on Table A.3.

Relational notations can be used in their reversed variant, e.g. $A \ni x \Leftrightarrow x \in A$. We can also use crossed out versions to state the opposite e.g. $A \not\subsetneq B \Leftrightarrow \neg A \subsetneq B$ (A is not a proper subset of B).

A less classical set operation we use sometimes is the disjoint union. This is a touchy subject with no perfect solution. The idea is to compute the union of two (or more) sets, without collapsing elements that appears in both. Moreover, elements are tagged so that we know from which of the source sets they come. Like in many cases, they are two philosophies to define it: explicit ("this is how the disjoint union is defined") or axiomatic ("this is how disjoint union should behave"). Category theory has an axiomatic definition of a generalization of disjoint union (the sum or coproduct). It is not very handy and does not define it uniquely. Explicit definitions of $A \uplus B$ usually build the set of elements that are either in A or B, accompanied by a tag to discriminate the origin. For instance, we can let

$$A \uplus B := \{(\varnothing, a) \mid a \in A\} \cup \{(\{\varnothing\}, b) \mid b \in B\}$$

Here, we choose $\varnothing$ and $\{\varnothing\}$ for the tags because they are very fundamental objects, but the choice does not matter at all. This definition is very simple but it has several flaws, most important ones are that this operator is not associative, not commutative and that $A \not\subseteq A \uplus B$. The latter point force us to explicit the tag; we do not want that.

Overall, there is no good solution that provides a formal definition and that is easy to use when writing math (in particular, that allows to forget about the tag). So, we are deliberately misusing the notations: we take the liberty to write $x \in A \uplus B$ and then distinguish cases whether $x \in A$ or $x \in B$, provided that exactly one of these possibility is true.

Let us also remark that we are using $\uplus$ for disjoint union, and not the usual $\sqcup$, because we keep the latter for the "join" operator of a lattice.

Let us also clarify the case of intervals. Closed intervals of real numbers are denoted $[a, b]$, and the corresponding open interval is denoted $]a, b[$. We do not use the style (a, b) because it is indistinguishable from the ordered pair. The order of bounds of a real interval does not matter since

$$[a, b] := \{t \cdot a + (1 - t) \cdot b \mid 0 \leqslant t \leqslant 1\}$$

So $[0, 1] = [1, 0]$. For integer intervals, we use the notation $[\![a, b]\!]$. In this case, the order matters since

$$[\![a, b]\!] := \{x \in \mathbb{Z} \mid a \leqslant x \leqslant b\}$$

For instance $[\![0, 1]\!] = \{0, 1\}$ but $[\![1, 0]\!] = \varnothing$. This is very useful for base cases of recursive formulae, for iterative definitions or limit cases.

Let E be a set and $\mathcal{R}$ be an order relation on E. Let $A \subseteq E$, we write $A^{\uparrow\mathcal{R}} := \{x \in E \mid \exists y \in E : y\mathcal{R}x\}$, the set of elements of E that are bigger than an element in A. Symmetrically, $A^{\downarrow\mathcal{R}} := \{x \in E \mid \exists y \in E : x\mathcal{R}y\}$. For convenience, given $a \in E$, we write $a^{\uparrow\mathcal{R}} := \{a\}^{\uparrow\mathcal{R}}$ and $a^{\downarrow\mathcal{R}} := \{a\}^{\downarrow\mathcal{R}}$. This notation can be generalized to any relation, though it would usually be ill-advised. Yet, an interesting and nice case is the case when $\mathcal{R}$ is a strict order.

A.4 Multisets

Multisets are basically sets that allow multiple occurrences of each element. Formally a multiset is an ordered pair (A, m) where A is a set, called underlying set, and $m : A \to \mathbb{N}$ a function giving the multiplicity of each element. Multisets are denoted as $\{\!\!\{\_\}\!\!\}$. For instance, $\{\!\!\{0, 0, 1\}\!\!\}$ is the multiset

$$(\{0, 1\}, (0 \mapsto 2, 1 \mapsto 0))$$

It also is

$$(\{0, 1, 2\}, (0 \mapsto 2, 1 \mapsto 0, 2 \mapsto 0))$$

Indeed, two multisets (A, m_a) and (B, m_B) are equal if

$$\begin{array}{lr}(\forall x \in A \cap B, m_A(x) = m_B(x)) \land & \wr \text{ elements in } A \text{ and } B \text{ have same multiplicity } \int \\ (\forall x \in A \setminus B, m_A(x) = 0) \land & \wr \text{ elements only in } A \text{ have multiplicity } 0 \int \\ (\forall x \in B \setminus A, m_B(x) = 0) & \wr \text{ elements only in } B \text{ have multiplicity } 0 \int\end{array}$$

We can always extend the underlying set by assigning a zero multiplicity to new elements. Sets can be seen as a particular case of multisets where multiplicity is only 0 or 1. We only allow multisets where multiplicity of each element is finite. We do not need more, and infinite multiplicity seems to be much less discussed in the literature.

Given a finite set E, we denote

$$\mathcal{M}(E) := \{E\} \times E^{\mathbb{N}}$$

the set of finite multisets whose underlying set is E.

Operations and relations on multisets use the same notations as their set analogs. Let $\mathcal{A} = (A, m_A)$, $\mathcal{B} = (B, m_B)$ and $\mathcal{C} = (C, m_C)$ be multisets. We extend tacitly m_A, m_B and m_C to $A \cup B \cup C$.

We have

$$\begin{aligned}\mathcal{A} \subseteq \mathcal{B} &:\Leftrightarrow A \subseteq B \land \forall a \in A, m_A(a) \leqslant m_B(a) \\ \mathcal{C} = \mathcal{A} + \mathcal{B} &:\Leftrightarrow \forall x \in A \cup B \cup C, m_C(x) = m_A(x) + m_B(x) \\ \mathcal{C} = \mathcal{A} \setminus \mathcal{B} &:\Leftrightarrow \forall x \in A \cup B \cup C, m_C(x) = \max(m_A(x) - m_B(x), 0) \\ \mathcal{C} = \mathcal{A} \cup \mathcal{B} &:\Leftrightarrow \forall x \in A \cup B \cup C, m_C(x) = \max(m_A(x), m_B(x)) \\ \mathcal{C} = \mathcal{A} \cap \mathcal{B} &:\Leftrightarrow \forall x \in A \cup B \cup C, m_C(x) = \min(m_A(x), m_B(x))\end{aligned}$$

In [DM79], the union is defined as we defined the sum. We did not make the same choice so that $\cup$ coincide with the union of sets.

