MÉMOIRE D'HABILITATION À DIRIGER DES RECHERCHES

présentée à

l'ÉCOLE NORMALE SUPÉRIEURE

Static analysis by abstract interpretation of concurrent programs

Analyse statique par interprétation abstraite de programmes concurrents

---

Antoine MINÉ

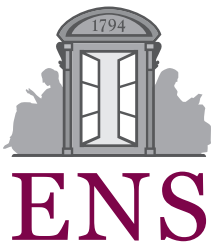
28 mai 2013

Rapporteurs:

Roberto Giacobazzi	*Università di Verona, Italie*
Nicolas Halbwachs	*Vérimag, France*
Manuel Hermenegildo	*IMDEA Software Institute & Technical University of Madrid, Espagne*

Examinateurs:

Ahmed Bouajjani	*Université Paris Diderot (Paris 7), France*
Patrick Cousot	*École normale supérieure, France & New York University, U.S.A.*
Éric Goubault	*Commissariat à l'énergie atomique, France*
Marc Pouzet	*École normale supérieure, France*



École Normale Supérieure

Département d'Informatique

Overview

This report presents the bulk of my research work from the completion of my PhD, in late 2004, until the present day. It is submitted in partial fulfillment of the requirements for the French qualification of *habilité à diriger des recherches* (accreditation to supervise research). This report is a brief synthesis of several results published in distinct articles. Due to page limitation some technical material (including proofs and raw experimental data) are omitted; the interested reader is invited to consult the cited articles for more information. Some of the contributions presented here were purely my own, others were pursued in a team work within the Abstraction group at ENS and, finally, some contributions are that of a PhD student that I helped supervise.

The overall aim of my research is the development of mathematically sound and practically efficient methods to check the correctness of computer software. Efficiency is achieved using approximations, while soundness is guaranteed by employing over-approximations of program behaviors. My research is grounded in the theory of abstract interpretation, a powerful mathematical framework facilitating the development, use, comparison, and composition of approximations in a sound way. I am mainly interested in developing new reusable abstraction components (so called abstract domains) that can be readily implemented, and in using them to develop static analyzers, which are computer programs able to check automatically the safety of software. While my early research was focused on inferring the values of variables in sequential programs, my current interest and latest results concern the analysis of concurrent programs, hence the title of this report.

The first two chapters of this report constitute an introduction. The first chapter is an informal introduction to the problem at hand, existing solutions, their strengths and their shortcomings. The second chapter presents prior mathematical and formal tools on which our work is based, including some notions of abstract interpretation, a description of existing abstract domains and their application to the static analysis of sequential programs. It also recalls some results I obtained during my PhD and that will be useful in the rest of the report. The subsequent chapters describe the work I performed after completing my PhD.

The third chapter is devoted to aspects of static analyzers that are specific to concurrent programs. This topic of personal research has led to the construction of a generic analysis method for concurrent programs, parametrized by the choice of abstract domains. The method is based on a notion of "interference" that abstracts thread interleavings in a sound way in order to achieve a thread-modular analysis. It is related to Jones' rely-guarantee proof method, and we make this connection formal in a first part. Then, we present an interference-based analysis in big-step form that is efficient and easy to implement. In a third part, we study the interaction of the analysis with weakly consistent memory models, found in modern processors and language specifications. The last part discusses how to adapt the analysis to exploit some properties of the scheduling (such as the use of real-time thread priorities and synchronization primitives).

The fourth and fifth chapters are devoted to the design of abstract domains. Although some of them found their application in the analysis of concurrent programs, they are actually generic and could be exploited in any kind of static analysis, for concurrent or sequential programs. The fourth chapter concerns numeric domains to infer linear equality and inequality relations, developed in collaboration with Liqian Chen while he visited ENS during his PhD. The initial motivation was to revise the classic polyhedra domain using sound floating-point arithmetic to improve its efficiency, but it unexpectedly yielded the construction of new, more expressive domains based on interval affine relations, which we also present. The fifth chapter concerns the abstraction of realistic data-types as found in the C programming language, including machine integers, floating-point numbers, and structured blocks of memory (structs, unions, and arrays). We design abstractions that are aware of the low-level memory representation of data-types, to support the analysis of programs that rely on assumptions about this representation (such as "type punning" constructions in C). The need for such abstractions was motivated by the analysis, in the scope of the Astrée and AstréeA static analyzers, of industrial C programs, where such low-level constructions are widespread.

The sixth chapter is devoted to the application of these methods to the design of static analyzer tools. It mainly reports on my experience with the Astrée analyzer, a team effort initiated during my PhD in 2001 that extended well beyond it and culminated in its industrialization in 2009. Much of my theoretical work could find some application in Astrée, as Astrée fuelled my research with not only practical problems to solve, but also concrete problems that could only be overcome by theoretical developments. This part also reports my own ongoing effort on AstréeA, an extension of Astrée that incorporates the interference abstraction presented above and aims at proving the absence of run-time error in concurrent embedded programs (while Astrée only considers synchronous programs). Additionally, this chapter presents the Apron abstract domain library, another, more academic, team effort, which aims at encouraging the research on numeric abstract domains.

The report concludes with some perspectives for future researches.

Résumé

Ce mémoire d'habilitation résume la majeure partie de mes recherches, depuis la fin de mon doctorat, fin 2004, jusqu'à aujourd'hui. Les travaux résumés dans ce mémoire ont par ailleurs été publiés dans plusieurs journaux et actes de conférences. Par manque de place, les développements les plus techniques sont omis (c'est en particulier le cas des preuves et des tables de résultats expérimentaux) ; le lecteur intéressé est invité à les consulter dans les articles cités. Certains des résultats présentés ici sont les miens propres, tandis que d'autres sont issus d'un travail en équipe au sein du groupe Abstraction, et d'autres enfin ont été obtenus par un doctorant que j'ai co-encadré.

Le but essentiel de mes recherches est le développement de méthodes fondées sur des bases mathématiques et performantes en pratique pour s'assurer de la correction des logiciels. J'utilise des approximations pour permettre une bonne performance, tandis que la validité des résultats est garantie par l'emploi exclusif de sur-approximations des ensembles des comportements des programmes. Ma recherche est basée sur l'interprétation abstraite, une théorie très puissante des approximations de sémantiques permettant aisément de les développer, les comparer, les combiner. Je m'emploie en particulier au développement de nouveaux composants réutilisables d'abstraction, les domaines abstraits, qui sont directement implantables en machine, ainsi qu'à leur utilisation au sein d'analyseurs statiques, qui sont des outils de vérification automatique de programmes. Mes premières recherches concernaient l'inférence de propriétés numériques de programmes séquentiels, tandis que mes recherches actuelles se tournent vers l'analyse de programmes concurrents, d'où le titre de ce mémoire.

Les deux premiers chapitres de ce mémoire constituent une introduction, tandis que les suivants présentent mon travail d'habilitation proprement dit. Le premier chapitre est une introduction informelle à la problématique de l'analyse de programmes, aux méthodes existantes, leurs forces et leurs faiblesses. Le deuxième chapitre présente de manière formelle les outils dont nous aurons besoin par la suite : les bases de l'interprétation abstraite, quelques domaines abstraits existants et la construction d'analyses statiques par interprétation abstraite, ainsi que quelques résultats utiles que j'ai obtenu en doctorat.

Le troisième chapitre est consacré aux aspects spécifiques de l'analyse de programmes concurrents. Cette recherche, très personnelle, a abouti à la construction d'une méthode d'analyse de programmes concurrents, paramétrée par le choix de domaines abstraits, et basée sur une notion d'interférence abstrayant les interactions entre *threads*. Ainsi, l'analyse construite est modulaire pour les *threads*. Cette méthode est reliée aux preuves *rely-guarantee* proposées par Jones, ce que nous montrons formellement dans une première partie. Nous construisons ensuite une analyse à grands pas basée sur les interférences, efficace et facile à implanter. Les deux dernière parties étudient les liens entre l'analyse et les modèles mémoires faiblement cohérents (désormais incontournables) ainsi que le raffinement de l'analyse pour tenir compte des propriétés spécifiques des ordonnanceurs temps-réels (nous étudions en particulier l'effet des priorités des *threads* et l'emploi d'objets de synchronisation).

Le quatrième et le cinquième chapitre sont consacrés à la constructions de domaines abstraits. Ceux-ci ne sont pas spécifiquement liés au problème de la concurrence ; ils sont utiles à l'analyse de tous programmes, séquentiels comme concurrents. Le chapitre 4 étudie des domaines numériques inférant des égalités et inégalités affines, développés en collaboration avec Liqian Chen, alors doctorant en visite à l'ENS. La motivation première était l'emploi de nombres à virgule flottante afin d'améliorer l'efficacité du domaine des polyèdres, mais ces travaux ont également débouché sur la découverte de nouveaux domaines, basés sur les relations affines à coefficients intervalles, que nous présentons également. Le chapitre 5 étudie les abstractions de types de données réalistes, comme ceux rencontrés dans le langage C : les entiers machines, les nombres à virgule flottante, et les blocs structurés (tableaux, structures, unions). Nos abstractions modélisent finement les détails de l'encodage en mémoire des données afin de permettre l'analyse de programmes qui en dépendent (par exemple, ceux utilisant le *type-punning*). Ces abstractions sont motivées par nos expériences d'analyses, avec les outils Astrée et AstréeA, de programmes C industriels ; ceux-ci employant fréquemment ce type de constructions de bas niveau.

Le sixième chapitre est consacré aux applications des méthodes présentées ci-dessus à la construction d'outils d'analyse statique. Il décrit en particulier mon travail sur l'outil Astrée que j'ai co-développé avec l'équipe Abstraction pendant et après mon doctorat, et qui a été industrialisé en 2009. Mes résultats théoriques et appliqués ont contribué au succès d'Astrée, tandis que celui-ci m'a fourni de nouveaux thèmes de recherches, sous la forme de problèmes concrets dont la résolution n'a pu se faire que grâce à des développements théoriques. Ce chapitre décrit également AstréeA, une extension d'Astrée utilisant l'abstraction d'interférences proposée plus haut pour l'analyse de programmes concurrents (Astrée étant limité aux programmes séquentiels). Il décrit également Apron, une bibliothèque de domaines abstraits numériques que j'ai co-développée. Il s'agit d'un outil plus académique, dont le but est d'encourager la recherche sur les domaines numériques abstraits.

Le mémoire se conclue par quelques perspectives sur des recherches futures.

Contents

Chapter 1

Introduction

In this short introductory chapter, we explain informally the meaning of our title "static analysis by abstract interpretation of concurrent programs." We expose the problem at hand, program verification, and give an overview of existing methods to solve it. We recall the concept of static analysis by abstract interpretation. Finally, we discuss the specific challenges related to the verification of concurrent programs.

1.1 Program verification

Programming is an error-prone activity and "bugs" (programming errors) are pervasive, resulting in spectacular failures (such as the Ariane failure in 1996 [Lio96]) and, more generally, economic losses (NIST evaluated their annual cost to the U.S. industry at $59.5 billion in 2002 [NIS02]). While it might seem acceptable in some cases to ship potentially erroneous programs and rely on regular updates to correct them, this is not the case for embedded software, which are often mission critical and cannot be corrected during missions.

Testing. The most widespread (and in many cases the only) method used to ensure the quality of software is testing. Many testing methods exist (black-box and white-box testing, unit and integration testing, etc.); all consist in executing parts or the whole of the program with selected or random inputs in a controlled environment, while monitoring its execution or its output. A variant is dynamic analysis, where an instrumented version of the program with extra checks is executed, so as to detect errors earlier, more reliably, or to detect errors having a non-deterministic but not always fatal outcome (such as memory errors [NS07]). Achieving an acceptable level of confidence with testing is generally costly ([WM11] reports that tests account for as much as 50% of the cost of developing software-based systems) and, even then, testing cannot completely eliminate bugs [NIS02].

Formal methods. Unlike testing, formal methods employ mathematical and logical tools to reason on the program itself, at compile-time. As such, they can prove without ambiguity the correctness of programs (or at least, clearly express what is proved and what is not) before they are run: these methods are sound. The idea of formally discussing about programs dates back from the early history of computer science: program proofs and invariants are attributed to Floyd [Flo67] and Hoare [Hoa69] in the late 60s, but may be latent in the work of Turing in the late 40s [Tur49] (as reported by Morris and Jones [MJ84]). The lack of automation severely hindered early efforts but, with the progress of both computers and formal manipulation software, there is, according to Hoare [Hoa03], some hope to design a "verifying compiler that guarantees correctness of a program before running it" (although this hope should be tempered by the accompanying increase in the complexity of the software to verify). Current methods can be classified into three categories [CC10]:

- **deductive methods** employ proof assistants (such as Coq [BC04]) or theorem provers (such as PVS [ORS92]); they rely on the user to provide the inductive invariants needed in the proof, and sometimes to interactively direct the proof itself;
- **model checking** [CES86] explores exhaustively and automatically finite models of programs; a per-program user intervention is required beforehand to abstract programs with an infinite or large state space into such models;
- **static analyses** analyze directly and without user intervention the source code at some level of abstraction; due to decidability and efficiency concerns, the abstraction is incomplete and can miss properties, resulting in false alarms (a.k.a. false positive, i.e., correct programs reported as incorrect) but never false negative (so that programs reported as correct are indeed correct despite the approximation).

In addition to these sound methods, we must also mention the use of formal tools in unsound contexts. Some versions of model checking perform a partial exploration of infinite or very large models (as in bounded model checking [BCCZ99]), or of infinite sequences of finite models (as in counter-example guided abstract refinement [CGJ$^+$00]). Another example is symbolic execution [Kin76], which executes the program on a symbolic abstract domain of properties, but on a single (finite) program path at a time, and must be aborted after a finite number (out of the generally infinite set) of paths have been investigated. As with testing, these unsound methods can miss errors as they rarely explore the set of all possible executions.

Sound static analysis. Our work focuses on sound static analysis. Due to a low precision, early static analyses have been applied mostly, and with some success, to non-critical domains such as optimizing compilers where speed and automation are more important than precision (a missed property results at worse in disabling some valid optimization, for a slight cost in efficiency). However, by carefully designing the abstraction used in the analysis, it becomes possible to infer properties related to the correctness of programs with no or few false alarms. This is the case, for instance, for Astrée [BCC$^+$03], a static analyzer that checks for the absence of run-time error

(such as arithmetic or memory overflows) in embedded synchronous C programs. Such an analysis does not require much user intervention: the correctness conditions are part of the programming language semantics (and not externally-provided program-specific conditions), the analysis is performed on the source code (and not a hand-crafted model) and automatically (not interactively). It is thus very attractive in an industrial context [DS07], where it can be operated by engineers with a limited knowledge of formal methods.

Astrée is specialized, by its choice of abstractions, to a class of properties and an (infinite) class of programs: it cannot express arbitrary program verification conditions and might perform poorly in terms of efficiency and false alarm rate on some programs. On its intended targets, however, Astrée scales up to large programs (one million lines or more) with a good precision (few or no false alarm).

We have participated to the design and implementation of Astrée, and several results described in this report were integrated into Astrée. Chapter 6, which is devoted to applications, reports our experience with Astrée.

A major promise of abstract interpretation is that more complex properties, generally thought to be out of the scope of static analysis, can nevertheless be tackled by designing adequate abstractions (including, for instance, temporal properties [Mas02] traditionally handled by model checking, and proofs of functional correctness [CCM10] traditionally handled with user-assisted theorem provers). In this work, we stay modest and focus on relatively simple properties: mainly discovering invariants on numeric program variables. Such properties are nevertheless challenging (as they are undecidable) and useful in practice (as they are sufficient to prove the absence of many kinds of run-time errors).

1.2 Abstract interpretation

Abstract interpretation is a very general theory of the approximation of program semantics, introduced by Patrick Cousot and Radhia Cousot in the late 70s [CC77]. It stems from the observation that, while there exists a wide variety of program semantics, they can be uniformly described as fixpoints of operators in partially ordered structures. This observation extends to the formal methods used to ensure the correctness of programs, including proof methods, model checking, type checking, type inference, and semantic-based static analysis. Having expressed seemingly unrelated semantics in a uniform framework, it becomes possible to compare them in term of the amount of information they carry (understood as the set of program properties they can express). Abstract interpretation is thus a unifying force in formal program semantics.

Example 1.2.1. Big-step semantics model programs as input-output relations, forgetting the history of the computations modeled by small-step operational semantics. The latter can express properties on the length (number of steps) of computations while the former cannot: it is an abstraction [Cou02].

End of example.

Additionally, abstract interpretation presents semantics in a constructive form (often as limits of finite or, possibly uncountable, transfinite iterations). It expresses properties as a function of programs, which opens the way to property inference. This is in contrast to deductive methods, which can only verify statements provided externally by the user.

Example 1.2.2. In [CC84], Cousot ant Cousot present a constructive version of Owicki–Gries–Lamport proof method for parallel programs [OG76, Lam77] and derive static analyzers by abstraction. In Sec. 3.2, we will apply the same method to Jones' rely–guarantee proof method [Jon81].
End of example.

According to Rice [Ric53] all non-trivial program properties are undecidable. Even in constructive form, the semantics that express them cannot always be computed by a program in finite time. Abstract interpretation provides a systematic method to derive computable abstract approximate semantics:

- A first step is to choose a level of abstraction. The set of concrete semantic objects is replaced with a (partially ordered) set of abstract ones carrying less information. Ideally the abstraction forgets all the properties we do not care about (and properties that are not necessary to prove those we care about).

- Operators on the concrete world are then (systematically) mapped to operators on the abstract one. As even abstract operators may be too complex, it is sometimes useful, for the sake of efficiency, to over-approximate them (the abstract partial order modeling the relative precision of properties and operators).

- Concrete fixpoints of operators are replaced with fixpoints of abstract operators, generally approximated by iteration with extrapolation to ensure termination in finite time even when the abstract partial order has infinite chains [CC92b].

One fundamental application of abstract interpretation is the derivation of static analyzers that are, by construction, sound: any property proved in the abstract also holds in the original, concrete semantics. The abstract interpretation methodology helps tremendously on the semantic aspects (i.e., what is computed). Constructing an effective analysis additionally requires algorithms and data-structures (i.e., how it is computed), which are generally borrowed from other fields in mathematics and computer science.

Example 1.2.3. The polyhedra domain, introduced by Cousot and Halbwachs in [CH78], abstracts a set of points in a vector space as a polyhedron that encloses them. One way of implementing it is through linear programming. We describe this domain in Sec. 2.4.2 and extend it in Chap. 4.
End of example.

Abstract interpretation also studies the abstractions for themselves. It states which desirable properties abstractions should possess, if possible (such as being a Moore family, enjoying Galois connections, being complete, etc.). It also studies operators to manipulate and combine them (such as reduced products, completions, etc.). This encourages a modular approach to abstraction, where a set of abstract values and atomic abstract operators are bundled into a reusable building block, called an abstract domain.

We present Abstract interpretation formally in Chap. 2 and recall its main results, with a special focus on the design of static analyses for numeric properties, illustrated on an idealized language on real numbers. This introductory chapter recalls a few classic numeric abstract domains, while Chaps. 4

and 5 are devoted to the construction of new abstract domains. More precisely, Chap. 4 presents variations and extensions on the classic polyhedra domain, while Chap. 5 introduces domains adapted to more realistic data-types found in actual programming languages (such as machine integers, floating-point numbers, and structured data). The design of Apron, a library of numeric abstract domains, is described in Chap. 6.

1.3 Concurrent programs

Concurrent programming consists in designing software as collections of interacting computing processes, each following its flow of instructions. This is in contrast to sequential programs, i.e., executing a single flow of instructions. The processes of a concurrent program may run in parallel on different execution units (processors or cores) of a computer or on different computers, or be scheduled on a single processor through time-slicing, or a combination of these methods. The use of concurrent programming is not new, dating from the work by Dijkstra in the 60s [Dij65]. Since the mid-2000s and the advent of consumer multi-core computers, the development of concurrent programs has intensified: exploiting the parallelism in today's computers is considered the main (if only) way to improve the performance of software [Sut05].

Even without true parallel execution, some software benefit from a decomposition into largely independent processes. This is the case for instance for web-servers, where each request is handled by a distinct process executing a protocol instance, or for event-driven applications, where processes wait for inputs on different channels without inhibiting the progress of processes computing outputs. Concurrent programming has also entered the embedded critical world. For instance, Integrated Modular Avionics (IMA) [WW07] suggests transitioning, in avionic applications, from networks of processors executing a single task each and communicating on a bus into single processors executing many concurrent tasks communicating in a shared memory. Reducing the number of hardware components (buses and processors) has clear benefits in terms of cost, dependability, and scalability; however, it results in an increase in software complexity, and so, software verification cost.

Concurrent programming is now an integrated part of many programming languages (including object-oriented and functional languages) and many models exist to support it (examples include shared memory, message-passing, and transactional memory). In this work, we focus on low-level concurrency. We thus consider simple imperative C-like languages, ignoring issues related to objects, higher-order constructions, and focusing on the thread model, where processes execute in a shared memory. This model is pervasive in embedded concurrent software. Some parts of our work will consider additional restrictions, such as the use of a fixed number of processes and a real-time scheduler, which is motivated by our application to the verification of embedded avionic software.

Verification. The major drawback of concurrent programs is that they are hard to design, and hard to verify, even more so than sequential ones. Even a seemingly simple problem, such as mutual exclusion, can be difficult to solve correctly (an early example is given by Dijkstra [Dij65]). Executing a concurrent program is (in first approximation, using Lamport's sequential consistency model [Lam78]) achieved by interleaving the execution of its processes, according to some scheduler algorithm. Schedulers are highly non-deterministic, resulting in a combinatorial explosion of the set of possible executions. Testing and symbolic execution perform poorly as they rely on sampling finite executions or program paths: they can explore a tiny fraction of the large execution space while errors (such as data-races) often appear only in difficult-to-reach corner cases. Even the set of possible program configurations grows tremendously as each process features its own control space and local variables, so that model checking, which employs an exhaustive state-space exploration, also has difficulties scaling up. Unsound partial exploration techniques have thus been proposed, such as context-bounded model checking [QR05] which only allows a finite (generally small) number of context switches.