Let E a set and $\sqsubseteq$ a partial order on E. We denote $\sqsubset_{\mathrm{DM}}$, and called Dershowitz–Manna order (or multiset order) induced by $\sqsubseteq$, the relation on $\mathcal{M}(E)$ defined by

$$\forall (\mathcal{A}, \mathcal{B}) \in \mathcal{M}(E)^2, \mathcal{A} \sqsubset_{\mathrm{DM}} \mathcal{B} :\Leftrightarrow \exists (\mathcal{X}, \mathcal{Y}) \in \mathcal{M}(E)^2 : \begin{pmatrix} \quad \mathcal{X} \neq \varnothing \\ \land\ \mathcal{X} \subseteq \mathcal{B} \\ \land\ \mathcal{A} = (\mathcal{B} \setminus \mathcal{X}) + \mathcal{Y} \\ \land\ \forall y \in \mathcal{Y}, \exists x \in \mathcal{X} : y \sqsubset x \end{pmatrix}$$

We denote $\sqsubseteq_{\mathrm{DM}}$ the reflexive closure of $\sqsubset_{\mathrm{DM}}$. In other words, to get a smaller multiset, we can remove an occurrence of an element x of B to add as many elements as we want that are lower than x. In [HO80], there is another, equivalent, definition which is also closer to the interpretation. The alternative definition can be written:

$$\forall(\mathcal{A},\mathcal{B}) \in \mathcal{M}(E)^2, \mathcal{A} \sqsubset_{\mathrm{DM}} \mathcal{B} :\Leftrightarrow \mathcal{A} \neq \mathcal{B} \wedge \left(\begin{array}{l}\textbf{let } (A, m_A) := \mathcal{A} \textbf{ in let } (B, m_B) := \mathcal{B} \textbf{ in}\\ \forall x \in E, m_A(x) > m_B(x) \Rightarrow\\ \quad \exists y \in E : x \sqsubset y \wedge m_A(x) < m_B(x)\end{array}\right)$$

This relation has some good properties. First, $\sqsubseteq_{\mathrm{DM}}$ is a partial order on $\mathcal{M}(E)$. Moreover, if $\sqsubseteq$ is well-founded, then $\sqsubseteq_{\mathrm{DM}}$ is a well-founded order on $\mathcal{M}(E)$. The converse is also true, but less interesting for us. These propositions are well-known properties of multiset ordering (see [DM79]).

Notation	Meaning
$\{\!\{a_1, \dots, a_n\}\!\}$	multiset containing $a_1, \dots, a_n$
$\mathcal{M}(E)$	set of finite multisets whose underlying set is E
$\sqsubseteq_{\mathrm{DM}}$	Dershowitz–Manna order induced by $\sqsubseteq$
$A \subseteq B$	multiset inclusion (comparison of multiplicities)
$A + B$	multiset sum (sum of multiplicities)
$A \setminus B$	multiset difference (difference of multiplicities)
$A \cup B$	multiset union (maximum of multiplicities)
$A \cap B$	multiset intersection (minimum of multiplicities)

Table A.4: Multiset notations

A.5 Function Theory

We use Bourbaki-style functions. A partial function from A to B is a 3-tuple (A, B, G) where A and B are sets and $G \subseteq A \times B$ such that $\forall x \in A, \forall(y, y') \in B^2, (x, y) \in G \wedge (x, y') \in G \Rightarrow y = y'$. It is a total function if in addition $\forall x \in A, \exists y \in B : (x, y) \in G$.

The set of total functions from A to B is denoted B^A. We can also write $f : A \to B$ to define such a function. When f is only a partial function (i.e. a function for which some elements of A may not have an image), we write $f : A \rightharpoonup B$. The support of f, denoted $\operatorname{supp}(f)$, is the set of elements of A which have an image, i.e. given a function (A, B, G), $\operatorname{supp}(f) := \{x \in A \mid \exists y \in B : (x, y) \in G\}$. If f is a total function, $\operatorname{supp}(f) = A$. Writing $f(x)$ when x is not in the support of f has no meaning. We define $\operatorname{Im}(f) := \{y \in B \mid \exists x \in A : (x, y) \in G\} = \{f(x) \mid x \in \operatorname{supp}(A)\} \subseteq B$. The way functions are defined is merely a formalization issue that appears when talking about partial functions and related difficulties. As much as possible, functions will be treated without opening the box.

Given a function $f : A \rightharpoonup B$ and X a set, we write $f_{|X}$ the restriction of f to $X \cap A$, that is, the function from X to B that is defined only on $X \cap \operatorname{supp}(f)$ and is equal to f on this subset. Dually, given a set Y, we define the corestriction $f^{|Y}$ to be the function from A to $Y \cap B$ equal to f where applicable. We have, $\operatorname{supp}\left(f^{|Y}\right) = \{a \in \operatorname{supp}(f) \mid f(a) \in Y\}$. Thus, in general, a corestriction is a partial function. If we make explicit how functions are built, it is even easier (but not as intuitive): $f_{|X} = (X \cap A, B, G \cap (X \times B))$ and $f^{|Y} = (A, Y \cap B, G \cap (A \times Y))$. Of course, a function can be restricted and corestricted.

Another interesting way to build function is to update a single image. Let $f : A \rightharpoonup B$, $a \in A$ and $b \in B$. We denote $f\left[a \mapsto b\right]$ the function

$$\begin{aligned} f\left[a \mapsto b\right] &: A \rightharpoonup B \\ &x \mapsto \begin{cases} b & \text{if } x = a \\ f(x) & \text{if } x \neq a \wedge x \in \operatorname{supp}(f) \end{cases} \end{aligned}$$

It is simply the function f, but the image of a is overridden to be b. Consequently, $\operatorname{supp}(f) \cup \{a\} = \operatorname{supp}(f\left[a \mapsto b\right])$ (no matter if $a \in \operatorname{supp}(f)$). We can iterate this process: $f\left[a_1 \mapsto b_1\right]\left[a_2 \mapsto b_2\right] = (f\left[a_1 \mapsto b_1\right])\left[a_2 \mapsto b_2\right]$ for any $(a_1, a_2) \in A^2$ (with $a_1 \neq a_2$) and $(b_1, b_2) \in B^2$. And more generally, given $n \in \mathbb{N}$, $(a_i)_{i \in [\![1,n]\!]} \in A^n$ such that $\forall (i,j) \in [\![1,n]\!]^2, a_i = a_j \Rightarrow i = j$ and $(b_i)_{i \in [\![1,n]\!]} \in B^n$, we define

$$f\left[a_i \mapsto b_i\right]_{i=1}^{n} = \begin{cases} f & \text{if } n = 0 \\ f\left[a_n \mapsto b_n\right]\left[a_i \mapsto b_i\right]_{i=1}^{n-1} & \text{otherwise} \end{cases}$$

or by unfolding the definition

$$f\left[a_i \mapsto b_i\right]_{i=1}^{n} = (A, B, \{(x,y) \in G \mid \forall i \in [\![1,n]\!], x \neq a_i\} \cup \{(a_i, b_i) \mid i \in [\![1,n]\!]\})$$

With this second version, we can directly generalize it to any indexing set.

A useful feature of functional languages is partial application. For curried function, this adapts directly into math. But usually in math, for readability reasons, we use uncurried functions. For instance, let E, F and G be sets, $u : E \times F \to G$ and $(e, f) \in E \times F$. We can write $x \mapsto u(x, f)$ and $y \mapsto u(e, y)$, but this is a bit verbose. Thus, we also use a special symbol to be a placeholder: $\_$. Any argument that receives this symbol is unbounded. For instance $(x \mapsto u(x, f)) = u(\_, f)$ and $(y \mapsto u(e, y)) = u(e, \_)$. We also write $\__i$ where $i \in \mathbb{N}^*$ to reorder unbounded parameters. For instance $((y, x) \mapsto u(x, y)) = u(\__2, \__1)$.

Notation	**Meaning**
B^A	set of functions from A to B
$f : A \to B$	f is a function from A to B
$f : A \rightharpoonup B$	f is a partial function from A to B
$\operatorname{supp}(f)$	support of f
$\operatorname{Im}(f)$	image of f
$f_{\vert X}$	restriction of f to X
$f^{\vert Y}$	corestriction of f to Y
$f\,[a \mapsto b]$	f except that a maps to b

Table A.5: Logic notations

A.6 Standard Objects

We denote by π_i the i^{th} projection, that is the function that yields the i^{th} component of a tuple:

$$\pi_i\left((a_1, \ldots, a_n)\right) := a_i$$

with $i \leqslant n$.

For all finite set E, we write $\mathfrak{S}_E$ the symmetric group of E, that is the set of all bijective functions of $E \to E$. For all $n \in \mathbb{N}^*$, we write $\mathfrak{S}_n$ the symmetric group of degree n, that is $\mathfrak{S}_n := \mathfrak{S}_{[\![1,n]\!]}$.

For all set E, we write $E^\star$ the KLEENE star of E, i.e. $E^\star$ is the set of finite sequences (that is, tuples) of elements of E:

$$E^\star := \bigcup_{i=0}^{\infty} E^i$$

and we write

$$E^+ := \bigcup_{i=1}^{\infty} E^i$$

the set of finite non-empty sequences of elements of E. The KLEENE star notation looks like notation of numerical set without 0, yet the star is different: $\mathbb{N}^\star$ is the set of finite sequences of natural numbers, while $\mathbb{N}^*$ is the set of positive natural numbers.

Given integers a, b and n, we write $a \bmod n$ the binary modulo operation applied to a and n (that is, the remainder of the Euclidean division of a by n). We denote $a \equiv b\,[n]$ the congruence relation between a and b modulo n, that is

$$\exists k \in \mathbb{Z} : a - b = kn$$

Given a set E, we write Id_E (or simply Id) the identity function:

$$\begin{aligned}\mathrm{Id}_E : E &\to E\\ x &\mapsto x\end{aligned}$$

Given a set E, a function $f : E \to E$, and an integer $n \in \mathbb{N}$, we write f^n the n^{th} iterate of f. Formally

$$f^n := \begin{cases} \mathrm{Id}_E & \text{if } n = 0 \\ f \circ f^{n-1} & \text{if } n > 0 \end{cases}$$

A.7 Local Notations

In this work, we introduce many notations that are not standard. They are recalled here as the user may lose where they come from and what they mean. For each notation, we specify where it was introduced, and we recall briefly its meaning. Notations of each part are mostly independent.

A.7.1 Notations of Part I "Semantics of Mixed C and x86 Assembly"

Table A.6: Notations of Part I

Notation	**Defined**	**Meaning**
$\mathbb{V}$	p. 44	set of Asclepius variables
$\mathbb{F}$	p. 44	set of Asclepius functions
$\mathbb{P}_{\mathrm{C}}$	def. 4.2 p. 44	set of Asclepius programs
$\mathbb{F}[P]$	def. 4.2 p. 44	set of functions in program P
$\mathrm{Body}_P(f)$	def. 4.2 p. 44	body of statement f in program P
$\|x\|$	not. 4.1 p. 44	size of lvalue or type x
$\mathrm{Stat}_{\mathrm{C}}$	not. 4.2 p. 45	set of Asclepius statements
Π	def. 4.4 p. 45	set of paths
$\mathrm{Cmp}^{\pi}(S)$	def. 4.5 p. 46	components of statement S at path π
$\mathrm{Cmp}_P(f)$	def. 4.5 p. 46	components of function f in program P
$\mathrm{Cmp}(P)$	def. 4.5 p. 46	components of program P
$\mathbb{L}_{\mathrm{C}}$	p. 46	set of Asclepius labels
$\mathrm{at}_P(C)$	p. 46	label at component C in program P
$\mathrm{after}_P(C)$	p. 46	label after component C in program P
$\mathrm{in}_P(C)$	p. 46	set of labels in components C in program P
${}^{l}S^{m}$	not. 4.3 p. 47	statement S preceded by label l and followed by label m
$\mathbb{A}$	p. 48	set of memory addresses
$\mathbb{I}$	p. 48	set of possible values for a byte
$\mathbb{E}_{\mathrm{C}}$	p. 48	set of Asclepius environments
$\mathbb{H}_{\mathrm{C}}$	p. 48	set of Asclepius heaps
$\mathbb{M}_{\mathrm{C}}$	p. 48	set of Asclepius memory states
ω_a	def. 4.8 p. 50	invalid address error
$[\![op]\!]$	not. 4.6 p. 51	semantics of operator op