This verification problem is further complicated by the advent of weakly consistent memory models [ABBM10]. These execution models take into account the various hardware and software optimisations that are present in today's computer, such as non-coherent caches and out-of-order execution units. They exhibit executions that do not obey Lamport's model of sequential consistency. In order to be of any use, program verification must be sound with respect to these new execution models.

Another complexity added by concurrency is the emergence of new kinds of programming errors, that cannot occur in sequential programs:

- **data-races** occur when two processes simultaneously access the same memory location and one access at least is a write;
- **deadlock** is a situation where a subset of processes wait for each other in a circular fashion, thus blocking indefinitely all the concerned processes;
- **livelocks** are similar, but processes execute without making any progress (e.g., busy waiting) instead of blocking;
- **starvation** occurs when a process is indefinitely denied a resource, which is held by a process or passed along a set of conspiring processes.

Our focus is on sound static analysis with the intend to scale up to large programs. We will side-step the combinatorial explosion of executions by employing thread-modular techniques. Ideally, the analysis of a program should be reduced to the independent analysis of each of his processes. Note that employing existing sequential program static analyses on each process ignores their interaction, and is thus not sound. We will however show that a sound analysis can be constructed with only minor modifications to a sequential process analysis. The resulting analysis is almost as efficient as for sequential programs. Our focus is on (mostly numeric) invariant inference for concurrent programs. We will be able to prove invariance properties, including the absence of run-time error, of data-race, and of deadlock. However, the absence of livelock and starvation belongs to the class of liveness properties [LS85], which cannot be expressed with mere invariants and remain out of reach for our analysis. Our design is described, on the theoretical level, in Chap. 3. Its application to the construction of the AstréeA static analyzer, an extension to Astrée, is described in Chap. 6.

Chapter 2

Background

This chapter introduces formally notions and notations, and recalls existing results that are at the foundation of our work and will serve in subsequent chapters. We provide a short overview of abstract interpretation, focusing on its application to the design of sound static analyses. We also present a simple numeric sequential programming language, its semantics, and its static analysis. Chapter 3 will illustrate our concurrent program analysis method on an multi-thread extension of this language. Finally, we present several classic abstractions that parametrize sequential and concurrent analyses. Chapters 4 and 5 will present novel variants and extensions of these domains.

2.1 Notations

We introduce briefly the standard notations we use, which are drawn from various fields of mathematics and computer science. An index of all the notations introduced here and later, as well as an index of all notions, are available in the Appendix.

Partial orders. A *partially ordered set* $(A, \sqsubseteq)$ is a set A equipped with a binary reflexive, transitive, anti-symmetric relation $\sqsubseteq$. When it exists, the *least upper bound* (also called *join*) of a pair of elements $a, b \in A$ is denoted $a \sqcup b$, and its *greatest lower bound* (also called *meet*) is denoted $a \sqcap b$. Note that, when they exist, joins and meets are unique. A *lattice* $(A, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is a partially ordered set with a *least element* $\bot$ and a *greatest element* $\top$ in A, and a least upper bound $\sqcup$ and a greatest lower bound $\sqcap$ for every pair of elements in A. A lattice is *complete* when joins and meets exist for sets of arbitrary size; we denote the join and meet of $S \subseteq A$ respectively as $\sqcup S$ and $\sqcap S$.[1]

Example 2.1.1. A useful example of complete lattice is the powerset $(\mathcal{P}(X), \subseteq, \emptyset, X, \cup, \cap)$ of an arbitrary set X.
End of example.

A *complete partial order* is a partial order $(A, \sqsubseteq)$ such that, for any $X \subseteq A$, if every pair of elements in X has a least upper bound in X, then X has a least upper bound in A.

Partial orders $(A_1, \sqsubseteq_1)$, $(A_2, \sqsubseteq_2)$ can be combined element-wise: we define $\langle a_1, a_2\rangle \sqsubseteq \langle a'_1, a'_2\rangle \overset{\text{def}}{\iff} a_1 \sqsubseteq_1 a'_1 \wedge a_2 \sqsubseteq_2 a'_2$. The same holds for complete partial orders and (complete) lattices: $\bot \overset{\text{def}}{=} \langle \bot_1, \bot_2\rangle$, $\top \overset{\text{def}}{=} \langle \top_1, \top_2\rangle$, $\langle a_1, a_2\rangle \sqcup \langle a'_1, a'_2\rangle \overset{\text{def}}{=} \langle a_1 \sqcup_1 a'_1, a_2 \sqcup_2 a'_2\rangle$, and $\langle a_1, a_2\rangle \sqcap \langle a'_1, a'_2\rangle \overset{\text{def}}{=} \langle a_1 \sqcap_1 a'_1, a_2 \sqcap_2 a'_2\rangle$. Likewise, $(A, \sqsubseteq)$ extends element-wise to functions from arbitrary sets X to A: $f \sqsubseteq g \overset{\text{def}}{\iff} \forall x \in X : f(x) \sqsubseteq g(x)$, and similarly for complete partial orders and (complete) lattices.

[1] A more economical definition of complete lattices is: a partial order $(A, \sqsubseteq)$ with arbitrary joins. The other lattice operators can be derived from the join as: $\bot \overset{\text{def}}{=} \sqcup \emptyset$, $\top \overset{\text{def}}{=} \sqcup A$, and $\sqcap X \overset{\text{def}}{=} \sqcup \{ y \in A \mid \forall x \in X : y \sqsubseteq x \}$.

Functions. Given two sets A and B, we denote as $A \to B$ the set of functions from A (called the *codomain*) to B (called the *domain*). We often use the *lambda notation* $\lambda x \in A.\, f(x)$, or more concisely $\lambda x.\, f(x)$, to denote functions. If f is a function, then $f[x \mapsto v]$ is the function that maps x to v and other elements $y \neq x$ to $f(y)$; its domain is that of f plus x. Likewise $f[\forall x \in X : x \mapsto g(x)]$ maps elements $x \in X$ to $g(x)$ and other elements $y \notin X$ to $f(y)$. We will use the notation $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ to define a function in extension from scratch. When $A' \subseteq A$, $f_{|_{A'}}$ denotes the restriction of $f \in A \to B$ to a function in $A' \to B$.

When $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ are partial orders, then $f \in A \to B$ is *monotonic* if $\forall a, a' \in A : a \sqsubseteq_A a' \implies f(a) \sqsubseteq_B f(a')$. It is a *join-morphism* if, for any $X \subseteq A$, if $\sqcup_A X$ exists, then so does $\sqcup_B \{ f(x) \mid x \in X \}$ and $f(\sqcup_A X) = \sqcup_B \{ f(x) \mid x \in X \}$. This implies, in particular, $f(\bot_A) = \bot_B$.

Dependent types. Given a set A and a family $(B_a)_{a \in A}$ of sets indexed by A, we denote as $\Pi a{:}A.B_a$ the set of functions f from A to $\cup_{a \in A} B_a$ such that $\forall a \in A : f(a) \in B_a$. This generalizes function spaces $A \to B$ to the case where the domain B can be different for every element of the codomain A.

Fixpoints. A *fixpoint* of a function $f \in A \to A$ is any element $a \in A$ such that $f(a) = a$. When $a \sqsubseteq f(a)$, we say that a is a *pre-fixpoint* while, when $f(a) \sqsubseteq a$, a is said to be a *post-fixpoint*. We denote as $\mathrm{lfp}\, f$ the *least fixpoint* of f, when it exists. Moreover, $\mathrm{lfp}_a\, f$ is the least fixpoint of f greater than or equal to a.

Semantics. We denote *semantic functions* with double brackets, as in $\mathbb{X}[\![y]\!]$, where y is a syntactic object and $\mathbb{X}$ denotes the kind of objects (such as $\mathbb{S}$ for statements, $\mathbb{E}$ for expressions, $\mathbb{P}$ for programs). Subscripts over $\mathbb{X}$ are used to distinguish several kinds of semantics. Abstract semantics are distinguished using a $\sharp$ superscript.

Sequences. Given a set Σ, we denote as Σ^n the set of *sequences* of exactly n elements from Σ. The set of finite sequences is $\Sigma^* \overset{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \Sigma^n$. The set of infinite sequences is denoted Σ^ω, while the set of all sequences is $\Sigma^\infty \overset{\text{def}}{=} \Sigma^* \cup \Sigma^\omega$. The empty sequence is denoted as ε. Sequence concatenation

is denoted as . where $t \cdot t' = t$ when $t \in \Sigma^\omega$. It is naturally extended to sets of sequences: $A \cdot B \stackrel{\text{def}}{=} \{ a \cdot b \mid a \in A, b \in B \}$.

Traces. *Traces* generalize sequences. Given a set Σ of states and a set $\mathcal{A}$ of actions, a trace is a non-empty finite or infinite sequence of states in Σ interspersed with actions in $\mathcal{A}$, which we note as $\sigma_0 \stackrel{a_1}{\to} \sigma_1 \stackrel{a_2}{\to} \cdots \sigma_{n-2} \stackrel{a_{n-1}}{\to} \sigma_{n-1}$ (for a finite trace of length n) or $\sigma_0 \stackrel{a_1}{\to} \sigma_1 \stackrel{a_2}{\to} \cdots$ (for an infinite trace), where $\forall i : \sigma_i \in \Sigma, a_i \in \mathcal{A}$. As for sequences, we note $\mathcal{T}r^n(\Sigma, \mathcal{A})$, $\mathcal{T}r^*(\Sigma, \mathcal{A}) \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \mathcal{T}r^n(\Sigma, \mathcal{A})$, $\mathcal{T}r^\omega(\Sigma, \mathcal{A})$, and $\mathcal{T}r^\infty(\Sigma, \mathcal{A}) \stackrel{\text{def}}{=} \mathcal{T}r^*(\Sigma, \mathcal{A}) \cup \mathcal{T}r^\omega(\Sigma, \mathcal{A})$ respectively the set of traces of length n, of finite length, of infinite length, and the set of all traces. The concatenation of two traces t and t' by an action $a \in \mathcal{A}$ is denoted $t \stackrel{a}{\to} t'$: when t is infinite, $t \stackrel{a}{\to} t' = t$; otherwise, if $t = \sigma_0 \stackrel{a_1}{\to} \cdots \stackrel{a_n}{\to} \sigma_n$ and $t' = \sigma'_0 \stackrel{a'_1}{\to} \cdots$, then $t \stackrel{a}{\to} t' = \sigma_0 \stackrel{a_1}{\to} \cdots \stackrel{a_n}{\to} \sigma_n \stackrel{a}{\to} \sigma'_0 \stackrel{a'_1}{\to} \cdots$. When the action set $\mathcal{A}$ is a singleton, we will dispense from the (constant) action in traces, denoting them simply as $\sigma_0 \to \sigma_1 \to \cdots \to \sigma_n$, and sometimes assimilating traces to sequences.

Vectors. We use linear algebra: vectors are denoted as $\vec{V}$ and matrices as $\mathbf{M}$. The null vector is denoted as $\vec{0}$. The components of a vector $\vec{V}$ are denoted as $V_1, \ldots, V_n$. The columns of a matrix $\mathbf{M}$ are denoted as $\vec{M}_1, \ldots, \vec{M}_m$ and its elements as $M_{1,1}, \ldots, M_{n,m}$. Matrix-vector and matrix-matrix products are denoted as $\mathbf{M} \times \vec{V}$ and $\mathbf{M} \times \mathbf{N}$, while the dot product of vectors is denoted as $\vec{V} \cdot \vec{W}$. We overload the relational operators on vectors and matrices to denote the element-wise relation so that, for instance, $\vec{V} \geq \vec{W}$ means $\forall i : V_i \geq W_i$. Given a (column) vector $\vec{C}$, $\vec{C}^t$ denotes its transpose (row). We also denote as $\mathbf{M}^t$ the transpose of a matrix. Finally, we denote as $\vec{e}_i$ the $i-$th *basis vector*, i.e., the vector with all components set to 0, except the $i-$th which is set to 1.

Substitutions. We denote as $e[e_1/e_2]$ the (syntactic) operation of substituting in e every occurrence of e_1 with e_2.

2.2 Elements of abstract interpretation

We recall some core definitions and results of abstract interpretation, focusing on those that will be useful later to us (see [CC92a] for an in-depth presentation).

Abstractions and concretizations. A *semantic domain* is a set of elements carrying information about our objects of study (here, programs). We wish to quantify information, hence, a semantic domain is a partially ordered set $(\mathcal{D}, \sqsubseteq)$, where $d \sqsubseteq d'$ means that d' carries less information than d. We say that a semantic domain $(\mathcal{D}^\sharp, \sqsubseteq^\sharp)$, called the *abstract domain*, is an abstraction of another semantic domain, the *concrete domain* $(\mathcal{D}, \sqsubseteq)$, if each abstract element $d^\sharp \in \mathcal{D}^\sharp$ represents some concrete information $\gamma(d^\sharp) \in \mathcal{D}$ and the structure respects the information order: i.e., $\gamma \in \mathcal{D}^\sharp \to \mathcal{D}$ is a monotonic function; it is called the *concretization function*.

Remark. Two abstract elements can represent the same concrete one: γ needs not be injective.
End of remark.

When $\mathcal{D}^\sharp$ has arbitrary meets, it forms a so-called *Moore family* [CC79b] and we can define an *abstraction function* $\alpha \in \mathcal{D} \to \mathcal{D}^\sharp$ as:

$$\alpha(d) \stackrel{\text{def}}{=} \bigsqcap^\sharp \{ d^\sharp \mid d \sqsubseteq \gamma(d^\sharp) \} \ .$$

By definition, $\alpha(d)$ is the best (i.e., most precise) abstraction of d in $\mathcal{D}^\sharp$. The pair (α, γ) enjoys many well-known interesting properties:

- (α, γ) is a *Galois connection*:
  $\forall d \in \mathcal{D}, d^\sharp \in \mathcal{D}^\sharp : d \sqsubseteq \gamma(d^\sharp) \iff \alpha(d) \sqsubseteq^\sharp d^\sharp$;
- $\forall d \in \mathcal{D} : d \sqsubseteq (\gamma \circ \alpha)(d)$;
- $\forall d^\sharp \in \mathcal{D}^\sharp : (\alpha \circ \gamma)(d^\sharp) \sqsubseteq d^\sharp$.

When γ is injective (which is equivalent to state that α is surjective), then we actually have $\alpha \circ \gamma = \lambda d^\sharp . d^\sharp$, and the pair (α, γ) is called a *Galois injection*.

Example 2.2.1. In the interval domain (described in more details in Sec. 2.4.1), sets of reals in $\mathcal{D}$ (ordered by subset inclusion) are abstracted as intervals in $\mathcal{D}_i^\sharp$ with finite or infinite bounds:

$$\begin{array}{l}
\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{R}) \\
\sqsubseteq \stackrel{\text{def}}{=} \subseteq \\
\mathcal{D}_i^\sharp \stackrel{\text{def}}{=} \{ [a, b] \mid a \in \mathbb{R} \cup \{-\infty\}, b \in \mathbb{R} \cup \{+\infty\}, a \leq b \} \cup \{\bot^\sharp\} \\
d_1^\sharp \sqsubseteq_i^\sharp d_2^\sharp \iff_{\text{def}} d_1^\sharp = \bot^\sharp \vee \\
\qquad\qquad (d_1^\sharp = [a, b] \wedge d_2^\sharp = [c, d] \wedge a \geq c \wedge b \leq d) \\
\gamma_i([a, b]) \stackrel{\text{def}}{=} \{ x \in \mathbb{R} \mid a \leq x \leq b \}, \ \gamma(\bot^\sharp) \stackrel{\text{def}}{=} \emptyset \\
\alpha_i(X) \stackrel{\text{def}}{=} \begin{cases} \bot^\sharp & \text{if } X = \emptyset \\ [\min X, \max X] & \text{otherwise} \end{cases}
\end{array}$$

(α_i, γ_i) forms a Galois injection.
End of example.

Operator abstraction. Given a concrete operator $f \in \mathcal{D} \to \mathcal{D}$, an abstraction of f is a function $f^\sharp \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ obeying the *soundness condition*:

$$\forall d^\sharp \in \mathcal{D}^\sharp : f(\gamma(d^\sharp)) \sqsubseteq \gamma(f^\sharp(d^\sharp)) \tag{2.1}$$

which states that computing in the abstract always yields less or as much information as in the concrete. Ideally, we would have $f \circ \gamma = \gamma \circ f^\sharp$, which means that the abstract computation does not lose any information with respect to the concrete one; in this case, we will call $f^\sharp$ an *exact abstraction*. Unfortunately, this seldom happens: it requires f to be forward-complete, that is, to map abstract properties to abstract properties [GRS98]. When an abstraction function α exists, then we can define $f^\sharp$ as:

$$f^\sharp \stackrel{\text{def}}{=} \alpha \circ f \circ \gamma \tag{2.2}$$

which is, by definition, the best abstraction of f. These definitions extent naturally to the case of $n-$ary operators.

Example 2.2.2. Anticipating again on Sec. 2.4.1, we consider the interval abstractions of $f \stackrel{\text{def}}{=} \lambda X. \{ -x \mid x \in X \}$ and $\lambda(X, Y). X \cup Y$. Then $f_i^\sharp \stackrel{\text{def}}{=} \lambda[a, b]. [-b, -a]$ is an exact abstraction of f, while $\cup_i^\sharp \stackrel{\text{def}}{=} \lambda[a, b], [c, d]. [\min(a, c), \max(b, d)]$ is the *best abstraction* of $\cup$ but is not exact.
End of example.

It is important to notice that, although the composition of exact abstractions is an exact abstraction, the composition of best abstractions is not necessarily a best abstraction. Hence, when building an analysis by combining a set of atomic abstract operations, imprecisions can accumulate to an overall poor result, even if each atomic operation is a best abstraction. Adding to this the occasional lack of a best abstraction function α, and the occasional lack of an algorithm to implement efficiently (or at all) $\alpha \circ f \circ \gamma$, it turns out that abstract analyses seldom output the optimal result expressible in the chosen abstract domain. Thus, in order to prove properties of a certain kind, a strictyl more expressive abstract semantic domain is often required.

Reduced product. To obtain more precision, it is convenient to combine existing domains into new, more powerful ones. Given two domains $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$ with concretizations γ_1 and γ_2, the product domain $\mathcal{D}^\sharp \stackrel{\text{def}}{=} \mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp$ with concretization $\gamma(\langle X_1^\sharp, X_2^\sharp\rangle) \stackrel{\text{def}}{=} \gamma_1(X_1^\sharp) \sqcap \gamma_2(X_2^\sharp)$ and ordering $\langle X_1^\sharp, X_2^\sharp\rangle \sqsubseteq^\sharp \langle Y_1^\sharp, Y_2^\sharp\rangle \stackrel{\text{def}}{\iff} X_1^\sharp \sqsubseteq_1^\sharp Y_1^\sharp \wedge X_2^\sharp \sqsubseteq_2^\sharp Y_2^\sharp$ can represent conjunctions of properties expressed in $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$.

While $f^\sharp \stackrel{\text{def}}{=} \lambda\langle X_1^\sharp, X_1^\sharp\rangle.\, \langle f_1^\sharp(X_1^\sharp), f_2^\sharp(X_2^\sharp)\rangle$ is a sound abstraction of f in $\mathcal{D}^\sharp$ when $f_1^\sharp$ and $f_2^\sharp$ are sound abstractions of f in, respectively, $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$, it does not bring any precision improvement with respect to separate analyses as each component is computed in isolation. This can be corrected by adding a reduction step that propagates information: $f^\sharp$ is replaced with $\rho^\sharp \circ f^\sharp$ where the reduction function $\rho^\sharp \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ satisfies the soundness condition $(\gamma \circ \rho^\sharp)(X^\sharp) = \gamma(X^\sharp)$, and the improvement condition $\rho^\sharp(X^\sharp) \sqsubseteq^\sharp X^\sharp$. When $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$ feature abstraction functions α_1 and α_2, an optimal reduction can be defined as $\rho^\sharp(X^\sharp) \stackrel{\text{def}}{=} \langle(\alpha_1 \circ \gamma)(X^\sharp), (\alpha_2 \circ \gamma)(X^\sharp)\rangle$. When no abstraction function exists or no efficient algorithm to compute $\rho^\sharp$ exists, one generally settles for a sound reduction that only partially propagates information. We refer the reader to [CCF+06] on how to design partially reduced products on a large scale.

Fixpoint theorems. In abstract interpretation, many objects are expressed as fixpoints of operators. The existence of fixpoints requires suitable hypotheses on those operators. We recall an important result due to Tarski [Tar55]:

Theorem 2.2.1. *The set of fixpoints of a monotonic function $f \in A \to A$ in a complete lattice A is a non-empty complete lattice.*

In particular, f has a least fixpoint. Additionally, least fixpoints are expressed as meets of post-fixpoints:

$$\mathrm{lfp}_a\, f = \bigsqcap \{\, b \in A \mid a \sqsubseteq b \wedge f(b) \sqsubseteq b \,\} \ .$$

This characterization is not very convenient to compute fixpoints algorithmically. Hence, another theorem, by Cousot and Cousot [CC79a], expresses fixpoints as limits of (possibly transfinite) *iteration* sequences:

Theorem 2.2.2. *If $f \in A \to A$ is a monotonic function in a complete partial order A and a is a pre-fixpoint of A, then the following sequence:*

$$x_\delta \stackrel{\text{def}}{=} \begin{cases} a & \text{if } \delta = 0 \\ f(x_\beta) & \text{if } \delta = \beta + 1 \\ \bigsqcup \{\, x_\beta \mid \beta < \delta \,\} & \text{if } \delta \text{ is a limit ordinal} \end{cases}$$

converges towards $\mathrm{lfp}_a\, f$. If f is additionally a join-morphism, then $\mathrm{lfp}_a\, f = x_\omega$ (i.e., the iteration converges after a countable number of steps).