— Continued on following page —

Table A.6 – Notations of Part I

Notation	Defined	Meaning
$\langle\!\vert l \vert\!\rangle$	def. 4.9 p. 51	semantics of lvalue l
$(\!\vert e \vert\!)$	def. 4.9 p. 51	semantics of expression e
$\mathbb{V}\left[l\right]$	not. 4.7 p. 51	set of living variables at label l (program is implicit)
$\mathbb{M}\left[V\right]$	not. 4.9 p. 52	set of ASCLEPIUS memory states where set of variables is V
Ω	p. 52	set of possible errors
$\tau_{\mathrm{C}}^{P}\left[{}^{a}S^{b}\right]$	p. 52	set of transition generated from statement ${}^{a}S^{b}$ in program P
ω_o	p. 54	out of memory error
$\mathbb{A}_{\mathbb{R}}$	p. 66	set of register byte addresses
$\mathbb{R}$	p. 67	set of register names
$\mathbb{H}_{\mathrm{asm}}$	p. 67	set of assembly heaps
$\mathrm{reg}\left(r\right)$	p. 67	address of first byte of register r
$\mathbb{E}_{\mathrm{asm}}$	p. 67	set of assembly environments
$\mathrm{Stat}_{\mathrm{asm}}$	p. 68	set of assembly instructions
$\mathrm{disassemble}\left(h\right)$	p. 68	disassemble next instruction to run in memory state h
ω_i	p. 68	illegal instruction error
$\mathbb{M}^{\flat}_{\mathrm{asm}}$	p. 69	set of memory states in the most concrete interpretation of assembly
$\mathbb{M}_{\mathrm{asm}}$	p. 79	set of assembly memory states in the more symbolic semantics
Stat	def. 8.2 p. 100	set of PANACEA statements
$\mathbb{P}$	def. 8.2 p. 100	set of PANACEA programs
$\mathbb{L}_{\mathrm{asm}}$	p. 101	set of assembly labels
$\mathbb{L}$	p. 101	set of PANACEA labels
$\mathbb{E}$	p. 104	set of PANACEA environments
$\mathbb{M}$	p. 104	set of PANACEA memory states

A.7.2 Notations of Part II “Abstract Interpretation and the Astrée Analyzer”

Table A.7: Notations of Part II

Notation	Defined	Meaning
$\mathrm{fp}\, f$	not. 11.7 p. 143	set of fixpoints of f
$\mathrm{lfp}_a\, f$	not. 11.7 p. 143	least fixpoint of f greater than a
$\mathrm{lfp}\, f$	not. 11.7 p. 143	least fixpoint of f
$\mathrm{gfp}_a\, f$	not. 11.7 p. 143	greatest fixpoint of f lower than a
$\mathrm{gfp}\, f$	not. 11.7 p. 143	greatest fixpoint of f

— Continued on following page —

Table A.7 – Notations of Part II

Notation	Defined	Meaning
$(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$	not. 11.8 p. 145	GALOIS connection

A.7.3 Notations of Part IV "A Reduced Product with Ghost Variables"

Table A.8: Notations of Part IV

Notation	Defined	Meaning
$\mathbb{V}$	p. 237	set of real variables
$\mathbb{I}$	p. 237	set of possible values of variables
$\mathbb{S}$	p. 237	set of memory states
⚲	p. 238	set of expression constructors
$C^{/n}$	p. 238	constructor C of arity n
$\mathbb{E}_V$	not. 19.1 p. 239	set of expressions with variables in V
$\mathbb{E}$	not. 19.2 p. 239	set of real expressions
$\trianglelefteq$	def. 19.1 p. 239	structural inclusion of expressions
$\mathbb{O}$	not. 19.4 p. 240	set of condition constructors
$\mathbb{C}_V$	not. 19.4 p. 240	set of conditions with variables in V
$\mathbb{C}$	not. 19.4 p. 240	set of real conditions
$Alloc\,(v, d)$	p. 241	allocates variable v in state d
$Kill\,(v, d)$	p. 241	kills variable v in state d
$Assign\,(a, d)$	p. 241	performs assignment a in state d
$Guard\,(g, d)$	p. 241	enforces guard g in state d
$\mathbb{D}$	p. 241	set of memory states sets with consistent support
$\mathbb{A}_V$	not. 19.7 p. 241	set of assignments with variables in V
$\mathbb{A}$	not. 19.7 p. 241	set of real assignments
$\mathcal{R}$	def. 19.2 p. 245	set of roles
$\mathscr{V}_i$	not. 19.8 p. 245	set of ghost variables at level i
$\mathscr{V}$	not. 19.8 p. 245	set of ghost variables
$\mathscr{E}$	not. 19.9 p. 245	set of ghost expressions
$\mathscr{C}$	not. 19.9 p. 245	set of ghost conditions
$\mathscr{A}$	not. 19.9 p. 245	set of ghost assignments (or directed constraints)
$\trianglelefteq$	def. 19.3 p. 245	ghost ordering
$\llparenthesis v \rrparenthesis$	def. 19.4 p. 246	semantics of ghost variable v
$\mathbb{U}_V$	def. 19.5 p. 248	set of constraints with variables in V
$\mathscr{U}$	def. 19.5 p. 248	set of ghost constraints
$[\![u]\!]$	p. 249	semantics of a ghost constraint
$\mathscr{G}$	not. 19.10 p. 251	set of constraint-DAGs