This theorem is constructive. It suggests a simple iterative way to compute fixpoints: we simply need to ensure that the involved sequences converge in finite time. Another remark is that the sequence x_δ is increasing for $\sqsubseteq$. Hence, the partial order, originally introduced to quantify information, also denotes a computation order for fixpoints (in fact, distinct orders can be used [Cou02], but this will not be necessary here).

Fixpoint approximation. Given a semantics expressed as $\mathrm{lfp}_a\, f$ in the concrete world, a natural idea is to abstract it as $\mathrm{lfp}_{a^\sharp}\, f^\sharp$ in the abstract world. We can then use fixpoint transfer theorems, such as [Cou02]:

Theorem 2.2.3. *If $f^\sharp \circ \alpha = \alpha \circ f$ and $a^\sharp = \alpha(a)$, then $\mathrm{lfp}_{a^\sharp}\, f^\sharp = \alpha(\mathrm{lfp}_a\, f)$.*

i.e., $\mathrm{lfp}_{a^\sharp}\, f^\sharp$ exists and is the best abstraction of $\mathrm{lfp}_a\, f$. However, in many cases, the condition $f^\sharp \circ \alpha = \alpha \circ f$ (also called backward completeness [GRS98]) is not satisfied. We must also consider the common case where $f^\sharp \neq \alpha \circ f \circ \gamma$ as the latter is too difficult to compute or does not exist at all (if there is no abstraction function α). When $f^\sharp \neq \alpha \circ f \circ \gamma$, it is even possible that $f^\sharp$ does not admit a least fixpoint (or any fixpoint at all). In all those cases, where no optimal fixpoint abstraction can be defined or computed, we settle for a sound abstraction, i.e., some $x^\sharp$ such that $\mathrm{lfp}_a\, f \sqsubseteq \gamma(x^\sharp)$. This can be easily achieved:

Theorem 2.2.4. *If $f^\sharp$ is a sound abstraction of f, $a \sqsubseteq \gamma(a^\sharp)$, and $x^\sharp$ satisfies $f^\sharp(x^\sharp) \sqsubseteq^\sharp x^\sharp$ and $a^\sharp \sqsubseteq^\sharp x^\sharp$, then $\mathrm{lfp}_a\, f \sqsubseteq \gamma(x^\sharp)$.*

that is, we abstract a concrete least fixpoint as an abstract post-fixpoint.

Fixpoint extrapolation. Theorem 2.2.2 suggests computing $\mathrm{lfp}_{a^\sharp}\, f^\sharp$ as the limit of the sequence defined as: $x_0^\sharp \stackrel{\text{def}}{=} a^\sharp$ and $x_{n+1}^\sharp \stackrel{\text{def}}{=} f^\sharp(x_n^\sharp)$. To enforce termination of such iterations in finite time, Cousot and Cousot introduced *widening* operators, which are binary operators $\triangledown \in (\mathcal{D}^\sharp \times \mathcal{D}^\sharp) \to \mathcal{D}^\sharp$ satisfying:

Definition 2.2.1.
- $\forall x^\sharp, y^\sharp : x^\sharp \sqsubseteq^\sharp x^\sharp \triangledown y^\sharp$ and $y^\sharp \sqsubseteq^\sharp x^\sharp \triangledown y^\sharp$;
- for any sequence $(y_i^\sharp)_{i\in\mathbb{N}}$, the sequence defined as $x_0^\sharp \stackrel{\text{def}}{=} y_0^\sharp$ and $x_{i+1}^\sharp \stackrel{\text{def}}{=} x_i^\sharp \triangledown y_i^\sharp$ is not strictly increasing.

We can then approximate $\mathrm{lfp}_a\, f$ with finite iterations:

Theorem 2.2.5. *If $f^\sharp$ is a sound abstraction of f and $a \sqsubseteq \gamma(a^\sharp)$, then the sequence $x_0^\sharp \stackrel{\text{def}}{=} a^\sharp$, $x_{i+1}^\sharp \stackrel{\text{def}}{=} x_i^\sharp \triangledown f^\sharp(x_i^\sharp)$ reaches a stable iterate $x_\beta^\sharp = x_{\beta+1}^\sharp$ for some $\beta < \omega$. Moreover, $\mathrm{lfp}_a\, f \sqsubseteq \gamma(x_\beta^\sharp)$.*

Intuitively, $\triangledown$ performs an extrapolation: it observes finite sequences of iterates and jumps higher and higher until it reaches (or overshots) the fixpoint. It is a form of inductive reasoning, in the logical sense of generalizing from finite examples, i.e., from the iterates (not to be confused with mathematical induction, which proceeds by applying induction axioms or rules, and is thus actually deductive in nature). Theorem 2.2.5 is, in fact, very general: it does not require $\mathrm{lfp}_{a^\sharp}\, f^\sharp$ to exist, nor any monotony nor join-morphism property on $f^\sharp$. However, it does not make any guarantee on the precision of the computed approximation, but only ensures soundness and termination.

Example 2.2.3. Anticipating again on Sec. 2.4.1, we present the classic interval widening:

$$[a,b] \triangledown_i [c,d] \stackrel{\text{def}}{=} \left[\begin{cases} a & \text{if } a \leq c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ +\infty & \text{otherwise} \end{cases} \right]$$

which sets unstable bounds to infinity. Consider the abstract function $f^\sharp \stackrel{\text{def}}{=} \lambda[a,b].\, [a,b] \cup_i^\sharp (([a,b] \cap_i^\sharp [0,10]) +_i^\sharp 1)$, modeling a loop increasing a counter while it is smaller than 10 ($\cap_i^\sharp$ and $+_i^\sharp$ are, respectively, the interval intersection and addition, which are exact). The iteration with widening starting from $[0,0]$ stabilizes at $[0,+\infty]$ after one iteration, which over-approximates the actual least fixpoint $[0,11]$.
End of example.

Sometimes, a fixpoint $x_\beta^\sharp = x_\beta^\sharp \triangledown f^\sharp(x_\beta^\sharp)$ is a strict post-fixpoint of $f^\sharp$: $f^\sharp(x_\beta^\sharp) \sqsubset x_\beta^\sharp$. Hence, the approximation $x_\beta^\sharp$ can be refined by performing a decreasing iteration without widening: $y_0^\sharp \stackrel{\text{def}}{=} x_\beta^\sharp$, $y_{i+1}^\sharp \stackrel{\text{def}}{=} f^\sharp(y_i^\sharp)$. This decreasing sequence can be infinite, so, Cousot and Cousot introduced a *narrowing* operator $\vartriangle$ to limit the refinement while enforcing termination (for example, by allowing each bound to be refined at most once).

The presentation of abstract interpretation using abstract domains $\mathcal{D}^\sharp$ and widenings $\triangledown$ has the benefit of clearly distinguishing the problem of abstracting a given concrete operator f in $\mathcal{D}^\sharp$ and that of abstracting fixpoints in $\mathcal{D}^\sharp$. The former problem is that of expressiveness, and influences the choice of $\mathcal{D}^\sharp$, while the later is that of termination. As demonstrated in [CC92b], computing in a domain with infinite chains using a widening is strictly more powerful than computing in a finite-chain restriction of the same domain (which does not require any widening). Intuitively, the widening adds a dynamic dimension to the abstraction, which is more flexible than relying only on the static choice of an abstract domain.

2.3 Sequential static analysis

We apply the previous notions to construct a simple static analysis. In order to present the construction concisely but in full formal details, we study a very simple artificial language: it is imperative, sequential, block-structured, procedure-less and with only global variables and one data-type: reals in $\mathbb{R}$. Later sections will introduce additional constructs (floating-point numbers in Sec. 2.4.4, concurrency in Chap. 3, and arrays and pointers in Chap. 5) while others (such as dynamic memory allocation, objects, recursive procedures, higher-order constructs, etc.) are out of the scope of this work. Despite the remaining limitations, the construction is nevertheless relevant to some real-life analysis problems (this is shown in Chap. 6 on a subset of C for embedded critical software).

2.3.1 Language

Our simple language is presented in Fig. 2.1. Statements *stat* include assignments $X \leftarrow e$, conditionals **if** $\cdots$ **then** $\cdots$ **endif**, loops **while** $\cdots$ **do** $\cdots$ **done**, and sequencing ;. A program *prog* is simply a statement. Expressions *expr* are numeric and include (real-valued) variables drawn from a fixed finite set $\mathcal{V}$, constants (or, more precisely, intervals with constant bounds $[c_1, c_2]$), unary and binary operators. Interval constants model the choice of a random value within the given bounds, which combines the modeling of classic constants $[c,c]$ and of non-deterministic inputs (such as sensors).

Statements are decorated with superscript labels ℓ, which denote syntactic locations and should be all distinct. There is a label at the beginning and the end of each statement, as well as a label ℓ_i to denote the location where a loop condition is tested before each new iteration. Additionally, expression operators are decorated with unique subscript labels ω. These denote the location of possible run-time errors. We denote respectively as $\mathcal{L}(P)$ and $\Omega(P)$ the (finite) sets of statement labels and error labels in a program P. Generally, the program P is implicit and we shorten the notations as $\mathcal{L}$ and Ω.

2.3.2 Transition system

Following Cousot and Cousot [CC77], we model program semantics as a *labelled transition system* $(\Sigma, \mathcal{A}, I, \tau)$, given as:
- Σ: a set of states;
- $\mathcal{A}$: a set of actions;
- $I \subseteq \Sigma$: a set of initial states;
- $\tau \subseteq \Sigma \times \mathcal{A} \times \Sigma$: a transition relation.

Transitions model execution steps: $(\sigma, a, \sigma') \in \tau$ means that the program can transition from state σ to state σ' by executing the action a. We will use the notation $\sigma \xrightarrow{a}_\tau \sigma'$ for $(\sigma, a, \sigma') \in \tau$.

Transition systems are a form of small-step semantics. They are independent from the choice of programming language and allow expressing very general results, some of which will be applied to our language in Sec. 2.3.3. Before this, we need to show how a program *prog* $\stackrel{\text{def}}{=}$ ${}^{\ell e}\, stat\, {}^{\ell x}$ in our language is effectively mapped to a transition system:

- As state space, we use $\Sigma \stackrel{\text{def}}{=} (\mathcal{L} \times \mathcal{E}) \cup \Omega$ where $\mathcal{E} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{R}$: a program execution is either at some syntactic location $\ell \in \mathcal{L}$ with environment $\rho \in \mathcal{E}$ mapping each variable $V \in \mathcal{V}$ to a real value $\rho(V) \in \mathbb{R}$, or it is in an error state $\omega \in \Omega$.

- Programs start at the first location with all variables initialized to 0, hence, we have $I \stackrel{\text{def}}{=} \{ \langle \ell e, \lambda V \in \mathcal{V}.\, 0 \rangle \}$.

- There is a single action $\mathcal{A} \stackrel{\text{def}}{=} \{ * \}$ that denotes an execution step.[2] As a consequence, we will assimilate τ to a subset of $\Sigma \times \Sigma$ and note $(\sigma, *, \sigma') \in \tau$ as $\sigma \to_\tau \sigma'$.

- The transition relation τ is defined by induction on the syntax of statements. It is shown in Fig. 2.3, where $\tau[\![{}^{\ell}\, stat\, {}^{\ell'}]\!]$ is the set of transitions generated by the statement ${}^{\ell}\, stat\, {}^{\ell'}$.

  The semantics uses the auxiliary semantic function $\mathbb{E}[\![e]\!]\rho$, defined in Fig. 2.2, to evaluate an expression e in an environment $\rho \in \mathcal{E}$. This function outputs a set of values and a set of possible run-time errors. Expression semantics are also defined by induction on the syntax, but in big-step form: their intermediate computation steps are not visible at the level of program transitions. Note that value sets are necessary because, due to non-deterministic constants $[c_1, c_2]$, an expression can have several values. In our simple real-based language, the only possible run-time errors are caused by divisions by zero.

[2] Multiple actions will appear later, in the semantics of concurrent programs (Sec. 3.1).

$$
\begin{array}{lcll}
\mathit{prog} & ::= & {}^{\ell}\,\mathit{stat}\,{}^{\ell'} & \textit{(program)} \\
{}^{\ell}\,\mathit{stat}\,{}^{\ell'} & ::= & {}^{\ell}X \leftarrow \mathit{expr}\,{}^{\ell'} & \textit{(assignment)} \\
 & | & {}^{\ell}\mathbf{if}\ \mathit{expr} \bowtie 0\ \mathbf{then}\ {}^{\ell_1}\,\mathit{stat}\,{}^{\ell_2}\ \mathbf{endif}^{\ell'} & \textit{(conditional)} \\
 & | & {}^{\ell}\mathbf{while}\ {}^{\ell_i}\ \mathit{expr} \bowtie 0\ \mathbf{do}\ {}^{\ell_1}\,\mathit{stat}\,{}^{\ell_2}\ \mathbf{done}^{\ell'} & \textit{(loop)} \\
 & | & {}^{\ell}\,\mathit{stat}\,;^{\ell_1}\ \mathit{stat}\,{}^{\ell'} & \textit{(sequence)} \\
\mathit{expr} & ::= & X & \textit{(variable } X \in \mathcal{V}\textit{)} \\
 & | & [c_1, c_2] & \textit{(constant interval, } c_1, c_2 \in \mathbb{R} \cup \{\pm\infty\}\textit{)} \\
 & | & \circ_\omega\ \mathit{expr} & \textit{(unary operation)} \\
 & | & \mathit{expr} \diamond_\omega \mathit{expr} & \textit{(binary operation)} \\
\bowtie & ::= & = \mid \neq \mid < \mid > \mid \leq \mid \geq & \textit{(relational operator)} \\
\circ & ::= & - & \textit{(unary arithmetic operator)} \\
\diamond & ::= & + \mid - \mid \times \mid / & \textit{(binary arithmetic operator)} \\
\ell & \in & \mathcal{L} & \textit{(statement label)} \\
\omega & \in & \Omega & \textit{(error location)}
\end{array}
$$

Figure 2.1: Syntax of our sequential language.

$\underline{\mathbb{E}[\![\ \mathit{expr}\ ]\!] \in \mathcal{E} \to (\mathcal{P}(\mathbb{R}) \times \mathcal{P}(\Omega))}$

$\mathbb{E}[\![\ X\ ]\!]\rho \stackrel{\text{def}}{=} \langle \{\, \rho(X)\,\}, \emptyset \rangle$

$\mathbb{E}[\![\ [c_1, c_2]\ ]\!]\rho \stackrel{\text{def}}{=} \langle \{\, x \in \mathbb{R} \mid c_1 \leq x \leq c_2\,\}, \emptyset \rangle$

$\mathbb{E}[\![\ \circ_\omega e\ ]\!]\rho \stackrel{\text{def}}{=}$
$\quad$ let $\langle V, O\rangle = \mathbb{E}[\![\ e\ ]\!]\,\rho$ in $\langle \{\, \circ\, v \mid v \in V\,\}, O\rangle$

$\mathbb{E}[\![\ e_1 \diamond_\omega e_2\ ]\!]\rho \stackrel{\text{def}}{=}$
$\quad$ let $\langle V_1, O_1\rangle = \mathbb{E}[\![\ e_1\ ]\!]\,\rho$ in
$\quad$ let $\langle V_2, O_2\rangle = \mathbb{E}[\![\ e_2\ ]\!]\,\rho$ in
$\quad \langle \{\, v_1 \diamond v_2 \mid v_1 \in V_1,\, v_2 \in V_2,\, \diamond \neq / \vee v_2 \neq 0\,\},$
$\quad\ O_1 \cup O_2 \cup \{\, \omega \ \text{ if } \diamond = / \wedge 0 \in V_2\,\}\rangle$

Figure 2.2: Semantics of expressions.

2.3.3 From traces to states

Maximal traces semantics. Transition systems $(\Sigma, \mathcal{A}, I, \tau)$ are only static mathematical descriptions of programs. Information about their dynamic behaviors emerge when considering sequences of transitions. The *maximal traces* semantics $\mathcal{M}$ expresses the most information about a program: it is the set of maximal finite or infinite traces, in $\mathcal{T}r^{\infty}(\Sigma, \mathcal{A})$, starting in a state in I and obeying the transition relation. Defining the *blocking states* B as the states without any successor $B \stackrel{\text{def}}{=} \{\, \sigma \mid \forall \sigma' \in \Sigma, a \in \mathcal{A} : \sigma \stackrel{a}{\not\to}_\tau \sigma'\,\}$, we can define $\mathcal{M}$ as:

$$
\begin{array}{rl}
\mathcal{M} \stackrel{\text{def}}{=} & \{\, \sigma_0 \stackrel{a_1}{\to} \cdots \stackrel{a_n}{\to} \sigma_n \mid \sigma_0 \in I \wedge \sigma_n \in B \wedge \\
 & \quad \forall i < n : \sigma_i \stackrel{a_{i+1}}{\to}_\tau \sigma_{i+1}\,\} \\
 & \cup \{\, \sigma_0 \stackrel{a_1}{\to} \cdots \mid \sigma_0 \in I \wedge \forall i \in \mathbb{N} : \sigma_i \stackrel{a_{i+1}}{\to}_\tau \sigma_{i+1}\,\}\ .
\end{array}
\tag{2.3}
$$

An equally important fact is that interesting program properties can also be modeled as sets of traces. Given a property $P \subseteq \mathcal{T}r^{\infty}(\Sigma, \mathcal{A})$, checking whether the program enjoys this property is achieved by testing whether $\mathcal{M} \subseteq P$.

Example 2.3.1. In the simple case where $\mathcal{A}$ is a singleton, we assimilate traces to sequences of states, in Σ^∞, and define the following properties:

- choosing $P \stackrel{\text{def}}{=} S^\infty$ checks that the program stays in a subset of states $S \subseteq \Sigma$ (invariance); checking for the absence of run-time error is achieved by setting $S \stackrel{\text{def}}{=} \Sigma \setminus \Omega$;
- choosing $P \stackrel{\text{def}}{=} \Sigma^*$ checks that the program terminates;
- choosing $P \stackrel{\text{def}}{=} \Sigma^* \cdot S \cdot \Sigma^\infty$ checks that the program necessarily reaches a state in $S \subseteq \Sigma$ (inevitability).

End of example.

Remark. In the presence of non-determinism (e.g., due to interval constants), we actually check that all executions spawning from any sequence of choices satisfy the target property.

End of remark.

$\underline{\tau[\![\,{}^{\ell}\,\mathit{stat}\,{}^{\ell'}\,]\!] \in \mathcal{P}(\Sigma \times \Sigma)}$

$\quad$ let $\forall e, \rho : \langle V^e_\rho, O^e_\rho\rangle = \mathbb{E}[\![\ e\ ]\!]\,\rho$ in
$\quad \tau[\![\,{}^{\ell}X \leftarrow e^{\ell'}\,]\!] \stackrel{\text{def}}{=}$
$\qquad \{\, (\langle \ell, \rho\rangle, \langle \ell', \rho[X \mapsto v]\rangle) \mid \rho \in \mathcal{E},\, v \in V^e_\rho\,\} \cup$
$\qquad \{\, (\langle \ell, \rho\rangle, \omega) \mid \rho \in \mathcal{E},\, \omega \in O^e_\rho)\,\}$

$\quad \tau[\![\,{}^{\ell}\mathbf{if}\ e \bowtie 0\ \mathbf{then}\ {}^{\ell_1}s^{\ell_2}\ \mathbf{endif}^{\ell'}\,]\!] \stackrel{\text{def}}{=}$
$\qquad \{\, (\langle \ell, \rho\rangle, \langle \ell_1, \rho\rangle) \mid \rho \in \mathcal{E},\, \exists v \in V^e_\rho : v \bowtie 0\,\} \cup$
$\qquad \{\, (\langle \ell, \rho\rangle, \langle \ell', \rho\rangle) \mid \rho \in \mathcal{E},\, \exists v \in V^e_\rho : v \not\bowtie 0\,\} \cup$
$\qquad \{\, (\langle \ell, \rho\rangle, \omega) \mid \rho \in \mathcal{E},\, \omega \in O^e_\rho)\,\} \cup$
$\qquad \tau[\![\,{}^{\ell_1}s^{\ell_2}\,]\!] \cup \{\, (\langle \ell_2, \rho\rangle, \langle \ell', \rho\rangle) \mid \rho \in \mathcal{E}\,\}$

$\quad \tau[\![\,{}^{\ell}\mathbf{while}\ {}^{\ell_i}e \bowtie 0\ \mathbf{do}\ {}^{\ell_1}s^{\ell_2}\ \mathbf{done}^{\ell'}\,]\!] \stackrel{\text{def}}{=}$
$\qquad \{\, (\langle \ell, \rho\rangle, \langle \ell_i, \rho\rangle) \mid \rho \in \mathcal{E}\,\} \cup$
$\qquad \{\, (\langle \ell_i, \rho\rangle, \langle \ell_1, \rho\rangle) \mid \rho \in \mathcal{E},\, \exists v \in V^e_\rho : v \bowtie 0\,\} \cup$
$\qquad \{\, (\langle \ell_i, \rho\rangle, \langle \ell', \rho\rangle) \mid \rho \in \mathcal{E},\, \exists v \in V^e_\rho : v \not\bowtie 0\,\} \cup$
$\qquad \{\, (\langle \ell_i, \rho\rangle, \omega) \mid \rho \in \mathcal{E},\, \omega \in O^e_\rho)\,\} \cup$
$\qquad \tau[\![\,{}^{\ell_1}s^{\ell_2}\,]\!] \cup \{\, (\langle \ell_2, \rho\rangle, \langle \ell_i, \rho\rangle) \mid \rho \in \mathcal{E}\,\}$

$\quad \tau[\![\,{}^{\ell}s_1;^{\ell_1}\ s_2{}^{\ell'}\,]\!] \stackrel{\text{def}}{=} \tau[\![\,{}^{\ell}s_1{}^{\ell_1}\,]\!] \cup \tau[\![\,{}^{\ell_1}s_2{}^{\ell'}\,]\!]$

Figure 2.3: Transition system generated by a program.

Partial traces semantics. The maximal trace semantics is difficult to compute as it involves infinite traces. A solution consists in observing the finite prefixes of finite and infinite executions, called *partial traces*, which leads to the following

semantics $\mathcal{F} \in \mathcal{Tr}^*(\Sigma, \mathcal{A})$:

$$\mathcal{F} \stackrel{\text{def}}{=} \{\, \sigma_0 \stackrel{a_1}{\to} \cdots \stackrel{a_n}{\to} \sigma_n \mid \sigma_0 \in I \wedge \forall i < n : \sigma_i \stackrel{a_{i+1}}{\to}_\tau \sigma_{i+1} \,\} \ . \tag{2.4}$$

$\mathcal{F}$ is an abstraction of $\mathcal{M}$. Indeed, $\mathcal{F} = \alpha_{pref}(\mathcal{M})$, where:

$$\alpha_{pref} \stackrel{\text{def}}{=} \lambda T. \{\, t \in \mathcal{Tr}^*(\Sigma, \mathcal{A}) \mid t \in T \vee \exists a, t' : t \stackrel{a}{\to} t' \in T \,\} \ . \tag{2.5}$$

This abstraction is not complete: $\mathcal{F}$ can prove strictly fewer properties than $\mathcal{M}$ due to the loss of information on infinite traces.

Example 2.3.2. α_{pref} collapses some sets containing infinite traces with sets not containing any, e.g.:

$$\alpha_{pref}(\{\sigma\}^\omega) = \alpha_{pref}(\{\sigma\}^*) = \{\sigma\}^* \ .$$

More generally, it is not possible with $\mathcal{F}$ to prove that programs with finite traces of unbounded length always terminate ($\mathcal{F}$ is nevertheless complete for bounded termination as $\forall n : \alpha_{pref}(T) \subseteq \bigcup_{i \leq n} \Sigma^i \iff T \subseteq \bigcup_{i \leq n} \Sigma^i$).
End of example.

Nevertheless, $\mathcal{F}$ can express invariance exactly. Indeed:

$$\begin{array}{l} \forall T \subseteq \mathcal{Tr}^\infty(\Sigma, \mathcal{A}),\ S \subseteq \Sigma : \\ \alpha_{pref}(T) \subseteq \mathcal{Tr}^*(S, \mathcal{A}) \iff T \subseteq \mathcal{Tr}^\infty(S, \mathcal{A}) \ . \end{array}$$

Another important feature of this semantics is that it can be expressed in fixpoint form, as $\mathcal{F} = \operatorname{lfp} F$ where:

$$\begin{array}{rl} F \stackrel{\text{def}}{=} \lambda X. I \cup \{\, \sigma_0 \stackrel{a_1}{\to} \cdots \sigma_i \stackrel{a_{i+1}}{\to} \sigma_{i+1} \mid & \\ \qquad \sigma_0 \stackrel{a_1}{\to} \cdots \sigma_i \in X \wedge \sigma_i \stackrel{a_{i+1}}{\to}_\tau \sigma_{i+1} \,\} \ . & \end{array} \tag{2.6}$$

F is a join-morphism that includes initial states and extends traces by adding a new transition at their end: it is a forward semantics.[3] By Thm. 2.2.2, $\operatorname{lfp} F$ can then be expressed as the limit of an iteration sequence, $\emptyset$, $F(\emptyset)$, $F^2(\emptyset)$, etc., which stabilizes at $\cup_{i<\omega} F^i(\emptyset)$.

Reachable state semantics. Computing $\operatorname{lfp} F$ by iteration is equivalent to exhaustive testing, i.e., running the program and observing all its executions, albeit in a non-standard (i.e., breadth-first) order. It does not terminate when the program has infinite executions. Thankfully, as we are interested in invariance properties, it is sufficient to observe the set of *reachable states* $\mathcal{R} \subseteq \Sigma$, which is an abstraction of $\mathcal{F}$. We have $\mathcal{R} \stackrel{\text{def}}{=} \alpha_{reach}(\mathcal{F})$ where:

$$\alpha_{reach} \stackrel{\text{def}}{=} \lambda T. \{\, \sigma \mid \exists \sigma_0 \stackrel{a_0}{\to} \cdots \sigma_n \in T : \exists i \leq n : \sigma = \sigma_i \,\} \ . \tag{2.7}$$

And the associated concretization is simply:

$$\gamma_{reach} \stackrel{\text{def}}{=} \lambda S. \mathcal{Tr}^*(S, \mathcal{A}) \ .$$

The abstraction is complete for reachability as:

$$\begin{array}{l} \forall T \subseteq \mathcal{Tr}^*(\Sigma, \mathcal{A}),\ S \subseteq \Sigma : \\ \alpha_{reach}(T) \subseteq S \iff T \subseteq \mathcal{Tr}^*(S, \mathcal{A}) \ . \end{array}$$

However, α_{reach} forgets all information related to the ordering of states in executions.

[3] There also exists a fixpoint characterization of the maximal trace semantics $\mathcal{M}$ [Cou02], but it is a backward semantics that cannot enforce $\sigma_0 \in I$. Unlike $\mathcal{F}$, we are not aware of any forward fixpoint characterization of $\mathcal{M}$.

$$\underline{eq[\!\![^\ell\ stat\ ^{\ell'}]\!\!] \in \mathcal{P}(Equations[(\mathcal{X}_\ell)_{\ell \in \mathcal{L}}])}$$

$$eq[\!\![^{\ell e}\ stat\ ^{\ell x}]\!\!] \stackrel{\text{def}}{=} \{\, \mathcal{X}_{\ell e} = \{\, \lambda V. 0 \,\} \,\} \cup eq_{st}[\!\![^{\ell e}\ stat\ ^{\ell x}]\!\!]$$

$$eq_{st}[\!\![^\ell X \leftarrow e^{\ell'}]\!\!] \stackrel{\text{def}}{=} \{\, \mathcal{X}_{\ell'} = \mathbb{S}_\mathcal{E}[\![X \leftarrow e]\!] \mathcal{X}_\ell \,\}$$

$$\begin{array}{l} eq_{st}[\!\![^\ell \textbf{if}\ e \bowtie 0\ \textbf{then}\ ^{\ell_1} s^{\ell_2}\ \textbf{endif}^{\ell'}]\!\!] \stackrel{\text{def}}{=} \\ \quad \{\, \mathcal{X}_{\ell_1} = \mathbb{S}_\mathcal{E}[\![e \bowtie 0]\!] \mathcal{X}_\ell \,\} \cup eq_{st}[\!\![^{\ell_1} s^{\ell_2}]\!\!] \cup \\ \quad \{\, \mathcal{X}_{\ell'} = \mathcal{X}_{\ell_2} \cup \mathbb{S}_\mathcal{E}[\![e \not\bowtie 0]\!] \mathcal{X}_\ell \,\} \end{array}$$

$$\begin{array}{l} eq_{st}[\!\![^\ell \textbf{while}\ ^{\ell_i} e \bowtie 0\ \textbf{do}\ ^{\ell_1} s^{\ell_2}\ \textbf{done}^{\ell'}]\!\!] \stackrel{\text{def}}{=} \\ \quad \{\, \mathcal{X}_{\ell_i} = \mathcal{X}_\ell \cup \mathcal{X}_{\ell_2} \,\} \cup \{\, \mathcal{X}_{\ell_1} = \mathbb{S}_\mathcal{E}[\![e \bowtie 0]\!] \mathcal{X}_{\ell_i} \,\} \cup \\ \quad eq_{st}[\!\![^{\ell_1} s^{\ell_2}]\!\!] \cup \{\, \mathcal{X}_{\ell'} = \mathbb{S}_\mathcal{E}[\![e \not\bowtie 0]\!] \mathcal{X}_{\ell_i} \,\} \end{array}$$

$$eq_{st}[\!\![^\ell s_1;^{\ell_1} s_2{}^{\ell'}]\!\!] \stackrel{\text{def}}{=} eq_{st}[\!\![^\ell s_1{}^{\ell_1}]\!\!] \cup eq_{st}[\!\![^{\ell_1} s_2{}^{\ell'}]\!\!]$$

where:
let $\forall e, \rho : \langle V^e_\rho, - \rangle = \mathbb{E}[\![e]\!] \rho$ in
$\mathbb{S}_\mathcal{E}[\![X \leftarrow e]\!] R \stackrel{\text{def}}{=} \{\, \rho[X \mapsto v] \mid \rho \in R,\ v \in V^e_\rho \,\}$
$\mathbb{S}_\mathcal{E}[\![e \bowtie 0]\!] R \stackrel{\text{def}}{=} \{\, \rho \in R \mid \exists v \in V^e_\rho : v \bowtie 0 \,\}$

Figure 2.4: Equation system generated by a program.

A fixpoint characterisation of $\mathcal{R}$ can be constructed by fixpoint abstraction, using Thm. 2.2.3. We define the function $R \in \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ as:

$$R \stackrel{\text{def}}{=} \lambda S. I \cup \{\, \sigma \mid \exists \sigma' \in S, a \in \mathcal{A} : \sigma' \stackrel{a}{\to}_\tau \sigma \,\} \tag{2.8}$$

and note that $R \circ \alpha_{reach} = \alpha_{reach} \circ F$, which implies that $\mathcal{R} = \operatorname{lfp} R$. Computing $\operatorname{lfp} R$ by iteration corresponds to a breadth-first exploration of reachable sets. It terminates if Σ is finite (even though $\mathcal{F}$ may be infinite). However, Σ is often infinite, or so large that the reachable subset cannot be represented in extension in a computer. We will have to resort to further abstractions.

2.3.4 Equational semantics

Before abstracting further, we apply the reachable set abstraction on the transition systems generated by our language described in Sec. 2.3.1, and restate the semantics in a more convenient, equation-based form. This classic form dates back from the beginning of abstract interpretation [CC79b] and it is effectively used in academic and industrial tools (such as Interproc [LAJ11] and Sparrow [Ya]).

The principle is to partition the set of reachable states $\mathcal{R} \subseteq \Sigma$ by their syntactic program location in $\mathcal{L}$. Given a program $P \stackrel{\text{def}}{=} {}^{\ell e}\ stat\ ^{\ell x}$, we associate a variable $\mathcal{X}_\ell$ with value in $\mathcal{P}(\mathcal{E})$ to each syntactic location $\ell \in \mathcal{L}(P)$ (later abbreviated as $\mathcal{L}$), such that $\mathcal{X}_\ell \stackrel{\text{def}}{=} \{\, \rho \mid \langle \ell, \rho \rangle \in \mathcal{R} \,\}$. As $\mathcal{R} = \operatorname{lfp} R$, $(\mathcal{X}_\ell)_{\ell \in \mathcal{L}}$ is the least family, for the element-wise subset ordering on $\mathcal{L} \to \mathcal{P}(\mathcal{E})$, satisfying: $(\rho \in \mathcal{X}_\ell \wedge \langle \ell, \rho \rangle \to_\tau \langle \ell', \rho' \rangle) \vee \langle \ell', \rho' \rangle \in I \implies \rho' \in \mathcal{X}_{\ell'}$. It is then a simple process to massage the definition of P's transition system from Fig. 2.3 into a set of equations of the form $\mathcal{X}_\ell = F_\ell(\mathcal{X}_{\ell_1}, \ldots, \mathcal{X}_{\ell_n})$. This leads to the set of equations $eq[\!\![^{\ell e}\ stat\ ^{\ell x}]\!\!]$ presented in Fig. 2.4. This set contains an equation defining the initial states $\mathcal{X}_{\ell e}$, as well as statement equations defined by induction on the program syntax: the function $eq_{st}[\!\![^\ell\ stat\ ^{\ell'}]\!\!]$ generates a set of equations binding the variables for all the locations in ${}^\ell\ stat\ ^{\ell'}$ except ℓ. Moreover, the translation uses two auxiliary semantic functions, $\mathbb{S}_\mathcal{E}[\![X \leftarrow e]\!]$ and

$^{\ell 1}i \leftarrow 2;$
$^{\ell 2}n \leftarrow [-\infty, +\infty];$
$^{\ell 3}\textbf{while } ^{\ell 4}\ i < n \textbf{ do}$
$\quad ^{\ell 5}\textbf{if } [0,1] = 0 \textbf{ then}$
$\quad\quad ^{\ell 6}i \leftarrow i + 1$
$\quad ^{\ell 7}\textbf{endif}$
$^{\ell 8}\textbf{done}$
$^{\ell 9}$

(a)

$\mathcal{X}_{\ell 1} = \{\, [i \mapsto 0, n \mapsto 0] \,\}$
$\mathcal{X}_{\ell 2} = \mathbb{S}_{\mathcal{E}}[\![\, i \leftarrow 2 \,]\!]\mathcal{X}_{\ell 1}$
$\mathcal{X}_{\ell 3} = \mathbb{S}_{\mathcal{E}}[\![\, n \leftarrow [-\infty, +\infty] \,]\!]\mathcal{X}_{\ell 2}$
$\mathcal{X}_{\ell 4} = \mathcal{X}_{\ell 3} \cup \mathcal{X}_{\ell 8}$
$\mathcal{X}_{\ell 5} = \mathbb{S}_{\mathcal{E}}[\![\, i < n \,]\!]\mathcal{X}_{\ell 4}$
$\mathcal{X}_{\ell 6} = \mathcal{X}_{\ell 5}$
$\mathcal{X}_{\ell 7} = \mathbb{S}_{\mathcal{E}}[\![\, i \leftarrow i + 1 \,]\!]\mathcal{X}_{\ell 6}$
$\mathcal{X}_{\ell 8} = \mathcal{X}_{\ell 5} \cup \mathcal{X}_{\ell 7}$
$\mathcal{X}_{\ell 9} = \mathbb{S}_{\mathcal{E}}[\![\, i \geq n \,]\!]\mathcal{X}_{\ell 4}$

(b)

$\mathcal{X}_{\ell 1} = \{\, \langle x,\, n\rangle \mid i = 0 \wedge n = 0 \,\}$
$\mathcal{X}_{\ell 2} = \{\, \langle x,\, n\rangle \mid i = 2 \wedge n = 0 \,\}$
$\mathcal{X}_{\ell 3} = \{\, \langle x,\, n\rangle \mid i = 2 \,\}$
$\mathcal{X}_{\ell 4} = \{\, \langle x,\, n\rangle \mid 2 \leq i \leq \max(2, n) \,\}$
$\mathcal{X}_{\ell 5} = \{\, \langle x,\, n\rangle \mid 2 \leq i \leq n - 1 \wedge n \geq 3 \,\}$
$\mathcal{X}_{\ell 6} = \{\, \langle x,\, n\rangle \mid 2 \leq i \leq n - 1 \wedge n \geq 3 \,\}$
$\mathcal{X}_{\ell 7} = \{\, \langle x,\, n\rangle \mid 3 \leq i \leq n \wedge n \geq 3 \,\}$
$\mathcal{X}_{\ell 8} = \{\, \langle x,\, n\rangle \mid 2 \leq i \leq n \wedge n \geq 3 \,\}$
$\mathcal{X}_{\ell 9} = \{\, \langle x,\, n\rangle \mid i = \max(2, n) \,\}$

(c)

Figure 2.5: Example program (a), its equation system (b), and its smallest solution in assertional form (c).

$\mathbb{S}_{\mathcal{E}}[\![\, e \bowtie 0 \,]\!]$, that respectively model the effect of assigning an expression to a variable and filtering environments according to the outcome of a test. These are defined, in turn, using the expression semantics $\mathbb{E}[\![\, e \,]\!]$ from Fig. 2.3. The family $(\mathcal{X}_\ell)_{\ell \in \mathcal{L}}$ we seek is then the least solution of this system, which is nothing more than a least fixpoint. To lighten the presentation, the equation system does not track error states in Ω.

Example 2.3.3. Figure 2.5.(a) presents an example program that increments i from 2 to some user-input value n. Figure 2.5.(b) presents the associated equation system.

End of example.

The family $(\mathcal{X}_\ell)_{\ell \in \mathcal{L}}$ defines program *invariants*: whenever a program execution passes through location ℓ, its environment ρ always satisfies $\rho \in \mathcal{X}_\ell$. There is a deep connection between this presentation and Floyd–Hoare logic [Flo67, Hoa69]: Cousot and Cousot [CC77] showed that any solution of $\mathcal{X}_\ell = F_\ell(\mathcal{X}_{\ell_1}, \ldots, \mathcal{X}_{\ell_n})$ is an *inductive invariant*, leading to valid *Hoare triples*. Moreover, the least solution we compute corresponds to the most precise invariant. We will present a similar connection for concurrent programs in Sec. 3.2.

Example 2.3.4. Figure 2.5.(c) presents the least solution of the system in Fig. 2.5.(b). It is presented as logical assertions on the variable pair $\langle x,\, n\rangle$ at each program location, which makes the connection with Hoare logic more apparent.

End of example.

Remark. Equation system semantics are not limited to computing reachability, they can also model partial traces: $\mathcal{X}_\ell$ will then collect the partial traces that end in a state of the form $\langle \ell,\, \rho\rangle$. This opens the way to history-sensitive static analyses, such as traces partitioning [MR05].

End of remark.

$\underline{\mathbb{S}[\![\ stat\ ]\!] \in (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega)) \to (\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega))}$

let $\forall e, \rho : \langle V^e_\rho,\, O^e_\rho\rangle = \mathbb{E}[\![\, e \,]\!]\, \rho$ in

$\mathbb{S}[\![\, X \leftarrow e \,]\!]\langle R,\, O\rangle \stackrel{\text{def}}{=}$
$\quad \langle \emptyset,\, O\rangle \sqcup \bigsqcup_{\rho \in R} \langle \{\, \rho[X \mapsto v] \mid v \in V^e_\rho \,\},\, O^e_\rho\rangle$

$\mathbb{S}[\![\, e \bowtie 0 \,]\!]\langle R,\, O\rangle \stackrel{\text{def}}{=}$
$\quad \langle \emptyset,\, O\rangle \sqcup \bigsqcup_{\rho \in R} \langle \{\, \rho \mid \exists v \in V^e_\rho : v \bowtie 0 \,\},\, O^e_\rho\rangle$

$\mathbb{S}[\![\, \textbf{if } e \bowtie 0 \textbf{ then } s \textbf{ endif} \,]\!]\mathcal{X} \stackrel{\text{def}}{=}$
$\quad (\mathbb{S}[\![\, s \,]\!] \circ \mathbb{S}[\![\, e \bowtie 0 \,]\!])\mathcal{X} \sqcup \mathbb{S}[\![\, e \not\bowtie 0 \,]\!]\mathcal{X}$

$\mathbb{S}[\![\, \textbf{while } e \bowtie 0 \textbf{ do } s \textbf{ done} \,]\!]\mathcal{X} \stackrel{\text{def}}{=}$
$\quad \mathbb{S}[\![\, e \not\bowtie 0 \,]\!](\text{lfp}\, \lambda \mathcal{Y}.\, \mathcal{X} \sqcup (\mathbb{S}[\![\, s \,]\!] \circ \mathbb{S}[\![\, e \bowtie 0 \,]\!])\mathcal{Y})$

$\mathbb{S}[\![\, s_1;\ s_2 \,]\!] \stackrel{\text{def}}{=} \mathbb{S}[\![\, s_2 \,]\!] \circ \mathbb{S}[\![\, s_1 \,]\!]$

Figure 2.6: Big-step semantics.

2.3.5 Big-step semantics

Another popular way of presenting the reachability semantics, also used effectively in tools (such as Astrée, Sec. 6.2) is as input-output functions on states (or, equivalently, relations on states). We present such a semantics for our language in Fig. 2.6. Given a set R of environments before a statement *stat* is executed, $\mathbb{S}[\![\ stat\ ]\!]$ computes the set of environments reached at the end of the statement. Moreover, given a set of error locations O, it returns O enriched with the location of all the errors encountered while executing the statement in an environment in R. The join $\sqcup$ we use corresponds to the pair-wise set union of environment sets and error location sets: $\langle V_1,\, O_1\rangle \sqcup \langle V_2,\, O_2\rangle \stackrel{\text{def}}{=} \langle V_1 \cup V_2,\, O_1 \cup O_2\rangle$. The semantics of a program $P \in \mathit{stat}$ is then:

$$\mathbb{P} \stackrel{\text{def}}{=} \mathbb{S}[\![\, P \,]\!]\, \langle \{\, \lambda V.\, 0 \,\},\, \emptyset\rangle \ . \tag{2.9}$$

There are several points of note. Firstly, this is a *big-step semantics*: it does not record the states at intermediate syntactic locations (although errors occurring at intermediate statements are recorded and appear in the output). As a consequence, the presentation in Fig. 2.6 completely dispenses from statement locations. Secondly, it involves a least fixpoint for each program loop. Each such fixpoint computes a loop invariant, corresponding to the syntactic location named ℓ_i in Fig. 2.1, which is then filtered by the loop exit condition to obtain the environments reachable at the end of the loop. For any statement, $\mathbb{S}[\![\ stat\ ]\!]$ is a join-morphism in the product of powerset complete lattices $\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega)$, which justifies the existence of the fixpoint. Finally, $\mathbb{P}$ outputs the set of environments at the end of the program, and the set of errors that can be encountered at any point during the execution of the program. For instance, in case of non-termination, $\mathbb{P}$ will output an empty set of environments but nevertheless includes all the errors that may occur in the program.