— Continued on following page —

Table A.8 – Notations of Part IV

Notation	**Defined**	**Meaning**
$\overset{\circ\!-}{g}$	not. 19.11 p. 251	source of constraint-DAG g
$\overset{-\!\circ}{g}$	not. 19.11 p. 251	sink of constraint-DAG g
$\langle\!\langle g\rangle\!\rangle_n$	def. 19.7 p. 251	semantics of node n in constraint-DAG g
$\langle\!\langle g\rangle\!\rangle$	def. 19.7 p. 251	semantics of constraint-DAG g
$\mathfrak{N}$	def. 20.1 p. 254	set of domain names
$\mathfrak{o}(r)$	def. 20.2 p. 254	owner of role r
$\Subset$	def. 20.2 p. 254	role and variable belonging relation
$\textsc{Alloc}(v, d)$	def. 20.3 p. 258	allocates variable v in abstract state d
$\textsc{Kill}(v, d)$	def. 20.3 p. 258	kills variable v in abstract state d
$\textsc{Assign}(a, d)$	def. 20.3 p. 258	performs assignment a in abstract state d
$\textsc{Guard}(g, d)$	def. 20.3 p. 258	enforces guard g in abstract state d
$\succ_n$	def. 20.4 p. 260	ghost constraint triggering relation in domain named n
$\langle\!\langle g\rangle\!\rangle^\sharp$	def. 20.5 p. 262	abstract semantics of constraint-DAG g
$\langle\!\!\mid(new, old, g)\mid\!\!\rangle^\sharp$	def. 20.6 p. 262	allocates variables in new, enforces g and kills variables in old
$\mathrm{MVar}(e)$	not. 20.2 p. 267	multiset of variables in e
$\curlyeqprec$	def. 20.8 p. 268	order on expressions used in $\precsim$
$\precsim$	def. 20.9 p. 269	order on constraints used in $\rightarrowtriangle$
$\mathrm{depth}(v)$	def. 20.10 p. 269	depth of variable v
$\rightarrowtriangle$	def. 20.11 p. 270	termination relation
$\mathrm{depth}(d)$	def. 20.12 p. 274	depth of state d
与	p. 282	set of possible values of a slice of a variable
卅	p. 283	set of possible variable slices
ǂ	p. 283	set of sliced variables
$\langle\!\cdot e\cdot\!\rangle$	p. 285	state of a slice of sliced expression e
与'	p. 287	set of possible values of a slice of an expression
卅'	p. 287	set of possible expression slices
ǂ'	p. 287	set of sliced expressions
$\widetilde{ǂ}'$	p. 287	set of modified sliced expressions
$\wr\, e\,\vert$	p. 287	cut on the left the modified sliced expression e
$\vert\, e\,\wr$	p. 288	cut on the right the modified sliced expression e
$\wr\, e\,\wr$	p. 288	cut both sides of the modified sliced expression e
$\circledast e\circledast$	p. 289	compressed sliced expression e

Appendix B

By-products

THE IMPLEMENTATION of the reduced product with ghost variables and assembly handling made several by-products as external tools or libraries, that are presented in this appendix. We do not mention libraries that are internal to ASTRÉE.

B.1 OColor

OCOLOR is an OCAML pretty printing library that benefit from semantics tags defined by OCAML's standard module `Format`. It has a public version available at [Che20b]. The purpose of this library is to emit ANSI escape sequences (see [ISO02]) to produce a nice and more readable output. It is about 4 kloc long and has no run-time dependency.

Among all ANSI escape sequences, we are mainly interested in Select Graphic Rendition (SGR) sequences. These sequences apply styles and are the most frequently used. In the following we will refer to them when writing "ANSI escape sequences". For completeness, we may note that there are other useful ANSI escape sequences, such as those that allow mouse integration, or fine cursor control[*]. ANSI escape sequences are interpreted completely linearly without memory: each sequence define a change in style that is applied on the current style state. ANSI sequences allow setting the foreground color, the background color, whether the text is underlined, doubly underlined or neither, whether the text is bold.... Some styles can be independently reverted, like foreground and background colors, blink, conceal, reverse video, underlined and overlined[†]. Other styles cannot be reverted, like faint, fraktur, framed, italic and encircled[‡]. When styles cannot be removed independently, we still may reset the current style to the default style, and apply the new formatting we want (that is, the old format, except what we want removed). But unlike LaTeX or HTML, there are no well-parenthesized tags: in LaTeX, writing `\textcolor{blue}{a\textcolor{red}{b}c}` produces abc. There is no

[*]These sequences are very useful to write text-based user interface (TUI), even if they are often hidden in an adequate library.

[†]Bold off is specified by ECMA-48 (see [ECM91]) as doubly underlined, and most shells seem to implement it this way.

[‡]And, in practice, bold.

equivalent using ANSI escape code: we can apply the sequence to color the “a” in blue, then a sequence to color the “b” in red, but then, there is no sequence to cancel the last color; to color the “c” in blue, we have to apply the sequence for blue foreground again. That is, we must remember how it was before applying the red style.

Most libraries using ANSI escape sequences simply reset the current style at the end. This prevents compositional use. For instance, in our example, “c” would get the default color (typically white on black terminals, and black on white terminals), because formatting “b” would reset the current style.

Thanks to `Format`’s semantics tags, OColor keeps a history of previously applied types. It works by storing mutable stacks in the closure of tag management functions. There is one stack for each kind of style: a stack for the foreground color, one for the background color, one for the underline status (simply underlined, doubly underlined or not underlined at all).... Each stack, is a pair of a first class module and its state, this allows each stack to be handled differently. For colors stack, when the closing tag is encountered, the stack is popped and the previous color is applied. For styles that cannot be removed independently, like faint, we reset the whole state, and we reapply the style described by the top of each stack.