The equivalence between the big-step semantics $\mathbb{S}[\![\ stat\ ]\!]$ and the reachable state semantics $\mathcal{R}$ is proved in [Min12d] and relies on a notion of control paths (we omit the proof here but we will nevertheless introduce control paths in Sec. 3.3, as they are also useful to study concurrent programs). As a last remark, note that the big-step semantics is similar to Scott's denotational semantics as both view program semantics as input-output functions (although our semantics is far

simpler as we only consider first-order programs). The connection is stated formally by Cousot in [Cou02] and further explored by Schmidt in [Sch09].

2.3.6 Environment abstraction

In order to construct a computable and efficient semantics able to reason about the reachable states, we now abstract the semantic domain $\mathcal{P}(\Sigma)$. More precisely, we abstract the environment sets involved in the equational and big-step semantics; we do not abstract the (finite) sets $\mathcal{L}$, $\mathcal{V}$, nor Ω, so that the resulting analysis remains flow-sensitive, field-sensitive, and precise about the location of errors that can occur.

We first focus on inferring properties of environments (in $\mathcal{E} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{R}$) and ignore error inference for now. We thus start from a numeric abstract domain, which is a partially ordered set $(\mathcal{E}^\sharp, \sqsubseteq^\sharp_\mathcal{E})$ abstracting environment sets $(\mathcal{P}(\mathcal{E}), \subseteq)$, and featuring a monotonic concretization $\gamma_\mathcal{E} \in \mathcal{E}^\sharp \to \mathcal{P}(\mathcal{E})$. In order to abstract both the equational and big-step semantics, only a few abstract operators are actually needed:

- an assignment: $\mathbb{S}^\sharp_\mathcal{E}[\![X \leftarrow e]\!] \in \mathcal{E}^\sharp \to \mathcal{E}^\sharp$ abstracting the function $\mathbb{S}_\mathcal{E}[\![X \leftarrow e]\!]$ from Fig. 2.4;
- a filter: $\mathbb{S}^\sharp_\mathcal{E}[\![e \bowtie 0]\!] \in \mathcal{E}^\sharp \to \mathcal{E}^\sharp$ abstracting $\mathbb{S}_\mathcal{E}[\![e \bowtie 0]\!]$;
- a join: $\cup^\sharp_\mathcal{E} \in \mathcal{E}^\sharp \times \mathcal{E}^\sharp \to \mathcal{E}^\sharp$ abstracting $\cup$;
- a widening: $\triangledown_\mathcal{E} \in \mathcal{E}^\sharp \times \mathcal{E}^\sharp \to \mathcal{E}^\sharp$;
- an initial state: $E^\sharp_0 \in \mathcal{E}^\sharp$ abstracting $\{ \lambda V. 0 \}$.

These abstract operators must obey the soundness condition $f(\gamma(x^\sharp)) \subseteq \gamma(f^\sharp(x^\sharp))$ (2.1) and, for the widening, the termination condition we presented in Def. 2.2.1. Moreover, in order to construct an effective analyzer, we need to provide a data-structure to encode in a computer the elements from $\mathcal{E}^\sharp$, and algorithms to implement the abstract operators. Example numeric domains will be presented in Sec. 2.4; we assume for now that one is given and work from the operators it provides to derive an analysis in a generic way.

To handle errors, we additionally ask for an abstract operator $\mathbb{E}^\sharp_\Omega[\![e]\!] \in \mathcal{E}^\sharp \to \mathcal{P}(\Omega)$ that returns the errors encountered when evaluating the expression e in an abstract environment. The soundness condition is thus:

$$\mathbb{E}^\sharp_\Omega[\![e]\!] R^\sharp \supseteq \bigcup_{\rho \in \gamma_\mathcal{E}(R^\sharp)} snd(\mathbb{E}[\![e]\!]\rho) \ . \tag{2.10}$$

Our abstraction of $\mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega)$ is then $\mathcal{D}^\sharp \stackrel{\text{def}}{=} \mathcal{E}^\sharp \times \mathcal{P}(\Omega)$, with order $\langle R^\sharp_1, O_1 \rangle \sqsubseteq^\sharp \langle R^\sharp_2, O_2 \rangle \stackrel{\text{def}}{\iff} R^\sharp_1 \sqsubseteq^\sharp_\mathcal{E} R^\sharp_2 \wedge O_1 \subseteq O_2$ and concretization $\gamma \stackrel{\text{def}}{=} \lambda \langle R^\sharp, O \rangle. \langle \gamma_\mathcal{E}(R^\sharp), O \rangle$. Sound operators on $\mathcal{D}^\sharp$ are derived systematically from those on $\mathcal{E}^\sharp$:

- assignment: $\mathbb{S}^\sharp[\![X \leftarrow e]\!] \langle R^\sharp, O \rangle \stackrel{\text{def}}{=} \langle \mathbb{S}^\sharp_\mathcal{E}[\![X \leftarrow e]\!] R^\sharp, O \cup \mathbb{E}^\sharp_\Omega[\![e]\!] R^\sharp \rangle$;
- filter: $\mathbb{S}^\sharp[\![e \bowtie 0]\!] \langle R^\sharp, O \rangle \stackrel{\text{def}}{=} \langle \mathbb{S}^\sharp_\mathcal{E}[\![e \bowtie 0]\!] R^\sharp, O \cup \mathbb{E}^\sharp_\Omega[\![e]\!] R^\sharp \rangle$;
- join: $\langle R^\sharp_1, O_1 \rangle \sqcup^\sharp \langle R^\sharp_2, O_2 \rangle \stackrel{\text{def}}{=} \langle R^\sharp_1 \cup^\sharp_\mathcal{E} R^\sharp_2, O_1 \cup O_2 \rangle$;
- widening: $\langle R^\sharp_1, O_1 \rangle \triangledown \langle R^\sharp_2, O_2 \rangle \stackrel{\text{def}}{=} \langle R^\sharp_1 \triangledown_\mathcal{E} R^\sharp_2, O_1 \cup O_2 \rangle$;
- initial state: $D^\sharp_0 \stackrel{\text{def}}{=} \langle E^\sharp_0, \emptyset \rangle$.

Abstract equational semantics. A static analyzer based on equation systems can then be constructed in three steps. Firstly, we construct an abstract equation system, featuring a family of variables $(\mathcal{X}^\sharp_\ell)_{\ell \in \mathcal{L}}$ with value in $\mathcal{E}^\sharp$ and equations of the form $\mathcal{X}^\sharp_\ell = F^\sharp_\ell(\mathcal{X}^\sharp_{\ell_1}, \ldots, \mathcal{X}^\sharp_{\ell_n})$. This is easily done by replacing occurrences of concrete operators $\mathbb{S}_\mathcal{E}[\![\ ]\!]$ and $\cup$ in Fig. 2.4 with their abstract versions $\mathbb{S}^\sharp_\mathcal{E}[\![\ ]\!]$ and $\cup^\sharp_\mathcal{E}$. Secondly, we insert widenings in order ensure that the system is solvable with finite iterations. This is done by replacing equations $\mathcal{X}^\sharp_\ell = F^\sharp_\ell(\mathcal{X}^\sharp_{\ell_1}, \ldots, \mathcal{X}^\sharp_{\ell_n})$ with $\mathcal{X}^\sharp_\ell = \mathcal{X}^\sharp_\ell \triangledown F^\sharp_\ell(\mathcal{X}^\sharp_{\ell_1}, \ldots, \mathcal{X}^\sharp_{\ell_n})$. Widenings need not be inserted at all syntactic locations; it is sufficient to ensure that each dependency cycle in the equation system traverses a widening point. Given the very structured nature of our language, a natural choice is to widen at syntactic locations ℓ_i, corresponding to loop invariants. Finally, we must devise an iteration scheme. A simple idea is to use a work-list based algorithm. Other iteration schemes and choices of widening points exist. They may have an impact on efficiency, but also on precision (we refer the reader to Bourdoncle [Bou93] for an in-depth presentation).

This presentation of a static analyzer in equational form is reminiscent of forward data-flow analyses [Kil73], but it is more powerful as it allows infinite-height abstract domains.

$$\underline{\mathbb{S}^\sharp[\![\mathit{stat}]\!] \in (\mathcal{E}^\sharp \times \mathcal{P}(\Omega)) \to (\mathcal{E}^\sharp \times \mathcal{P}(\Omega))}$$

$$\mathbb{S}^\sharp[\![\textbf{if } e \bowtie 0 \textbf{ then } s \textbf{ endif}]\!]\mathcal{X}^\sharp \stackrel{\text{def}}{=} (\mathbb{S}^\sharp[\![s]\!] \circ \mathbb{S}^\sharp[\![e \bowtie 0]\!])\mathcal{X}^\sharp \sqcup^\sharp \mathbb{S}^\sharp[\![e \not\bowtie 0]\!]\mathcal{X}^\sharp$$

$$\mathbb{S}^\sharp[\![\textbf{while } e \bowtie 0 \textbf{ do } s \textbf{ done}]\!]\mathcal{X}^\sharp \stackrel{\text{def}}{=} \mathbb{S}^\sharp[\![e \not\bowtie 0]\!](\lim \lambda \mathcal{Y}^\sharp. \mathcal{Y}^\sharp \triangledown (\mathcal{X}^\sharp \sqcup^\sharp (\mathbb{S}^\sharp[\![s]\!] \circ \mathbb{S}^\sharp[\![e \bowtie 0]\!])\mathcal{Y}^\sharp))$$

$$\mathbb{S}^\sharp[\![s_1 ; s_2]\!] \stackrel{\text{def}}{=} \mathbb{S}^\sharp[\![s_2]\!] \circ \mathbb{S}^\sharp[\![s_1]\!]$$

Figure 2.7: Abstract big-step semantics.

Abstract big-step semantics. A *big-step static analyzer* is even simpler to construct. It is sufficient to replace concrete operators with abstract ones in the semantics of Fig. 2.6 and insert a widening at each fixpoint computation, in the semantics of loops. The resulting semantics is shown in Fig. 2.7. The notation $\lim \lambda \mathcal{Y}^\sharp. \mathcal{Y}^\sharp \triangledown F^\sharp(\mathcal{Y}^\sharp)$ denotes the limit reached (in finite time) by iterating $F^\sharp$ with a widening.

The big-step presentation is appealing for two reasons. A first reason is that it stays very close to the structure of the program, following its control flow. A big-step static analyzer is an abstract interpretation in a literal way: it interprets the program, but manipulates an abstract environment representing many concrete environments, instead of a single one. A second reason is that it makes a parsimonious use of abstract elements: while the equational form maintains an abstract element for each syntactic location in $\mathcal{L}$ at all time, the big-step semantics forgets all the intermediate steps. More precisely, as the semantics is defined by induction on the program structure, the memory requirement is linear in the depth of the syntax tree (the number of nested conditionals and loops) instead of being linear in its size. The associated gain in memory is critical when analyzing large programs with complex abstract domains.

There are, however, two associated drawbacks. Firstly, unlike equation-based semantics, we do not have any freedom in the iteration scheme: it is fixed by the syntax of the program. Secondly, it often performs superfluous computations. For instance, when encountering nested loops, the inner loop is re-analyzed fully for each iteration of the outer loop, even if the abstract elements have not changed. By comparison, an equation solver based on a work-list algorithm would avoid such computations. These drawbacks can be mitigated: there is experimental evidence [Bou93] that the iteration order fixed

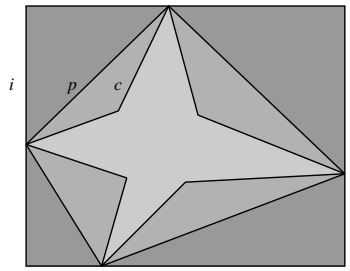


Figure 2.8: Abstraction of a star-shaped concrete element c in the polyhedra p and interval i domains.

$$\mathcal{D}_i^\sharp \stackrel{\text{def}}{=} (\mathcal{V} \to \mathcal{I}) \cup \{\bot^\sharp\}$$
where $\mathcal{I} \stackrel{\text{def}}{=} \{\, [a,b] \mid a \in \mathbb{R} \cup \{-\infty\},\, b \in \mathbb{R} \cup \{+\infty\},\, a \leq b \,\}$

$$R_1^\sharp \sqsubseteq_i^\sharp R_2^\sharp \iff \begin{cases} R_1^\sharp = \bot^\sharp & \text{or} \\ \forall V :\, R_1^\sharp(V) \subseteq R_2^\sharp(V) & \text{if } R_1^\sharp, R_2^\sharp \neq \bot^\sharp \end{cases}$$

$$\alpha_i(R) \stackrel{\text{def}}{=} \begin{cases} \bot^\sharp & \text{if } R = \emptyset \\ \lambda V.\, [\min\{\rho(V) \mid \rho \in R\},\, \max\{\rho(V) \mid \rho \in R\}] & \text{if } R \neq \emptyset \end{cases}$$

$$\gamma_i(R^\sharp) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } R^\sharp = \bot^\sharp \\ \{\, \rho \in \mathcal{E} \mid \forall V :\, \rho(V) \in R^\sharp(V) \,\} & \text{if } R^\sharp \neq \bot^\sharp \end{cases}$$

Figure 2.9: Interval abstract elements.

by the program syntax is often optimal, and redundant computations can be avoided by caching results, in particular loop invariants of inner loops, which trades memory for speed.

2.4 Numeric abstractions

In this section, we recall in some details two well-known numeric domains: intervals and polyhedra. They are very classic, dating from the early days of abstract interpretation. They are also the foundation upon which we develop new domains in Chaps. 4 and 5. Additionally, we introduce here floating-point numbers, that will be considered in those chapters.

Many numeric abstract domains have been proposed in the literature. We refer the reader to [Min04b, §2.4.5] for an overview. They vary in their expressiveness as well as their cost versus precision trade-off. For instance, intervals are very cheap but not very precise, while polyhedra are more expressive, more precise, and more expensive. Figure 2.8 presents how intervals and polyhedra over-approximate, more or less tightly, the same star-shaped concrete domain.

2.4.1 Intervals

The *interval abstraction* consists in inferring, for each variable, an upper and a lower bound on its possible values. It was introduced early by Cousot and Cousot [CC76] and it is still widely used as it is efficient and yet able to provide valuable information on program executions. Bound properties are useful, for instance, to prove the absence of arithmetic overflow or out-of-bound array access.

$$R_1^\sharp \cup_i^\sharp R_2^\sharp \stackrel{\text{def}}{=} \begin{cases} R_2^\sharp & \text{if } R_1^\sharp = \bot^\sharp \\ R_1^\sharp & \text{if } R_2^\sharp = \bot^\sharp \\ \lambda V.\ \ [\min(R_1^\sharp(V), R_2^\sharp(V)), \\ \qquad \max(R_1^\sharp(V), R_2^\sharp(V))] & \text{otherwise} \end{cases}$$

$$R_1^\sharp \triangledown_i R_2^\sharp \stackrel{\text{def}}{=} \begin{cases} R_2^\sharp & \text{if } R_1^\sharp = \bot^\sharp \\ R_1^\sharp & \text{if } R_2^\sharp = \bot^\sharp \\ \lambda V.\, R_1^\sharp(V) \triangledown R_2^\sharp(V) & \text{otherwise} \end{cases}$$
where:
$$[a,b] \triangledown [c,d] \stackrel{\text{def}}{=} \left[\begin{cases} a & \text{if } a \leq c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ +\infty & \text{otherwise} \end{cases} \right]$$

Figure 2.10: Interval join and widening operators.

The domain of abstract elements $\mathcal{D}_i^\sharp$ is formally presented in Fig. 2.9, with its order $\sqsubseteq_i^\sharp$, its abstraction function α_i, and its concretization γ_i (which forms a Galois injection). It is simply a point-wise extension over $\mathcal{V}$ of the interval domain from Ex. 2.2.1 where least elements $\bot^\sharp$, representing empty intervals, coalesce into a single least element representing the empty set of environments.

We present in Fig. 2.10 the join abstraction $\cup_i^\sharp$, which is optimal but not exact: joining $[1,2]$ with $[4,5]$ yields $[1,5]$, which contains spurious values, such as 3. Note that $\mathcal{D}_i^\sharp$ is actually a complete lattice and $\cup_i^\sharp$ is its least upper bound. The interval widening $\triangledown_i$, presented in Fig. 2.10, is similar to the join (which it over-approximates) but ensures termination by replacing unstable upper bounds with $+\infty$ and lower bounds with $-\infty$, so that intervals cannot grow indefinitely.

Finally, we define an abstract assignment operator. As $\mathcal{D}_i^\sharp$ enjoys a Galois connection, it is possible to define semantically the best abstraction as $\mathbb{S}_i^\sharp[\![\, X \leftarrow e \,]\!] \stackrel{\text{def}}{=} \alpha_i \circ \mathbb{S}[\![\, X \leftarrow e \,]\!] \circ \gamma_i$. However, this does not provide an algorithm to compute it. Thus, we opt for an alternate definition based on abstract expression evaluation. We modify the concrete semantics of expressions from Fig. 2.2 so that it takes as argument a map from variables to intervals and outputs a single interval as well as a set of runtime errors. Moreover, we replace each concrete operator $\circ$ or $\diamond$ on reals with an abstract operator $\circ_i^\sharp$ or $\diamond_i^\sharp$ on intervals, and take care of detecting and propagating divisions by zero. The corresponding definitions are given in Fig. 2.11.

Remark. The semantics of Fig. 2.2 is sound but not the best abstraction. The loss of precision comes from handling different occurrences of the same variable as distinct variables. For instance, we have $\mathbb{E}_i^\sharp[\![\, X - _\omega X \,]\!][X \mapsto [-1,1]] = \langle [-2,2], \emptyset \rangle$, while, in fact, this expression evaluates to 0, which is exactly representable as an interval.
End of remark.

It is possible to design an abstract filter operator $\mathbb{S}_i^\sharp[\![\, e \bowtie 0 \,]\!]$ along the same principles. This is slightly complicated by the fact that we must evaluate expressions backward, in order to infer intervals for variables at the leaves of the expression tree given the interval of the intended result of the whole expres-

$$\mathbb{S}^\sharp_i[\![X \leftarrow e]\!]\langle R^\sharp, O\rangle \stackrel{\text{def}}{=} \text{let } \langle I, O'\rangle = \mathbb{E}^\sharp_i[\![e]\!] R^\sharp \text{ in } \begin{cases}\langle \bot^\sharp, O \cup O'\rangle & \text{if } I = \bot^\sharp \\ \langle R^\sharp[X \mapsto I], O \cup O'\rangle & \text{if } I \neq \bot^\sharp\end{cases}$$

where :

$$\mathbb{E}^\sharp_i[\![X]\!]R^\sharp \stackrel{\text{def}}{=} \langle R^\sharp(X), \emptyset\rangle$$

$$\mathbb{E}^\sharp_i[\![[c_1, c_2]]\!]R^\sharp \stackrel{\text{def}}{=} \langle [c_1, c_2], \emptyset\rangle$$

$$\mathbb{E}^\sharp_i[\![\circ_\omega e]\!]R^\sharp \stackrel{\text{def}}{=} \text{let } \langle I, O\rangle = \mathbb{E}^\sharp_i[\![e]\!] R^\sharp \text{ in } \langle \circ^\sharp_i I, O\rangle$$

$$\mathbb{E}^\sharp_i[\![e_1 \diamond_\omega e_2]\!]R^\sharp \stackrel{\text{def}}{=} \text{let } \langle I_1, O_1\rangle = \mathbb{E}^\sharp_i[\![e_1]\!] R^\sharp \text{ in let } \langle I_2, O_2\rangle = \mathbb{E}^\sharp_i[\![e_2]\!] R^\sharp \text{ in } \langle I_1 \diamond^\sharp_i I_2, O_1 \cup O_2 \cup \{ \omega \mid \diamond = / \wedge 0 \in I_2 \}\rangle$$

where :

$$-^\sharp_i [a, b] \stackrel{\text{def}}{=} [-b, -a]$$

$$[a, b] +^\sharp_i [c, d] \stackrel{\text{def}}{=} [a + c, b + d]$$

$$[a, b] -^\sharp_i [c, d] \stackrel{\text{def}}{=} [a - d, b - c]$$

$$[a, b] \times^\sharp_i [c, d] \stackrel{\text{def}}{=} [\min(ac, bc, ad, bd), \max(ac, bc, ad, bd)]$$

$$[a, b]/^\sharp_i[c, d] \stackrel{\text{def}}{=} \begin{cases}\bot^\sharp & \text{if } c = d = 0 \\ [\min(a/c, a/d, b/c, b/d), \max(a/c, a/d, b/c, b/d)] & \text{else if } 0 \leq c \\ [-b, -a]/^\sharp_i[-d, -c] & \text{else if } d \leq 0 \\ ([a, b]/^\sharp_i[c, 0]) \cup^\sharp_i ([a, b]/^\sharp_i[0, d]) & \text{otherwise}\end{cases}$$

$$\forall R^\sharp : \circ^\sharp_i \bot^\sharp \stackrel{\text{def}}{=} \bot^\sharp \diamond^\sharp_i R^\sharp \stackrel{\text{def}}{=} R^\sharp \diamond^\sharp_i \bot^\sharp \stackrel{\text{def}}{=} \bot^\sharp$$

Figure 2.11: Interval assignment.

sion (i.e., at the root of the expression tree). We do not present it here; we refer instead the reader to [Min04b] for an example algorithm that combines forward and backward abstract evaluations in a way reminiscent to the HC4-revise algorithm [BGGP99] used in constraint programming.

Example 2.4.1. On the program example of Fig. 2.5, an analysis based on the interval domain will be able to prove that $i \geq 2$ at the end of the program, at $\ell 9$. In this case, we find the best interval abstraction of the concrete result $i = \max(2, n)$.
End of example.

Remark. Our intervals use real bounds, and so, do not directly provide an effective computer representation nor algorithms. In practice, we use machine representable bounds, which leads to a slightly weaker domain. For instance, intervals with rational bounds of arbitrary precision (augmented with $+\infty$ and $-\infty$) lack the abstraction function α_i.[4] Another solution, discussed in Sec. 2.4.4, consists in using floating-point bounds, which leads to an efficient implementation but also to some precision degradation due to rounding errors.
End of remark.

2.4.2 Polyhedra

The *polyhedra domain* was introduced by Cousot and Halbwachs in [CH78] to infer affine inequalities on program variables.

Polyhedra are much more expressive than intervals; in particular, they are *relational* (they can express relationships between variables). They are also more precise, even in the context of inferring variable bounds, as they can compensate from a loss of precision in the interval domain due to non-optimal combinations of operators, the need for relational invariants locally, or the use of a widening.

Example 2.4.2. In order to infer invariants of a certain form at the end of a loop, it is often necessary to infer loop invariants of a strictly more complex form. Consider the simple loop:

while $i < 5000$ **do**
  $i \leftarrow i + 1$;
  if $[0, 1] = 0$ **then** $x \leftarrow x + 1$ **endif**
done

An interval analysis with widening (and a decreasing iteration) will infer that $i = 5000$ and $x \geq 0$ when the loop terminates, but it will not find any upper bound on x because it is never tested explicitly. An analysis with polyhedra will infer that $x \leq 5000$ because it is able to infer the loop invariant $x \leq i$, and the loop exit condition on i will then refine the value of x.

End of example.

Polyhedra are based on the theory of linear algebra. In the following, we assimilate the set of environments $\mathcal{V} \to \mathbb{R}$ to a vector space $\mathbb{R}^n$ by fixing an order on the set of variables $\mathcal{V} \stackrel{\text{def}}{=} \{ V_1, \ldots, V_n \}$. We thus denote an environment $\rho \in \mathcal{V} \to \mathbb{R}$ as a vector $\vec{V} \stackrel{\text{def}}{=} (\rho(V_1), \ldots, \rho(V_n))$.

Double description method

Representation. Semantically, the elements of the polyhedra abstract domain $\mathcal{D}^\sharp_p$ are closed, convex (and possibly unbounded) polyhedra of $\mathbb{R}^n$. There exists two convenient syntactic representations for polyhedra:

- as a finite set of *affine constraints* $\mathcal{C} = \{ \sum_{i=1}^n A_{1i}V_i \leq B_1, \ldots, \sum_{i=1}^n A_{mi}V_i \leq B_m \}$, which we also denote as a pair $\langle \mathbf{A}, \vec{B}\rangle$ composed of a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and a vector $\vec{B} \in \mathbb{R}^m$;
- as a finite set of vector *generators*: points $\{\vec{P}_1, \ldots, \vec{P}_p\}$ and *rays* $\{ \vec{R}_1, \ldots, \vec{R}_r \}$, which we denote as a pair $[\mathbf{P}, \mathbf{R}]$ of matrices $\mathbf{P} \in \mathbb{R}^{n \times p}$ and $\mathbf{R} \in \mathbb{R}^{n \times r}$.

The concretization of a set of constraints is the set of vectors satisfying all the constraints, while the concretization of a set of generators is the sum of a *convex combination* of its points and a *conical combination* of its rays (allowing unbounded polyhedra):

$$\gamma_p(\mathcal{C}) \stackrel{\text{def}}{=} \gamma_p(\langle \mathbf{A}, \vec{B}\rangle) \stackrel{\text{def}}{=} \{ \vec{V} \mid \mathbf{A} \times \vec{V} \leq \vec{B} \}$$

$$\gamma_p([\mathbf{P}, \mathbf{R}]) \stackrel{\text{def}}{=} \{ (\textstyle\sum_{j=1}^p \alpha_j \vec{P}_j) + (\sum_{j=1}^r \beta_j \vec{R}_j) \mid \forall j : \alpha_j \geq 0, \beta_j \geq 0, \sum_{j=1}^p \alpha_j = 1 \} \ . \tag{2.11}$$

Note that there is no abstraction function: some vector sets do not have a best over-approximation as a convex closed polyhedron (such as the disk $X^2 + Y^2 \leq 1$ which can be defined as the limit of infinitely many polyhedra, none of which is optimal). Additionally, syntactic representations are not

[4] Indeed, some subsets of $\mathbb{Q}$ have no least upper bound. This is the case for instance of $A \stackrel{\text{def}}{=} \{ x \in \mathbb{Q} \mid x^2 \leq 2 \}$, and so $\alpha_i(A) \stackrel{\text{def}}{=} [\min A, \max A]$ is not an interval with rational bounds.

$$[\mathbf{P}, \mathbf{R}] \sqsubseteq_p^\sharp \langle \mathbf{A}, \vec{B} \rangle \iff^{\text{def}} \forall i : \mathbf{A} \times \vec{P}_i \leq \vec{B} \wedge \forall i : \mathbf{A} \times \vec{R}_i \leq \vec{0}$$

$$P^\sharp =_p^\sharp Q^\sharp \iff^{\text{def}} P^\sharp \sqsubseteq_p^\sharp Q^\sharp \wedge Q^\sharp \sqsubseteq_p^\sharp P^\sharp$$

$$[\mathbf{P_1}, \mathbf{R_1}] \cup_p^\sharp [\mathbf{P_2}, \mathbf{R_2}] \stackrel{\text{def}}{=} [\mathbf{P_1}\,\mathbf{P_2}, \mathbf{R_1}\,\mathbf{R_2}]$$

$$\mathbb{S}_p^\sharp[\![\, \vec{A} \cdot \vec{V} + b \leq 0 \,]\!]\mathcal{C} \stackrel{\text{def}}{=} \mathcal{C} \cup \{\, \vec{A} \cdot \vec{V} + b \leq 0 \,\}$$

$$\mathbb{S}_p^\sharp[\![\, \vec{A} \cdot \vec{V} + [b, c] \leq 0 \,]\!] \stackrel{\text{def}}{=} \mathbb{S}_p^\sharp[\![\, \vec{A} \cdot \vec{V} + b \leq 0 \,]\!]$$

$$\mathbb{S}_p^\sharp[\![\, e = 0 \,]\!] \stackrel{\text{def}}{=} \mathbb{S}_p^\sharp[\![\, e \leq 0 \,]\!] \circ \mathbb{S}_p^\sharp[\![\, -e \leq 0 \,]\!]$$

when e is not affine: $\mathbb{S}_p^\sharp[\![\, e \bowtie 0 \,]\!] \stackrel{\text{def}}{=} \lambda P^\sharp.\, P^\sharp$

$$\mathbb{S}_p^\sharp[\![\, V_i \leftarrow [-\infty, +\infty] \,]\!][\mathbf{P}, \mathbf{R}] \stackrel{\text{def}}{=} [\mathbf{P}, \mathbf{R}\,(\vec{e}_i)\,(-\vec{e}_i)]$$

$$\mathbb{S}_p^\sharp[\![\, V_i \leftarrow e \,]\!] \stackrel{\text{def}}{=} [V_{n+1}/V_i] \circ \mathbb{S}_p^\sharp[\![\, V_i \leftarrow [-\infty, +\infty] \,]\!] \circ \mathbb{S}_p^\sharp[\![\, V_{n+1} - e = 0 \,]\!]$$

$$\mathcal{C}_1 \triangledown_p \mathcal{C}_2 \stackrel{\text{def}}{=} \{\, c \in \mathcal{C}_1 \mid \mathcal{C}_2 \sqsubseteq_p^\sharp \{c\} \,\} \cup \{\, c_2 \in \mathcal{C}_2 \mid \exists c_1 \in \mathcal{C}_1 : \mathcal{C}_1 =_p^\sharp (\mathcal{C}_1 \setminus \{c_1\}) \cup \{c_2\} \,\}$$

Figure 2.12: Double description polyhedral operators.

unique. For instance, the constraint sets $\{\, X - Y \leq 0, X \leq 0, Y \leq 0 \,\}$ and $\{\, X - Y \leq 0, Y \leq 0 \,\}$ represent the same polyhedron: the constraint $X \leq 0$ is redundant in the former polyhedron and can be removed. In order to improve the efficiency in memory, it is desirable to remove redundant constraints. Nevertheless, two minimal constraint sets (i.e., where no constraint is redundant) can represent the same polyhedron: both $\{\, X \leq 0, -X \leq 0, Y \leq 0, -Y \leq 0 \,\}$ and $\{\, X - Y \leq 0, X + Y \leq 0, -X \leq 0 \,\}$ represent the point $(0, 0)$. The algorithms defined below compute on the syntactic representation of polyhedra. However, any such algorithm $f^\sharp$ will satisfy $\gamma_p(P^\sharp) = \gamma_p(Q^\sharp) \implies \gamma_p(f^\sharp(P^\sharp)) = \gamma_p(f^\sharp(Q^\sharp))$; hence, we argue that they have a semantic meaning at the level of polyhedra in $\mathcal{D}_p^\sharp$, irrespective of the chosen representation.

Polyhedral abstract operators are often much easier to define on one representation than on the other. Hence, the benefit of the double description method is to simplify the design of the domain by reducing its complexity to a single operation: converting from one representation to the another one when more convenient. The standard conversion algorithm is due to Chernikova and later improved by LeVerge [LeV92]. In addition to converting, it also minimizes the output representation. Modern versions are highly optimized and quite complex; we do not discuss them here and refer instead the reader to [LeV92].

Abstract operators. Assuming that both the constraint and the generator representations are available, we present the polyhedra abstract operators in Fig. 2.12. The partial order $\sqsubseteq_p^\sharp$ precisely models polyhedra inclusion: $P^\sharp \sqsubseteq_p^\sharp Q^\sharp \iff \gamma_p(P^\sharp) \subseteq \gamma_p(Q^\sharp)$. It is thus possible to test for the semantic equality of syntactic representations, which we denote as $=_p^\sharp$, by double inclusion with respect to $\sqsubseteq_p^\sharp$. The polyhedra join $\cup_p^\sharp$ joins generators to compute the topological closure of the convex hull of its arguments, which is the smallest convex closed polyhedron containing both its arguments. An affine inequality test $\vec{A} \cdot \vec{V} + b \leq 0$ is modeled directly by constraint addition, which is an exact abstraction. Variants, such as $\vec{A} \cdot \vec{V} + [b, c] \leq 0$ and $\vec{A} \cdot \vec{V} + b = 0$, can be reduced to the affine inequality case. A non-deterministic assignment $V_i \leftarrow [-\infty, +\infty]$ is modelled by adding, as rays, the basis vectors $\vec{e}_i$ and $-\vec{e}_i$ corresponding to the variable V_i and its opposite. Assignments $V_i \leftarrow e$ are handled in three steps: firstly, a fresh variable V_{n+1} is used to hold the value of the expression e, then the "old" value of V_i is forgotten by a non-deterministic assignment and, finally, the new variable V_{n+1} is renamed into V_i by substitution in the constraint set, which is denoted as $[V_{n+1}/V_i]$.[5] This three-step operation is required in case V_i appears also in the assigned expression. When e is affine, all three operations are exact, so the affine assignment is also exact. Abstracting precisely non-affine assignments and tests is more challenging. In Fig. 2.12, we choose to model non-affine tests as the identity and, as a consequence, non-affine assignments reduce to the non-deterministic assignment $V_i \leftarrow [-\infty, +\infty]$, which is sound but not very precise. An alternate solution would be to use the interval domain locally, by converting the polyhedron to its bounding-box, applying the interval operation, and incorporating the range of the result into the original polyhedron. Another solution is to abstract expressions themselves, as we will discuss in Sec. 2.4.3. Finally, a simple solution to compute errors $\mathbb{E}_\Omega^\sharp[\![\, e \,]\!]$ in expressions is to also rely on the interval domain, applied on the bounding box of the polyhedron.

The final operator, widening $\mathcal{C}_1 \triangledown \mathcal{C}_2$, is defined intuitively by removing the constraints in $\mathcal{C}_1$ that are not satisfied by $\mathcal{C}_2$. There are, however, two subtle issues [BHRZ05]. Firstly, in order to terminate, the arguments must not contain redundant constraints. Secondly, in order to be independent from the choice of constraint representation, we must add constraints from $\mathcal{C}_2$ that are redundant with a constraint from $\mathcal{C}_1$, i.e., any $c_2 \in \mathcal{C}_2$ such that $\exists c_1 \in \mathcal{C}_1 : \mathcal{C}_1 =_p^\sharp (\mathcal{C}_1 \setminus \{c_1\}) \cup \{c_2\}$.

Remark. As for intervals, implementations of polyhedra do not use reals but computer-representable numbers. Traditionally, exact rational arithmetic (requiring arbitrary precision numbers) is used. Achieving soundness with inexact data-types, such as floating-point numbers is not straightforward; this is one of our contribution, discussed in Chap. 4.
End of remark.

Constraint-only method

A drawback of polyhedra is that their representation is unbounded: one can construct (minimal) polyhedra consisting of arbitrary many constraints or generators. This is exacerbated by the fact that one representation can be exponentially larger than another. In particular, a simple box $\wedge_i a_i \leq V_i \leq b_i$, expressed as only $2|\mathcal{V}|$ constraints, requires $2^{|\mathcal{V}|}$ generators (one for each corner of the box). To avoid such pathological cases, Simon and King [SK05] suggest abandoning the double description method and construct a polyhedra domain using solely constraints. In order to remove the need for generators, it is sufficient to provide a way to perform the following four operators using constraints only:

1. compute $\sqsubseteq_p^\sharp$;
2. remove redundant constraints;
3. compute $\mathbb{S}_p^\sharp[\![\, V_i \leftarrow [-\infty, +\infty] \,]\!]$;
4. compute $\cup_p^\sharp$.

The first two steps can be achieved using linear programming, and the last two using projection.

[5] After the non-deterministic assignment, V_i should not occur in the constraint set, so that all the occurrences of V_i in the result only come from that of V_{n+1}.

Linear programming. Given a polyhedron $\mathcal{C}$ in constraint form and a vector $\vec{A} \in \mathbb{R}^n$, the *linear programming* problem $LP(\mathcal{C}, \vec{V})$ consists in computing the minimum of $\vec{A} \cdot \vec{V}$ when $\vec{V}$ ranges in the polyhedron:

$$LP(\mathcal{C}, \vec{A}) \stackrel{\text{def}}{=} \min \{ \vec{A} \cdot \vec{V} \mid \vec{V} \in \gamma_p(\mathcal{C}) \} \ . \tag{2.12}$$

Efficient algorithms, such as the *Simplex algorithm*, exist to compute LP; we refer the reader to [Sch86] for more information on the algorithmic aspect. Linear programming can be used to model constraint entailment, and so, to compute $\sqsubseteq_p^\sharp$:

$$\begin{array}{l} \mathcal{C} \sqsubseteq_p^\sharp \{ \vec{A} \cdot \vec{V} \leq b \} \iff LP(\mathcal{C}, -\vec{A}) + b \geq 0 \\ \mathcal{C}_1 \sqsubseteq_p^\sharp \mathcal{C}_2 \stackrel{\text{def}}{\iff} \forall c \in \mathcal{C}_2 : \mathcal{C}_1 \sqsubseteq_p^\sharp \{c\} \ . \end{array} \tag{2.13}$$

A constraint $c = (\vec{A} \cdot \vec{V} \leq b) \in \mathcal{C}$ is redundant if and only if $\mathcal{C} \setminus \{c\} \sqsubseteq_p^\sharp \{c\}$, i.e. if $LP(\mathcal{C} \setminus \{c\}, -\vec{A}) + b \geq 0$. A simple algorithm to minimize constraint sets is to remove redundant constraints one by one until no more can be removed[6] ([SK05] proposes several efficiency improvements, such as testing for syntactic redundancy and testing against a bounding box, which are fast operations, before executing a linear programing check, which is slower; we also refer the reader to [HLL92, Imb93] on related optimizations).

Projection. Forgetting the value of a variable V_i can be achieved by eliminating all the occurrences of V_i in the constraint set $\mathcal{C}$, i.e., it is a projection. Note, however, that adding to $\mathcal{C}$ any conical combination (i.e., with positive coefficients) of constraints from $\mathcal{C}$ does not change its concretization, and it is possible to find combinations where V_i does not occur although V_i occurs in the constraints that are combined. To avoid losing any information not related to V_i in the forget operation, we need to take such constraints into account. This leads to *Fourier–Motzkin's elimination* algorithm, which combines pairs of constraints where the coefficients of V_i have opposite signs, in order to eliminate V_i, and keeps constraints with a null coefficient for V_i:

$$\begin{array}{l} FM(\mathcal{C}, V_i) \stackrel{\text{def}}{=} \\ \quad \{ (\vec{A} \cdot \vec{V} \leq b) \in \mathcal{C} \mid A_i = 0 \} \cup \\ \quad \{ A_i^+ c^- - A_i^- c^+ \mid c^+ = (\vec{A^+} \cdot \vec{V} \leq b^+) \in \mathcal{C}, \\ \quad\ c^- = (\vec{A^-} \cdot \vec{V} \leq b^-) \in \mathcal{C},\ A_i^+ > 0,\ A_i^- < 0 \} \ . \end{array} \tag{2.14}$$

This is actually an exact abstraction of $\mathbb{S}[\![V_i \leftarrow [-\infty, +\infty]]\!]$.

Join. Benoy et al. proved in [BKM05] that computing the join $\cup_p^\sharp$ can be reduced to the projection. Given two polyhedra $\langle \mathbf{A}_1, \vec{B}_1 \rangle$ and $\langle \mathbf{A}_2, \vec{B}_2 \rangle$, a first step is to construct a polyhedron on the extended variable set $\{ V_i, V_i^1, V_i^2 \mid V_i \in \mathcal{V} \} \cup \{ \sigma^1, \sigma^2 \}$ combining all these constraints as follows:

$$\begin{array}{l} \mathcal{C} \stackrel{\text{def}}{=} \{ \ \vec{V} = \vec{V^1} + \vec{V^2},\ \sigma_1 + \sigma_2 = 1,\ \sigma_1 \geq 0,\ \sigma_2 \geq 0, \\ \qquad \mathbf{A}_1 \vec{V^1} \leq \vec{B}_1 \sigma_1,\ \mathbf{A}_2 \vec{V^2} \leq \vec{B}_2 \sigma_2 \} \ . \end{array} \tag{2.15}$$

It expresses that $\vec{V}$ should be a convex combination, with coefficients σ_1 and σ_2, of a point $\frac{1}{\sigma_1}\vec{V^1}$ in the first polyhedron and a point $\frac{1}{\sigma_2}\vec{V^2}$ in the second polyhedron. The second step is to eliminate all the variables except $\vec{V}$ using FM. As for redundancy removal, several techniques can be applied to improve the efficiency of the join operator, for instance avoiding the generation of constraints that are known to be redundant. We refer the reader to [SK05] for an in-depth presentation of the constraint-only domain and its various optimizations.

[6] It is not possible to remove all the redundant constraints at once because there can exist pairs of mutually redundant constraints; in that case, we should avoid removing both.

2.4.3 Linearization

The construction of abstract assignments and tests on non-relational domains, such as the interval domain (Fig. 2.11), is based on generic algorithms parametrized by abstractions of the operators used in expressions; as a consequence, these domains can handle expressions of any shape. This is not the case for relational domains. There, the shape of tests and assignments is tightly tied to the properties exactly representable in the domain: as polyhedra can only represent affine inequalities, the polyhedra domain naturally abstracts affine assignments and tests, and reverts to coarse fall-back operators (respectively the forget and the identity operators) in other cases. To go further, we proposed in [Min04b] a notion of *expression abstraction*, whose main application is to transform non-affine expressions into affine (or near affine) ones. We recall this technique as it will be useful to handle floating-point expressions in Sec. 2.4.4 and Chap. 4.

The core idea is to put expressions into affine form where constant coefficients are replaced with intervals, which we call *interval affine forms*:

$$e ::= [a_0, b_0] + \sum_{k=1}^{n} [a_k, b_k] V_k \ .$$

Our goal here is to benefit both from the algebraic properties of affine forms and from the abstracting power of intervals.

Affine form algebra. We define the addition $\boxplus$ and subtraction $\boxminus$ of two affine forms, and the multiplication $\boxtimes$ and division $\boxslash$ of an affine form by an interval as follows, using interval arithmetic operators from Fig. 2.11:

$$\begin{array}{l} ([a_0, b_0] + \sum_k [a_k, b_k] V_k) \boxplus ([a'_0, b'_0] + \sum_k [a'_k, b'_k] V_k) \stackrel{\text{def}}{=} \\ \qquad ([a_0, b_0] +_i^\sharp [a'_0, b'_0]) + \sum_k ([a_k, b_k] +_i^\sharp [a'_k, b'_k]) V_k \\ ([a_0, b_0] + \sum_k [a_k, b_k] V_k) \boxminus ([a'_0, b'_0] + \sum_k [a'_k, b'_k] V_k) \stackrel{\text{def}}{=} \\ \qquad ([a_0, b_0] -_i^\sharp [a'_0, b'_0]) + \sum_k ([a_k, b_k] -_i^\sharp [a'_k, b'_k]) V_k \\ ([a_0, b_0] + \sum_k [a_k, b_k] V_k) \boxtimes [a', b'] \stackrel{\text{def}}{=} \\ \qquad ([a_0, b_0] \times_i^\sharp [a', b']) + \sum_k ([a_k, b_k] \times_i^\sharp [a', b']) V_k \\ ([a_0, b_0] + \sum_k [a_k, b_k] V_k) \boxslash [a', b'] \stackrel{\text{def}}{=} \\ \qquad ([a_0, b_0] /_i^\sharp [a', b']) + \sum_k ([a_k, b_k] /_i^\sharp [a', b']) V_k \ . \end{array} \tag{2.16}$$

Expression abstraction. We say that e' abstracts e in an environment set $R \subseteq \mathcal{E}$, which we denote as $e \sqsubseteq_R e'$, if $\forall \rho \in R : \mathbb{E}[\![e]\!]\rho \sqsubseteq \mathbb{E}[\![e']\!]\rho$, i.e., e' evaluates to more values and errors. When $e \sqsubseteq_R e'$, then $\mathbb{S}[\![X \leftarrow e]\!]R$ and $\mathbb{S}[\![e \bowtie 0]\!]R$ can be safely replaced with $\mathbb{S}[\![X \leftarrow e']\!]R$ and $\mathbb{S}[\![e' \bowtie 0]\!]R$. As a consequence, in the abstract, when $e \sqsubseteq_{\gamma(R^\sharp)} e'$, we can safely abstract $\mathbb{S}[\![X \leftarrow e]\!]\gamma(R^\sharp)$ and $\mathbb{S}[\![e \bowtie 0]\!]\gamma(R^\sharp)$ as $\mathbb{S}^\sharp[\![X \leftarrow e']\!]R^\sharp$ and $\mathbb{S}^\sharp[\![e' \bowtie 0]\!]R^\sharp$ respectively. This is especially interesting when e' is easier to abstract than e in our abstract domain.