With OColor, we can print abc by writing

```
Ocolor_format.printf "@{<blue>a@{<red>b@}c@}";
```

Many utility functions are provided as well as highly-configurable printers for common types such as Boolean, options, results, tuples, lists, arrays, iterable containers and iterable mappings. There are also some features to ease conditional pretty printing. Styles in semantic tags can be written in many ways, in particular, colors can be specified using 4-bit color names, RGB (red-green-blue) hexadecimal code, RGB decimal code, grayscale or X11 color name. We can also specify directly specific tags rather than names in the format string. We get the same result by writing

```
Ocolor_format.printf "%aa%ab%ac%a"
  pp_open_style Ocolor_types.(Fg (C4 blue))
  pp_open_style Ocolor_types.(Fg (C4 red))
  pp_close_style ()
  pp_close_style ();
```

Using OColor requires very few adaptations. Indeed, it is fully compatible with functions like `Format.fprintf`, it only needs to use formatters with semantic tag handling functions, thus, we only need to use OColor’s version of `printf`, `eprintf`, `asprintf` and `kasprintf`. We may also use `fprintf` with formatters defined in OColor. With default formatters, semantic tags are simply ignored.

OColor does not detect the current terminal but its capabilities can be configured. For instance, many modern terminals handle 24-bit colors, but some of them can only work with 8-bit or 4-bit colors. In such a case, OColor will replace colors with the closest color possible in the most expressive encoding available. The choice of the closest color is made according to the Euclidean distance in CIE[§] L*a*b* color space

[§] *French*: Commission internationale de l’éclairage, International Commission on Illumination (see [CIE19]) under CIE Standard Illuminant D_{65}, as it is made to be faithful to the human perception of colors.

This functionality is not only made to support outdated terminals, but also to adapt colors to the user's convenience. Indeed, while 24-bit colors are absolutely defined, 4-bit colors are just names, the real rendered color depend on the configuration of the terminal, allowing the user to choose its colors depending on the background. As a consequence, to find the closest 4-bit color, OColor should know terminal's configuration. It can be supplied manually, but several classical configurations (like the one used by gnome-terminal in Ubuntu) are ready to use.

OColor is very useful in Astrée where log files are very long and need to be more readable, and printers are almost always defined inductively. OColor is perfectly fitted to this task because, as long as each printer close all the tags it opened, sub-printers will not interfere with the previous formatting.

OColor also have a version for `Printf`, but because of the absence of semantic tags, it has no memory of the past, like most libraries using ANSI escape codes.

B.2 Epictetus

Epictetus stands for "Elegant Printer of Insanely Complex Tables Expressing Trees with Uneven Shapes". It is an OCaml library that provides functions to print tables in which columns may have arbitrarily many and deep sub-columns. The structure of each line is not necessarily the same, yet we want columns corresponding to the same position in the tree of columns to be aligned. It has a public version available at [Che20a]. The name Epictetus also recall the Greek philosopher Ἐπίκτητος who, as a Stoic, would have liked perfectly aligned tables. Epictetus is about 500 loc long and has no run-time dependency.

It was originally designed to made assembly rules generated by Amical (see section 9.4 "A more Concise Syntax for Assembling Rules" (page 128)) more readable, to be easily compared with Intel's manual. Afterward, it has been used to visualize many kinds of data, e.g. timing results.

Epictetus is also written to work nicely with OColor, or any use of semantics tags. Indeed, Epictetus needs to compute the length of each cell to align nicely columns. But OColor (among other), generate non-printable characters using semantics tags. Such characters should not be taken into account when computing the length of the code[¶]. Epictetus provides several ways to deduce the apparent lengths of strings to print.

B.3 Plato

Plato is an adaptation of relevant parts of Python's standard library to OCaml. It has a public version available at [Che20c]. Its name stands for "Python Library Adapted

[¶]Actually, we can note that `Format` ignores these characters with respect to formatting.

To OCaml". It is named after the Greek philosopher Πλάτων whose work is very long (as Python's standard library) and report the life and philosophy of Socrates, just like Plato brings a very handy library to OCaml's world. Plato is about 5 kloc long.

Initially, Plato was started to adapt into OCaml the Python's `pathlib` library, which is very convenient to work with filenames and files. We were especially interested in the normalization function that removes symbolic links from a path. As it was very helpful, other parts of Python's standard libraries were adapted, in particular `datetime`, `itertools`, `configparser` and their dependencies. Plato also provides functions on list, arrays and strings in place of Python's standard methods. In particular, the slice operator is implemented for all these types, with optimal asymptotic complexity each type allows.

For now, a very small part of Python's standard library is adapted to OCaml. Some parts do not make sense at all, like `importlib`, `gc` or `inspect`. Some libraries have already fair OCaml equivalent, like `re` or `json`. Other libraries can still be implemented when needed. An especially nice library to adapt would be `argparse` as it is very parameterizable and powerful, yet simple to use.

B.4 Socrates

Socrates is by far the largest by-product of Astrée development but also the most specific one. It is a very flexible testing tool that copes with all particularities of Astrée. Socrates stands for "Subtle Output Comparison for Reasonable and Automatic Test Execution for astreeS". It is written in Python and is a bit more than 10 kloc long. The name of this tool was chosen as a reference to the Socratic method: Socrates (Σωκρᾰ́της) asked questions until it manages to prove its interlocutor is wrong, often without explaining what is right (and, probably, without knowing it); a testing tool can only exhibit the tested program is wrong by asking it to answer many questions.

Testing Astrée is quite difficult. Since most parts are functors whose parameters have huge signature, unit testing with mocks is not realistic with existing tools. Socrates performs only functional tests: it runs analyses and compares the result with a standard.

Tests are stored in a DAG structure in JSON (JavaScript Object Notation) files. Each file describes nodes, whose children are in the current file or in a subdirectory[**]. Nodes in JSON files can also be leaves, which correspond to tests. Each leaf points to at least one C file to analyze. Each node of the DAG stores also configuration, it can be configuration of Astrée (like command-line options) or parameters for Socrates, like the timeout. The configuration with which we run an analysis is collected on the whole path from the root to the leaf. This structure allows each leaf to be accessed from multiple paths. For instance, all pure C test cases are run with assembly handling enabled and disabled. This is easy to achieve using a diamond at the top of the DAG, as shown in Figure B.1. All tests collected from the path on the left have assembly enabled,

[**]This is not a requirement, but a good practice.

while it is disabled when using the path of the right. One should be careful with such structures: by adding 3 nodes, we doubled the number of tests. Tests suite may have several roots which allows the user to easily select wanted tests, e.g. all tests, only mixed C-assembly tests or only pure C tests.