We now show how to abstract an arbitrary expression e into an affine form, which we call *linearization* and denote by $\text{lin}(e)$.

$$\begin{aligned}
&\mathrm{lin}(V) \stackrel{\text{def}}{=} V\\
&\mathrm{lin}([c_1,c_2]) \stackrel{\text{def}}{=} [c_1,c_2]\\
&\mathrm{lin}(-_\omega e) \stackrel{\text{def}}{=} \boxminus \mathrm{lin}(e)\\
&\mathrm{lin}(e_1 +_\omega e_2) \stackrel{\text{def}}{=} \mathrm{lin}(e_1) \boxplus \mathrm{lin}(e_2)\\
&\mathrm{lin}(e_1 -_\omega e_2) \stackrel{\text{def}}{=} \mathrm{lin}(e_1) \boxminus \mathrm{lin}(e_2)\\
&\mathrm{lin}(e_1 \times_\omega e_2) \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathrm{lin}(e_1) \boxtimes \mathrm{eval}(\mathrm{lin}(e_2))) \\ \mathrm{lin}(e_2) \boxtimes \mathrm{eval}(\mathrm{lin}(e_1))) \end{array} \right. \text{or}\\
&\mathrm{lin}(e_1/_\omega e_2) \stackrel{\text{def}}{=} \mathrm{lin}(e_1) \boxslash \mathrm{eval}(\mathrm{lin}(e_2))\\
&\text{where:}\\
&\mathrm{eval}([a_0,b_0] + \textstyle\sum_k [a_k,b_k]V_k) \stackrel{\text{def}}{=}\\
&\quad [a_0,b_0] +_i^\sharp \textstyle\sum_i^\sharp {}_{k=1}^{n} [a_k,b_k] \times_i^\sharp I^\sharp(V_k)
\end{aligned}$$

Figure 2.13: Expression linearization.

We suppose that we can extract from the abstract element $R^\sharp$ a map $I^\sharp \in \mathcal{V} \to \mathcal{I}$ associating to each variable its bounds. We then define $\mathrm{lin}(e)$ by induction on the syntax of the expression e as presented in Fig. 2.13. We have the following property: $e \sqsubseteq_{\gamma_i(I^\sharp)} \mathrm{lin}(e)$. Note that lin tries to keep expressions symbolic as much as possible, but resorts to abstracting affine forms into intervals (through eval) when non-linear constructions are encountered (such as divisions and multiplications). This abstraction is the reason why the transformation is parametrized by variable bounds $I^\sharp$ and it is sound only for abstract elements respecting these bounds.

Applications. We can feed an interval affine form directly to the interval domain, where it may provide an increase of precision thanks to the symbolic simplification performed by lin (e.g., $\mathrm{lin}(X -_\omega X) = 0$). To further simplify the affine form l output by lin, it is possible to ensure that the coefficients of all the variables are scalar, by distributing their contribution over the constant coefficient which is allowed to be an interval; we note the result $\mathrm{slin}(l)$:

$$\begin{aligned}
&\mathrm{slin}([a_0,b_0] + \textstyle\sum_k [a_k,b_k]V_k) \stackrel{\text{def}}{=}\\
&\textstyle\sum_k \frac{a_k+b_k}{2} V_k + \Big([a_0,b_0] +_i^\sharp \sum_i^\sharp {}_k [\frac{a_k-b_k}{2}, \frac{b_k-a_k}{2}] \times_i^\sharp I^\sharp(V_k)\Big).
\end{aligned} \tag{2.17}$$

The resulting interval affine form can be fed to the polyhedra domain, thus achieving a sound abstraction of assignments and tests for arbitrary expressions. We have chosen here to replace each interval with its midpoint, but other choices are equally possible (such as rounding to a set of predefined thresholds).

2.4.4 Floating-point numbers

Computers cannot manipulate real numbers, which are uncountable: they use finite approximations instead. A popular approximation is *floating-point numbers* (or floats) which can represent a wide range of values using a mantissa and an exponent of fixed bit-size. The large majority of programming languages and compilers now support the *IEEE 754 floating-point standard* [IEE85]; it is also widely supported natively in modern processors, making float computations very efficient. The algebra of floats differs significantly enough from the real one that an analysis assuming one semantics is not sound with respect to the other. A static analysis should support floats for two reasons: firstly, to analyze soundly programs manipulating floats and, secondly, to benefit from the efficiency of hardware float operations in order to improve the analysis time. The first aspect concerns the concrete semantics, while the second aspect concerns the implementation of abstract domains. They can be combined to construct a float analyzer for float programs.

We briefly recall some existing works on these two topics. We limit ourselves here to a real-based semantics of floats (a more precise semantics based on a bit-level representation of floats will be presented in Sec. 5.3). Moreover, we restrict the use of floats in implementations to intervals (float implementations of polyhedra will be considered in Sec. 4.1).

Floating-point arithmetic. Due to the limited precision of floats, not all reals can be represented as floats; we assimilate the set of floats $\mathbb{F}$ to a finite subset of reals $\mathbb{R}$. We denote as $R_{+\infty} \in \mathbb{R} \to \mathbb{F}$ the operation rounding a real up to a representable float, i.e.:

$$R_{+\infty}(x) \stackrel{\text{def}}{=} \min \{\, y \in \mathbb{F} \mid x \le y \,\} \ . \tag{2.18}$$

Likewise $R_{-\infty}$ denotes rounding down, so that $R_{-\infty}(x) \stackrel{\text{def}}{=} -R_{+\infty}(-x)$. Float implementations support alternate rounding modes, such as rounding to nearest or towards 0 but, for our purpose, it is sufficient to note that all implemented rounding functions R_r obey the relation:

$$\forall x :\ R_{-\infty}(x) \le R_r(x) \le R_{+\infty}(x) \ . \tag{2.19}$$

Arithmetic operators require some form of rounding as the real result is generally not representable in $\mathbb{F}$. We use circled operators $\oplus_r$, $\ominus_r$, $\otimes_r$, $\oslash_r$, tagged with a rounding direction r, to distinguish them from the operators on reals $+$, $-$, $\times$, $/$. Following the IEEE 754 standard, a floating-point operation with rounding mode r should be equivalent to computing the exact real result followed by rounding, i.e.:

$$\forall a,b,\in \mathbb{F} :\ \forall \cdot \in \{\, +,-,\times,/ \,\} :\ a \odot_r b \stackrel{\text{def}}{=} R_r(a \cdot b) \ . \tag{2.20}$$

Such operations are not always defined: / can result in a division by zero, and any operation can output a real result that cannot be rounded with R_r as it overflows $\mathbb{F}$. In this section, we consider such cases to be run-time errors.[7] To construct a concrete semantics of programs manipulating floating-point numbers, it is sufficient to replace $+$, $-$, $\times$, and / with $\oplus_r$, $\ominus_r$, $\otimes_r$, and $\oslash_r$ in $\mathbb{E}[\![\, e_1 \diamond_\omega e_2 \,]\!]$ in Fig. 2.2. As neither the negation nor the comparisons of floats incur any rounding, there is no need to change the semantics of $-_\omega e$ nor $e \bowtie 0$.

Floating-point intervals. It is straightforward to adapt the interval domain (Sec. 2.4.1) to use float bounds instead of real ones. As $\mathbb{F}$ is bounded (unlike $\mathbb{R}$), there is no need for infinite interval bounds, and so, we use bounds in $\mathbb{F}$ instead of $\mathbb{R} \cup \{+\infty, -\infty\}$. Thanks to (2.19), sound operators can be derived by always rounding upper bounds up and lower bounds down. For instance, the operator $+_i^\sharp$ in Fig. 2.11 is replaced with $\oplus_i^\sharp$ defined as:

$$[a,b] \oplus_i^\sharp [c,d] \stackrel{\text{def}}{=} [a \oplus_{-\infty} c,\ b \oplus_{+\infty} d] \tag{2.21}$$

and similarly for $\ominus_i^\sharp$, $\otimes_i^\sharp$, $\oslash_i^\sharp$. The result is an abstract static analysis implemented purely in floats, and that can soundly analyze programs manipulating floats.

[7] The IEEE 754 standard makes provision for special numbers, such as $+\infty$, $-\infty$, and NaN, which can be returned in these cases. Our simplified semantics assumes instead that the creation of a special number is an error, which is often what is intended by the programmer.

Floating-point linearization. Adapting the polyhedra domain (Sec. 2.4.2) from abstracting real-valued variables and expressions to abstracting float-valued ones is not much more complicated, as long as we keep representing polyhedra using arbitrary precision rationals and implementing the algorithms with exact arithmetic (replacing them with floats is much harder and will be discussed in Sec. 4.1). We proposed in [Min04a] to reuse the linearization and abstract the (highly non-linear) rounding function R_r as a non-deterministic choice in an error interval. For instance, assuming 32-bit single precision numbers, the rounding error can be bounded by $|R(x) - x| \leq \max(2^{-23}|x|, 2^{-159})$, denoting either a relative rounding error of magnitude 2^{-23}, or an absolute error of magnitude 2^{-159} (caused by computations on denormals, i.e., numbers close to zero). Given that the relative rounding error on an affine form l can be also expressed as an affine form $\varepsilon(l)$:

$$\begin{array}{l}\varepsilon([a_0, b_0] + \sum_k [a_k, b_k]V_k) \stackrel{\text{def}}{=} \\ \quad 2^{-23}(\max(|a_0|, |b_0|)[-1, 1] + \\ \qquad \sum_k \max(|a_k|, |b_k|)[-1, 1]V_k)\end{array} \tag{2.22}$$

the linearization algorithm from Fig. 2.13 can be easily modified to add rounding errors after each operation. For instance, we get:

$$\begin{array}{l}\mathrm{lin}(V) \stackrel{\text{def}}{=} V \\ \mathrm{lin}(\ominus_{r,\omega} e) \stackrel{\text{def}}{=} \boxminus \mathrm{lin}(e) \\ \mathrm{lin}(e_1 \oplus_{r,\omega} e_2) \stackrel{\text{def}}{=} \\ \quad \mathrm{lin}(e_1) \boxplus \mathrm{lin}(e_2) \boxplus \varepsilon(\mathrm{lin}(e_1) \boxplus \mathrm{lin}(e_2)) \boxplus [-2^{-159}, 2^{159}] \\ \mathrm{lin}(e_1 \otimes_{r,\omega} e_2) \stackrel{\text{def}}{=} \\ \quad \mathrm{eval}(\mathrm{lin}(e_1)) \boxtimes (\mathrm{lin}(e_2) \boxplus \varepsilon(\mathrm{lin}(e_2))) \boxplus [-2^{-159}, 2^{159}] \ .\end{array}$$

When evaluated with a real semantics, the affine form returned by $\mathrm{lin}(e)$ safely over-approximates the set of values computed by $\mathbb{E}[\![e]\!]$ with a float semantics. Hence, $\mathrm{lin}(e)$ or $\mathrm{slin}(\mathrm{lin}(e))$ can be directly fed to a domain, such as polyhedra, that reasons on reals.

2.5 Conclusion

In this chapter, we have recalled well-known notions of abstract interpretation and used them to construct a classic static analysis parametrized by a numeric abstract domain, of which we gave two examples. Our work, detailed in the following chapters, extends these results in several directions. More precisely, we will discuss: a generic extension to concurrency (Chap. 3), a specific extension to data-structures in the C language (Chap. 5), extended polyhedral domains (Chap. 4), and actual implementations and experimental results (Chap. 6).

Chapter 3

Analysis of concurrent programs

My main current research topic is the analysis of concurrent programs by abstract interpretation. This chapter presents the theoretical foundations of a practical analysis of concurrent programs to infer invariants and report soundly all run-time errors. As concurrent programming models are diverse and exhibit widely different semantics, we narrow our focus to a simple model using a fixed set of threads that communicate implicitly in a shared memory. This model is far from the only one, but it is realistic and fits very well an important application domain for program verification: embedded critical software. As in Sec. 2.3, we work on a simple artificial language in order to present our construction fully formally and focus on concurrency only, being understood that the method can be applied in real-life contexts (as reported in Sec. 6.3).

Our threaded language is presented in Sec. 3.1. It is extended in Sec. 3.4 to support explicit synchronization mechanisms: mutual exclusion locks, as well as priority-based real-time scheduling which is pervasive in the realm of embedded software.

A first attempt at providing a semantics and deriving a static analysis for our language is also described in Sec. 3.1. It follows the same principles as in Chap. 2, using transition systems, trace semantics, and state abstractions. We show, however, that this straightforward adaptation of sequential semantics to concurrent semantics comes at great cost in efficiency. Although we can abstract environments, we suffer, on the control state, from the combinatorial state explosion problem that plagues enumeration methods (such as explicit-state model checking). Thus, we propose an abstraction that limits the control state by reducing the analysis of a concurrent program to the analysis of its individual threads, complemented with a notion of interferences that model thread interactions. We take our inspiration from a thread-modular proof method, so-called rely-guarantee, introduced by Jones [Jon81]. We first formalize rely-guarantee as abstract interpretation, in Sec. 3.2, before applying abstractions to construct a static analysis in big-step form, in Sec. 3.3. As the sequential static analysis from Sec. 2.3, this analysis is parametrized by a choice of numeric abstract domains. In fact, we show that the concurrent analysis can be constructed from a sequential one with only minor changes.

As last extension, in Sec. 3.5, we study the effect of weakly consistent memory models on the soundness of our analysis. Such models are now accepted as realistic semantics of modern processors and are being included in the specification of most programming languages.

The work presented in this chapter has been published in more details in [BCC+10a, Min11, Min12d, Min12c]. In particular, we refer the reader to [Min12d, Min12c] for the proof of the theorems (which are omitted here). The analysis described here has been implemented within the AstréeA analyzer. We delay the discussion of the implementation and the experimental results to Chap. 6.

3.1 Concurrent language

We consider here a simple multi-thread extension of our language from Sec. 2.3.1. We presents its syntax, its semantics, and its abstraction into a static analyzer.

3.1.1 Syntax

The syntax of Fig. 2.1 is modified in a single but meaningful way: instead of being composed of a single statement, programs are now parallel compositions of several statements, called *threads*:

$$prog ::= {}^{\ell e_1}\, stat_1\, {}^{\ell x_1} \;||\; \cdots \;||\; {}^{\ell e_N}\, stat_N\, {}^{\ell x_N} \quad . \tag{3.1}$$

We identify threads by numbers and denote the set of all identifiers as $\mathcal{T} \stackrel{\text{def}}{=} \{\, 1, \ldots, N \,\}$. Note that the parallel operator $||$ can only appear at the top level, not inside statements, preventing the dynamic creation of threads. We assume here a fixed finite number N of threads, but Sec. 3.2.4 shows that some of our results can be adapted to the case where threads have an unbounded number of concurrent instances. Furthermore, we do not consider any concurrency-specific statement to control the scheduling or implement synchronization yet; they will be added in Sec. 3.4.

3.1.2 Semantics

Our goal is to design a static analysis for our concurrent language. We start by applying the method from Sec. 2.3 which was successful on sequential programs: we define a small-step concrete semantics as a transition system, derive concrete trace and state semantics making program properties apparent, and finally abstract them into an approximated but computable semantics.

We will see, however, that constructing a static analysis by abstracting the state semantics does not give acceptable performances. We will solve this problem in Sec. 3.2 by returning to the trace semantics, and abstracting it in another way.

Transition system

The transition system $(\Sigma, \mathcal{A}, I, \tau)$ modeling our program is defined as follows:

– As state space, we use $\Sigma \stackrel{\text{def}}{=} ((\mathcal{T} \to \mathcal{L}) \times \mathcal{E}) \cup \Omega$. The program can be in a state $\langle \ell, \rho \rangle \in (\mathcal{T} \to \mathcal{L}) \times \mathcal{E}$, where each thread $t \in \mathcal{T}$ has its own control location $\ell(t) \in \mathcal{L}$ and the environment $\rho \in \mathcal{E}$ is shared. Alternatively, it can be in an error state $\omega \in \Omega$.

– All the threads start at their first location and all the variables are initialized to 0: $I \stackrel{\text{def}}{=} \{ \langle \lambda t.\, \ell e_t,\, \lambda V.\, 0 \rangle \}$.

– The actions are the thread identifiers: $\mathcal{A} \stackrel{\text{def}}{=} \mathcal{T}$.

– The transition relation τ of the whole program is derived from the transition relation of its individual threads, each thread being seen as a sequential process. Given, for each thread $t \in \mathcal{T}$, the relation $\tau \left[{}^{\ell e_t}\, stat_t\, {}^{\ell x_t} \right]$ defined by Fig. 2.3, we define τ as:

$$\begin{array}{l} \langle \ell, \rho \rangle \xrightarrow{t}_{\tau} \langle \ell', \rho' \rangle \iff \\ \quad \langle \ell(t), \rho \rangle \to_{\tau \left[{}^{\ell e_t}\, stat_t\, {}^{\ell x_t} \right]} \langle \ell'(t), \rho' \rangle \wedge \\ \quad \forall t' \neq t : \ell(t') = \ell'(t') \ . \end{array} \tag{3.2}$$

This states that an execution step of the program is an execution step of any single thread t, which updates its control state $\ell(t)$ and the global memory ρ according to the sequential semantics, but leaves the control location of other threads $\ell(t')$, $t' \neq t$ intact.

We note that the transition system of the program is much larger than that of its individual threads: the control part of Σ has a larger size ($\mathcal{T} \to \mathcal{L}$ instead of $\mathcal{L}$), while any single transition in a thread yields $|\mathcal{L}|^{N-1}$ transitions in τ as it is duplicated for each possible control state of the other threads.

3.1.3 Trace and state semantics

The benefit of modeling concurrent programs as transition systems is that the trace and state semantics from Sec. 2.3.3 can be directly applied.

Maximal trace semantics. Recall that the maximal trace semantics $\mathcal{M}$ from (2.3) is the set of maximal finite or infinite traces in $\mathcal{T}r^{\infty}(\Sigma, \mathcal{A})$ starting in an initial state and obeying τ. It models effectively a program execution as the arbitrary interleaving of thread executions, where each assignment $X \leftarrow e$ or test $e \bowtie 0$ denotes an atomic operation. In particular, a thread cannot be preempted during the evaluation of an expression (this limitation will be addressed in Sec. 3.5). This simple and natural model of concurrent executions corresponds to sequential consistency, as coined by Lamport [Lam79] (Sec. 3.5 will also consider more complex execution models). Additionally, the trace semantics keeps track of which thread performs any given transition, using actions.

As stated before on sequential programs, the maximal trace semantics is very expressive: $\mathcal{M}$ captures much program information, and many properties can be expressed as a set of maximal traces P and simply checked by testing whether $\mathcal{M} \subseteq P$. Examples include: termination ($\mathcal{M} \subseteq \mathcal{T}r^{*}(\Sigma, \mathcal{A})$) and invariance ($\mathcal{M} \subseteq \mathcal{T}r^{\infty}(S, \mathcal{A})$ for some $S \subseteq \Sigma$). On concurrent programs, it additionally allows proving properties under *fairness conditions*. Fairness [Fra86] ensures that no thread is denied to run; it is a property enforced by many schedulers. Formally, we define the notion of a thread t enabled in a state σ as its ability to make a transition:

$$enbl_{\tau}(\sigma, t) \stackrel{\text{def}}{=} \exists \sigma' \in \Sigma : \sigma \xrightarrow{t}_{\tau} \sigma' \ . \tag{3.3}$$

An infinite trace $\sigma_0 \xrightarrow{a_0} \sigma_1 \xrightarrow{a_1} \sigma_2 \cdots$ is weakly fair for τ if no thread can be continuously enabled without running infinitely often:

$$\begin{array}{l} \forall t \in \mathcal{T} : (\exists i : \forall j \geq i : enbl_{\tau}(\sigma_j, t)) \implies \\ \qquad (\forall i : \exists j \geq i : a_j = t) \end{array}$$

while it is strongly fair if no thread can be infinitely often enabled (possibly intermittently) without running infinitely often:

$$\begin{array}{l} \forall t \in \mathcal{T} : (\forall i : \exists j \geq i : enbl_{\tau}(\sigma_j, t)) \implies \\ \qquad (\forall i : \exists j \geq i : a_j = t) \ . \end{array}$$

Given the set $\mathcal{F}air$ of (weakly or strongly) fair infinite traces for τ and of finite traces, a proof of a property P under fairness reduces to checking that $\mathcal{M} \cap \mathcal{F}air \subseteq P$.

Example 3.1.1. Consider the program :

$$prog \stackrel{\text{def}}{=} \textbf{while } X \geq 0 \textbf{ do } X \leftarrow X + 1 \textbf{ done } || \ X \leftarrow -1 \ .$$

Then, $\mathcal{M}$ contains an infinite trace where the second thread never runs. Thus, without any fairness condition, $\mathcal{M}$ does not always terminate: $\mathcal{M} \not\subseteq \mathcal{T}r^{*}(\Sigma, \mathcal{A})$. However, unless it makes a transition, the second thread is always enabled; hence this infinite trace is neither strongly nor weakly fair. Moreover, it is sufficient that the second thread makes a transition for the program to terminate. Thus, under both fairness conditions, *prog* always terminates: $\mathcal{M} \cap \mathcal{F}air \subseteq \mathcal{T}r^{*}(\Sigma, \mathcal{A})$.
End of example.