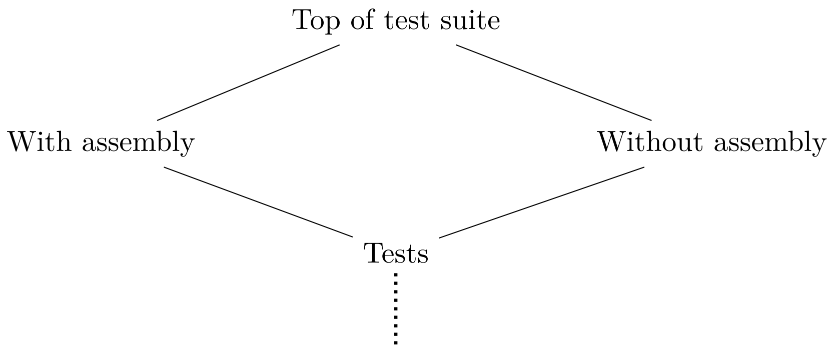


Figure B.1: A widget to duplicate tests

To check if a test is successful or not, we cannot compare the outputs: there are very unstable in general, especially during development, where there are many debug printing. Moreover, even with a single version of ASTRÉE, the output contains information on the environment (which are specific to each computer), execution time and starting date and time. In addition, alarms are not guaranteed to be issued in the same order, especially alarms about the same line of code. Logs are parsed to extract relevant data; the result is called summary. Then, we need to compare summaries meaningfully. For instance, the set of alarms should be the same, independently on the order. Summary comparison may also return non-fatal errors (i.e. warnings), e.g. when execution time of the test is very far from the standard (it may detect a major performance regression). To investigate errors more easily, standards are not reduced to the summary: we also keep the log file from which the standard was extracted. Also, this allows the user to regenerate summaries when we change the parser. Summaries are stored in JSON files, pointed by leaves of the DAG of tests. Each file contains a summary for each execution context, that is the number of summaries is at most the number of paths from the root[††].

In a legacy version of SOCRATES, configuration at each node was given as a list of command line options. This solution was deprecated since it has several limitations. Firstly, it is not always possible to override an option set before. Moreover, the analysis behavior does not depend directly on the command line options, but on the internal configuration of ASTRÉE resulting from these options. In particular, it changes if the semantics of an option change[‡‡] or the default value of a parameter is modified[§§]. This

[††]It is not necessarily equal since different paths may result in the same configuration.

[‡‡]This happens quite a lot during development. Typically, an option sets an integer parameter that has two uses. This parameter is split into two different ones, for each use, and command line options are updated accordingly.

[§§]For instance, during development, many debug flags are set by default, but should be disabled to lead to accepting standards that have been generated for another context, though the command line options are the same. The current solution used in SOCRATES describes the internal settings we expect in ASTRÉE; standards are identified by these internal settings. Thanks to a configuration file SOCRATES knows how to generate command line options to reach a given setting. In particular, this configuration file contains default values, so that no flag is generated if only the default value is specified for a parameter. Moreover, the language in this file is quite expressive and allows options to generate internally lists, sets, maps. It also handles command line options that are aliases for several other (with or without argument).

SOCRATES is also able to handle several versions and branches of ASTRÉE, so that it can be used with both ASTRÉEA and ASTRÉES on the same test suite. To do so, test cases and configuration options can be annotated with a branch (ASTRÉEA or ASTRÉES, for instance), and a complex condition on version number. Using these constraints, SOCRATES can exclude irrelevant tests and generate command-line arguments for different versions of ASTRÉE. In particular, it allows the semantics of a command line option to change between versions. This feature requires ASTRÉE to be able to give its own version and branch. The branch is an arbitrary string that does not contain the character `-`. Versions number follows the specification of PEP440[¶¶], which allows very expressive version numbers. Version specifiers are already expressive, but not easy enough to use. SOCRATES allows arbitrary logic propositions on version numbers, in which atoms are version specifiers as described in PEP440.

Changing the structure of the test suite is a major change, but it should be feasible. For instance, we explained how we started by using a test suite with command-line options, but now we specify internal settings we want. To perform such tasks, SOCRATES has a migrator framework to easily translate a test suite of an old version to a newer one. Even the migrator is made to be user-friendly as it does its best to perform migrations even indirectly, but with a minimal number of steps.

SOCRATES' main interface looks like a shell, with entry validation (with explicit error messages), advanced completion (fuzzy and context sensitive), persistent searchable history and common shortcuts. It is important SOCRATES stays alive between commands as it is stateful. For instance, we can walk on the DAG (using SOCRATES' `cd` command) to run a subset of tests. Also, when tests fail, results are in memory, and not written into files until the user decide to do so. It can print nice diffs between the standard and the actual output.... In such cases, to examine errors, update standards, and other more interactive tasks, SOCRATES provides a TUI. SOCRATES also has a non-interactive mode, but it is more limited as we cannot inspect errors as soon as SOCRATES is terminated. Its main purpose is to easily check a version is correct, typically using continuous integration.

To complete tests as fast as possible SOCRATES can detect the number of available cores on the current computer and run tests concurrently. It can also run tests using a

test in reasonable time, so we add an option. Later, when debug flags are disabled by default, we would like to remove the command line option otherwise, we simply accumulate a lot of such flags.

[¶¶]PEP 440 – Version Identification and Dependency Specification, `https://www.python.org/dev/peps/pep-0440/`

version of Astrée it finds on a remote host and which is securely deleted from the local host at the end. It is also able to send a local version of Astrée on a remote host and run tests there, which is interesting if the remote host is more powerful.

Socrates is also able to discover tests in folders and subfolders of its suite. It assists the user to create adequate JSON files, though, manual intervention is still needed to add some configuration. Socrates is aware of symbolic links, handles the gently, and, in particular, is protected against loops[***]. More generally, Socrates knows how to deal with most errors of the external world.

All these features (and many minor ones we skipped) make Socrates a very useful tool, pleasant to use. Since no restart is required when Astrée binary is changed or to select the native or bytecode version, it is common to keep Socrates running for days while developing. For now, it was designed for Astrée only. Hopefully, we believe it can be adapted to test a large variety of software, as long as they use files as input and the correctness of the test can be decided from the output. The main things to change are the parser, the summary comparator and the configuration file of command line options.

B.5 Virgil

Virgil is the more modest by-product of Astrée. Its purpose is to update virtual machine disk images, which are used for continuous integration. Virgil stands for "Virtualbox Image Renewal for Gitlab Integration for the Lazy". It is named after the ancient Roman poet Publius Vergilius Maro, who is also Dante's guide through Hell and Purgatory in the *Divine Comedy*; indeed, disk images are hellish to manage manually, and we are happy to have a guide through this ordeal. Virgil is written in Python and is about 200 loc long. Even small tools can be most supremely helpful.

[***]It should be also protected against loops with hard links, but it is not extensively tested.

RÉSUMÉ

Cette thèse est consacrée à l'analyse de logiciels de bas niveau, tels que les systèmes d'exploitation, par interprétation abstraite. L'analyse des OS est une question importante pour garantir la sûreté des systèmes logiciels puisqu'ils forment le niveau immédiatement au dessus du matériel et que toutes les tâches applicatives dépendent d'eux. Pour des applications critiques, on veut s'assurer que l'OS ne plante pas, mais aussi qu'il assure l'isolation des programmes, de sorte qu'un programme dont la fiabilité n'a pas été établie ne puisse perturber un programme de confiance. L'analyse de ce genre de programmes soulève des problèmes spécifiques. Cela provient du fait que les OS doivent contrôler le matériel avec des opérations qui n'ont pas de sens dans un programme ordinaire. De plus, comme les fonctionnalités matérielles sont en dehors du for du C, le code source contient des blocs de code assembleur mêlés au C. Ce sont les deux axes de cette thèse : gérer les mélanges de C et d'assembleur, et abstraire finement les opérations spécifiques aux logiciels de bas niveau. Ce travail est guidé par l'analyse d'un cas d'étude d'un partenaire industriel, ce qui a nécessité l'implémentation des méthodes proposées dans l'analyseur statique Astrée. La première partie s'intéresse à la formalisation d'un langage mélangeant des modèles simplifiés du C et de l'assembleur, depuis la syntaxe jusqu'à la sémantique. Cette spécification est importante pour définir ce qui est légal et ce qui constitue une erreur, tout en tenant compte de la complexité des interactions du C et de l'assembleur, tant en termes de données que de flot de contrôle. La seconde partie est une introduction sommaire à l'interprétation abstraite qui se limite à ce qui est utile par la suite. La troisième partie propose une abstraction de la sémantique des mélanges de C et d'assembleur. Il s'agit en fait d'une collection d'abstractions paramétriques qui gèrent chacun des aspects de cette sémantique. La quatrième partie s'intéresse à l'abstraction des opérations spécifiques aux logiciels de bas niveau. Les propriétés d'intérêt peuvent être facilement prouvées à l'aide de variables fantômes, mais pour des raisons techniques, il est difficile de concevoir un produit réduit de domaines abstraits qui supporte une gestion satisfaisante des variables fantômes. Cette partie construit un tel cadre très général ainsi que des domaines qui permettent de résoudre beaucoup de problèmes dont le nôtre. L'ultime partie présente quelques propriétés à prouver pour garantir l'isolation des programmes, qui n'ont pas été traitées car elles posent de nouvelles et complexes questions. On donne aussi quelques propositions d'amélioration du produit de domaines avec variables fantômes introduit dans la partie précédente, tant en termes de fonctionnalités que de performances.

MOTS CLÉS

Sémantique, Analyse statique, Analyse de programmes, Interprétation abstraite, Systèmes embarqués critiques, Code C, Assembleur, x86, Assembleur inline, Domaine abstrait, Produit réduit, Variables fantômes

ABSTRACT

This thesis is dedicated to the analysis of low-level software, like operating systems, by abstract interpretation. Analyzing OSes is a crucial issue to guarantee the safety of software systems since they are the layer immediately above the hardware and that all applicative tasks rely on them. For critical applications, we want to prove that the OS does not crash, and that it ensures the isolation of programs, so that an untrusted program cannot disrupt a trusted one. The analysis of this kind of programs raises specific issues. This is because OSes must control hardware using instructions that are meaningless in ordinary programs. In addition, because hardware features are outside the scope of C, source code includes assembly blocks mixed with C code. These are the two main axes in this thesis: handling mixed C and assembly, and precise abstraction of instructions that are specific to low-level software. This work is motivated by the analysis of a case study emanating from an industrial partner, which required the implementation of proposed methods in the static analyzer Astrée. The first part is about the formalization of a language mixing simplified models of C and assembly, from syntax to semantics. This specification is crucial to define what is legal and what is a bug, while taking into account the intricacy of interactions of C and assembly, in terms of data flow and control flow. The second part is a short introduction to abstract interpretation focusing on what is useful thereafter. The third part proposes an abstraction of the semantics of mixed C and assembly. This is actually a series of parametric abstractions handling each aspect of the semantics. The fourth part is interested in the question of the abstraction of instructions specific to low-level software. Interest properties can easily be proven using ghost variables, but because of technical reasons, it is difficult to design a reduced product of abstract domains that allows a satisfactory handling of ghost variables. This part builds such a general framework with domains that allow us to solve our problem and many others. The final part details properties to prove in order to guarantee isolation of programs that have not been treated since they raise many complicated questions. We also give some suggestions to improve the product of domains with ghost variables introduced in the previous part, in terms of features and performances.

KEYWORDS

Semantics, Static Analysis, Program Analysis, Abstract Interpretation, Critical Embedded Systems, C Code, Assembly, x86, Inline Assembly, Abstract Domains, Reduced Product, Ghost Variables