Partial trace semantics. Recall that the partial trace semantics $\mathcal{F}$ (2.4) is the set of finite prefixes of the maximal trace semantics $\mathcal{M}$. It is an abstraction $\mathcal{F} = \alpha_{pref}(\mathcal{M})$ (2.5) of the maximal trace semantics. It is more convenient to compute as it dispenses with infinite traces and it admits a characterization as the least fixpoint of a forward operator F (2.6), but it looses some information.

We showed in Ex. 2.3.2 that the abstraction α_{pref} makes it impossible to prove the termination of programs with computations of finite but unbounded length. On concurrent programs, it also makes proofs under fairness conditions impossible.

Example 3.1.2. Consider again the program from Ex. 3.1.1. We have $\alpha_{pref}(\mathcal{M} \cap \mathcal{F}air) = \alpha_{pref}(\mathcal{M})$. Indeed, $\mathcal{M} \cap \mathcal{F}air$ removes a single infinite trace from $\mathcal{M}$, the one where the second thread never runs, but it does not remove any strict partial trace: the prefix of length n of the infinite trace is also the prefix of a finite trace in $\mathcal{M}$ (for instance a prefix of the trace where the second thread runs only at the $n+1$-th step).
End of example.

Reachable state semantics. As for sequential programs, we are interested in inferring reachability to prove invariant properties. Liveness properties [LS85] (such as termination) and proofs under fairness assumptions [Fra86] are thus out of the scope of our work. We would like to consider them in future work and, for now, we refer the reader to Radhia Cousot's work for a discussion on these topics [Cou85]. As we focus on invariance, it is natural to further abstract the partial traces to only compute the reachable state semantics $\mathcal{R} = \alpha_{reach}(\mathcal{F})$ (2.7), which forgets the ordering of states. It can be expressed as a fixpoint $\mathcal{R} = \mathrm{lfp}\, R$ where the definition of R, introduced in (2.8), is recalled below:

$$R \stackrel{\text{def}}{=} \lambda S.\, I \cup \{ \sigma \mid \exists a \in \mathcal{A}, \sigma' \in S : \sigma' \xrightarrow{a}_{\tau} \sigma \} \ .$$

On concurrent programs, this abstraction also forgets about actions, and so, about thread identities. This suggests that we can remove actions from the original transition system to get a semantics which is more similar to that of a sequential program, and derive equation-based or big-step semantics that are convenient to turn into static analyzers. This is what we attempt in the following two sections.

3.1.4 Equational semantics

Following Sec. 2.3.4, we construct an equational version of the concrete state semantics by partitioning states by control locations ℓ (ignoring error states): we assign a variable $\mathcal{X}_\ell$ with value in the powerset of memory states $\mathcal{P}(\mathcal{E})$ to each location ℓ, and generate equations by induction on the syntax of programs. However, a control location ℓ is no longer a single, global syntactic point, but rather a per-thread syntactic point: $\ell \in \mathcal{T} \to \mathcal{L}$. To derive the equations, we consider the equations of each thread $eq\left[{}^{\ell e_t}\ stat_t\ {}^{\ell x_t}\right]$ seen as a sequential process, as defined Fig. 2.4, and join them in a way similar to the way τ is derived from $\tau\left[{}^{\ell e_t}\ stat_t\ {}^{\ell x_t}\right]$ in (3.2). More precisely, the set eq of equations is:

$$\begin{array}{l} eq \stackrel{\text{def}}{=} \{\ \mathcal{X}_{\ell_0} = \bigcup_{t\in\mathcal{T}} \{\ F(\mathcal{X}_{\ell_1},\ldots,\mathcal{X}_{\ell_n}) \mid \\ \qquad \exists(\mathcal{X}_{\ell'} = F(\mathcal{X}_{\ell'_1},\ldots,\mathcal{X}_{\ell'_n})) \in eq\left[{}^{\ell e_t}\ stat_t\ {}^{\ell x_t}\right] : \\ \qquad \forall i \leq n : \ell_i(t) = \ell'_i \wedge \forall t' \neq t : \ell_i(t') = \ell_0(t')\ \} \mid \\ \quad \ell_0 \in \mathcal{T} \to \mathcal{L}\ \}\ . \end{array} \tag{3.4}$$

Intuitively, for each concurrent control location $\ell_0 \in \mathcal{T} \to \mathcal{L}$, we are joining equations from any thread t where the left-hand side $\mathcal{X}_{\ell'}$ matches $\ell_0(t)$.

Given the concrete equation system, an effective analysis can be constructed by replacing computations on the concrete environment domain $\mathcal{P}(\mathcal{E})$ with computations on an abstract one $\mathcal{E}^\sharp$ and inserting widening points where needed, in a manner similar to Sec. 2.3.6. However, a major issue is that the obtained system is much larger than that of its individual threads. There are $|\mathcal{L}|^N$ variables instead of $|\mathcal{L}|$, and a single equation from a thread is repeated $|\mathcal{L}|^{N-1}$ times in the system. This blow-up is illustrated by the following example:

Example 3.1.3. Figure 3.1.(a) presents the parallel composition of two simple threads: t_1 increments X until it reaches Y, while t_2 concurrently increments Y until it reaches 10. Figure 3.1.(b) presents the equation system derived from Fig. 2.4 and (3.4). We do not expect the reader to read this system, but simply to appreciate its size and complexity, compared to the simplicity of the program (in fact, we even simplified the system for the sake of presentation by omitting some control locations and factoring some variables).
End of example.

Although this method is used in academic demonstration tools (such as Jeannet's ConcurInterproc [Jea11], an extension of the Interproc academic analyzer [LAJ11] to concurrent programs), it cannot scale up to realistic programs.

3.1.5 Big-step semantics

In order to stop the proliferation of equation variables, we turn towards big-step semantics, which make a parsimonious use of abstract elements (Sec. 2.3.5). The main issue to solve is the lack of syntactic structure to iterate on: there is no convenient inductive definition of the interleaving of threads as these interleavings are combinatorial in nature.

In [Mon07], Monniaux proposes a solution adapted to two threads, for the specific problem of analyzing a USB device driver running concurrently with an intelligent controller (modelled as a C program). The principle is to perform a big-step abstract interpretation by induction on the syntax of the device driver but, at each instruction, run an abstract interpretation of the complete model of the controller. The soundness of the approach relies on the fact that the controller is modeled as a non-deterministic choice of atomic actions in an infinite loop. Despite some optimizations (such as only running the abstract controller when the device reads or modifies a shared variable), the analysis for two threads is already quadratic in the size of the program as each instruction from the second thread is reanalyzed when analyzing (almost) each instruction of the first one. Thus, the soundness conditions and the scalability of this method are not adapted to more complex and general concurrent programs.

Summary. Following our failed attempts, we are now ready to state the desirable properties that a concurrent program analysis should possess:

1. keep information attached to syntactic program locations in $\mathcal{L}$ (and not to control states in $\mathcal{T} \to \mathcal{L}$);
2. avoid re-analyzing each thread instruction for each configuration of the other threads;
3. be defined by induction on the syntax of threads (big-step semantics);
4. abstract control-flow information (with controllable cost versus precision trade-off, if possible);
5. reuse existing abstractions and abstract domains.

The equation-based analysis presented above only achieves 5, while [Mon07] additionally achieves 1 and 3 in limited circumstances. Ideally, we would like the analysis of a concurrent program to be reduced to the independent analysis of each thread. Completely ignoring thread interactions is of course not sound. Nevertheless, the following sections show that we can construct a sound thread-modular analysis that almost reduces to independent thread analyses.

3.2 Rely-guarantee reasoning as abstract interpretation

In order to reach our goal, we take a detour through program proof methods, which already feature *thread-modular* methods. It will then only be a matter of formalizing them as abstract interpretation, expressing them in constructive (fixpoint) form, and applying abstractions.

3.2.1 Proof methods

Proof methods for sequential programs date back to the work of Floyd and Hoare [Flo67, Hoa69]. They consist in annotating program statements *stat* with preconditions $\{P\}$ and postconditions $\{Q\}$, that are logical assertions on the state of the program. A triple $\{P\}\ stat\ \{Q\}$ means that, if *stat* is executed in a program state satisfying $\{P\}$, then the output state satisfies $\{Q\}$. Hoare [Hoa69] proposed a set of axioms and rules that can be used to derive valid triples. These rules (which are well known and not repeated here) deduce a triple on a statement based on triples on its components, so that a proof tree naturally follows the syntactic structure of the program.

t_1	t_2
while $^{\ell 1}0 = 0$ **do**	**while** $^{\ell 4}0 = 0$ **do**
$\quad ^{\ell 2}$**if** $X < Y$ **then**	$\quad ^{\ell 5}$**if** $Y < 10$ **then**
$\quad\quad ^{\ell 3}X \leftarrow X + 1$	$\quad\quad ^{\ell 6}Y \leftarrow Y + 1$
$\quad$ **endif**	$\quad$ **endif**

(a)

$$\mathcal{X}_{\ell 1,\ell 4} = I \cup \mathbb{S}_{\mathcal{E}}[\![X \leftarrow X+1]\!]\mathcal{X}_{\ell 3,\ell 4} \cup \mathbb{S}_{\mathcal{E}}[\![X \geq Y]\!]\mathcal{X}_{\ell 2,\ell 4} \cup \mathbb{S}_{\mathcal{E}}[\![Y \leftarrow Y+1]\!]\mathcal{X}_{\ell 1,\ell 6} \cup \mathbb{S}_{\mathcal{E}}[\![Y \geq 10]\!]\mathcal{X}_{\ell 1,\ell 5}$$
$$\mathcal{X}_{\ell 2,\ell 4} = \mathcal{X}_{\ell 1,\ell 4} \cup \mathbb{S}_{\mathcal{E}}[\![Y \leftarrow Y+1]\!]\mathcal{X}_{\ell 2,\ell 6} \cup \mathbb{S}_{\mathcal{E}}[\![Y \geq 10]\!]\mathcal{X}_{\ell 2,\ell 5}$$
$$\mathcal{X}_{\ell 3,\ell 4} = \mathbb{S}_{\mathcal{E}}[\![X < Y]\!]\mathcal{X}_{\ell 2,\ell 4} \cup \mathbb{S}_{\mathcal{E}}[\![Y \leftarrow Y+1]\!]\mathcal{X}_{\ell 3,\ell 6} \cup \mathbb{S}_{\mathcal{E}}[\![Y \geq 10]\!]\mathcal{X}_{\ell 3,\ell 5}$$
$$\mathcal{X}_{\ell 1,\ell 5} = \mathbb{S}_{\mathcal{E}}[\![X \leftarrow X+1]\!]\mathcal{X}_{\ell 3,\ell 5} \cup \mathbb{S}_{\mathcal{E}}[\![X \geq Y]\!]\mathcal{X}_{\ell 2,\ell 5} \cup \mathcal{X}_{\ell 1,\ell 4}$$
$$\mathcal{X}_{\ell 2,\ell 5} = \mathcal{X}_{\ell 1,\ell 5} \cup \mathcal{X}_{\ell 2,\ell 4}$$
$$\mathcal{X}_{\ell 3,\ell 5} = \mathbb{S}_{\mathcal{E}}[\![X < Y]\!]\mathcal{X}_{\ell 2,\ell 5} \cup \mathcal{X}_{\ell 3,\ell 4}$$
$$\mathcal{X}_{\ell 1,\ell 6} = \mathbb{S}_{\mathcal{E}}[\![X \leftarrow X+1]\!]\mathcal{X}_{\ell 3,\ell 6} \cup \mathbb{S}_{\mathcal{E}}[\![X \geq Y]\!]\mathcal{X}_{\ell 2,\ell 6} \cup \mathbb{S}_{\mathcal{E}}[\![Y < 10]\!]\mathcal{X}_{\ell 1,\ell 5}$$
$$\mathcal{X}_{\ell 2,\ell 6} = \mathcal{X}_{\ell 1,\ell 6} \cup \mathbb{S}_{\mathcal{E}}[\![Y < 10]\!]\mathcal{X}_{\ell 2,\ell 5}$$
$$\mathcal{X}_{\ell 3,\ell 6} = \mathbb{S}_{\mathcal{E}}[\![X < Y]\!]\mathcal{X}_{\ell 2,\ell 6} \cup \mathbb{S}_{\mathcal{E}}[\![Y < 10]\!]\mathcal{X}_{\ell 3,\ell 5}$$

(b)

Figure 3.1: A concurrent program example (a) and its (simplified) equational semantics (b).

Owickie–Gries–Lamport. On concurrent programs, proof methods were pioneered by Owicki, Gries, and Lamport [OG76, Lam77, Lam80]. The core idea is to add to Hoare's rules a new rule for the parallel operator:

$$\frac{\{P_1\}\, s_1\, \{Q_1\} \quad \{\, P_2\,\}\, s_2\, \{Q_2\}}{\{P_1 \wedge P_2\}\, s_1 \mid\mid s_2\, \{Q_1 \wedge Q_2\}}\ .$$

This realizes two of our goals: assertions are attached to program locations and the proof reflects the structure of the program. However, this rule has an important restriction: it can only be applied if the proofs of $\{P_1\}\, s_1\, \{Q_1\}$ and $\{P_2\}\, s_2\, \{Q_2\}$ do not interfere. Checking interference freedom requires proving that each assertion appearing in one of the proofs is invariant by the statements of the other threads: for instance, if the assertion Φ appears in the first proof tree, and the triple $\{P_2'\}\, s_2'\, \{Q_2'\}$ appears in the second one, we must additionally prove that $\{\Phi \wedge P_2'\}\, s_2'\, \{\Phi\}$. Hence, the proof checking is not thread-modular. We cannot hope to design a modular inference scheme on such bases.

Rely-Guarantee. Jones introduced *rely-guarantee* methods in [Jon81] as a way to address the modularity issues of Owicki–Gries–Lamport proof methods. In Jones' method, thread interferences are explicitly provided as part of the annotation, instead of being checked implicitly in the proof checking. Hoare triples $\{P\}\ stat\ \{Q\}$ are replaced with quintuples:

$$R, G \vdash \{P\}\ stat\ \{Q\} \tag{3.5}$$

where P and Q are, as before, assertions on program states, while R (Rely) and G (Guarantee) are assertions on program transitions. Quintuples have the following intuitive semantics: if P holds before *stat* is executed and the effect of all other threads is included in R, then Q is true after *stat* has been executed and its effect is included in G. The rule for parallel composition then becomes, without side-condition:

$$\frac{R \cup G_2, G_1 \vdash \{P_1\}\, s_1\, \{Q_1\} \qquad R \cup G_1, G_2 \vdash \{P_2\}\, s_2\, \{Q_2\}}{R, G_1 \cup G_2 \vdash \{P_1 \wedge P_2\}\, s_1 \mid\mid s_2\, \{Q_1 \wedge Q_2\}}\ .$$

checking t_1 :	
t_1	$R_1 = G_2$
while $^{\ell 1}0 = 0$ **do**	X is unchanged
$\quad ^{\ell 2}$**if** $X < Y$ **then**	Y is incremented
$\quad\quad ^{\ell 3}X \leftarrow X + 1$	$Y \leq 10$
$\quad$ **endif**	

checking t_2 :	
$R_2 = G_1$	t_2
Y is unchanged	**while** $^{\ell 4}0 = 0$ **do**
	$\quad ^{\ell 5}$**if** $Y < 10$ **then**
	$\quad\quad ^{\ell 6}Y \leftarrow Y + 1$
	$\quad$ **endif**

$\ell 1 : 0 \leq X \leq Y \leq 10$	$\ell 4 : 0 \leq X \leq Y \leq 10$
$\ell 2 : 0 \leq X \leq Y \leq 10$	$\ell 5 : 0 \leq X \leq Y \leq 10$
$\ell 3 : 0 \leq X \leq Y - 1 \leq 10$	$\ell 6 : 0 \leq X \leq Y \leq 9$

Figure 3.2: Rely-guarantee proof sketch for the program in Fig. 3.1.(a).

Example 3.2.1. Figure 3.2 presents a rely-guarantee reasoning on the program example from Fig. 3.1.(a). The presentation is simplified: although we show the invariant at each control location $\ell 1, \ldots, \ell 6$, each thread is checked in turn with respect to a global assertion on the transitions of the other thread. Each thread guarantees exactly what the other one relies on: $R_1 = G_2$ and $R_2 = G_1$.

End of example.

The rely-guarantee method is indeed modular: each thread can be checked without looking at the syntax of the other threads, but only at the rely assertions. Intuitively, by modeling the effects a thread has on others threads, rely and guarantee assertions form an abstraction of the semantics of threads.

3.2.2 Interference semantics

As mentioned in Sec. 2.3.4, there exists a connection between the reachability semantics $\mathcal{R}$ and Hoare's proof method for sequential programs. This connection was exposed by Cousot and Cousot in [CC77]. In [CC84], they showed a similar connection for the proof methods of concurrent programs of Owicki, Gries, and Lamport. One of our contributions [Min12c] is a similar connection for Jones' rely-guarantee method. We develop formally this result in this section.

We start from the partial trace semantics $\mathcal{F}$, which we decompose into two complementary abstractions: thread-local reachable states and inter-thread interferences.

Local states. For each thread $t \in \mathcal{T}$, we define its reachable *local states* $\mathcal{Rl}(t)$ as the state abstraction $\mathcal{R}$ where the state control part is reduced to that of t only. The control state of the other threads $t' \neq t$ is stored in *auxiliary variables*, called $pc_{t'}$ (as our language only features real-valued variables, we assume, for the sake of simplicity, that the syntactic locations $\mathcal{L}$ are real numbers). For each thread $t \in \mathcal{T}$, the set Σ_t of local states is:

$$\begin{array}{l} \Sigma_t \stackrel{\text{def}}{=} (\mathcal{L} \times \mathcal{E}_t) \cup \Omega, \text{ where} \\ \mathcal{E}_t \stackrel{\text{def}}{=} \mathcal{V}_t \to \mathbb{R} \\ \mathcal{V}_t \stackrel{\text{def}}{=} \mathcal{V} \cup \{\, pc_{t'} \mid t' \in \mathcal{T}, t' \neq t \,\} \end{array} \tag{3.6}$$

and we define the reachable local states $\mathcal{Rl} \in \Pi t{:}\mathcal{T}.\Sigma_t$ as:

$$\begin{array}{l} \mathcal{Rl}(t) \stackrel{\text{def}}{=} \pi_t(\mathcal{R}) \\ \text{where } \pi_t(\langle \ell, \rho \rangle) \stackrel{\text{def}}{=} \langle \ell(t), \rho\,[\forall t' \neq t : pc_{t'} \mapsto \ell(t')] \rangle \\ \text{extended naturally from } \Sigma \to \Sigma_t \text{ to } \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma_t) \ . \end{array} \tag{3.7}$$

Thanks to the auxiliary variables, the projection π_t to the local state space is a bijection and no information is lost (see also Ex. 3.2.2 below on the importance of auxiliary variables).

Interferences. For each thread $t \in \mathcal{T}$, the *interferences* it causes $\mathcal{I}(t) \in \mathcal{P}(\Sigma \times \Sigma)$ is the set of transitions produced by t in the partial trace semantics $\mathcal{F}$:

$$\begin{array}{l} \mathcal{I}(t) \stackrel{\text{def}}{=} \alpha_{itf}(\mathcal{F}), \text{ where} \\ \alpha_{itf}(X) \stackrel{\text{def}}{=} \lambda X.\, \{\, \langle \sigma_i, \sigma_{i+1} \rangle \mid \\ \qquad\qquad \exists \sigma_0 \stackrel{a_0}{\to} \cdots \stackrel{a_{n-1}}{\to} \sigma_n \in X : a_i = t \,\} \ . \end{array} \tag{3.8}$$

Hence, it is a subset of the transition relation τ of the program reduced to transitions that appear in actual executions.

Fixpoint characterization. We now propose a characterization of $\mathcal{Rl}$ and $\mathcal{I}$ directly in terms of fixpoints of operators on the transition system, which do not require to compute $\mathcal{R}$ nor $\mathcal{F}$. We first express $\mathcal{Rl}$ in fixpoint form as a function of $\mathcal{I}$:

$$\begin{array}{l} \mathcal{Rl}(t) = \mathrm{lfp}\, R_t(\mathcal{I}), \text{ where} \\ R_t \in (\mathcal{T} \to \mathcal{P}(\Sigma \times \Sigma)) \to \mathcal{P}(\Sigma_t) \to \mathcal{P}(\Sigma_t) \\ R_t \stackrel{\text{def}}{=} \lambda Y.\, \lambda X. \\ \quad \pi_t(I) \cup \{\, \pi_t(\sigma') \mid \exists \pi_t(\sigma) \in X : \\ \qquad\qquad \sigma \stackrel{t}{\to}_\tau \sigma' \vee \exists t' \neq t : \langle \sigma, \sigma' \rangle \in Y(t') \,\} \ . \end{array} \tag{3.9}$$

The function $R_t(Y)$ is similar to R (2.8) used to compute classic reachability $\mathcal{R}$, but it explores the reachable states by interspersing two kinds of steps: firstly, steps from the transition relation of the thread t and, secondly, interference steps from other threads, which are provided in the argument Y.

We now express $\mathcal{I}$ as a function of $\mathcal{Rl}$:

$$\begin{array}{l} \mathcal{I}(t) = B(\mathcal{Rl})(t), \text{ where} \\ B \in \Pi t{:}\mathcal{T}.\mathcal{P}(\Sigma_t) \to \mathcal{T} \to \mathcal{P}(\Sigma \times \Sigma) \\ B \stackrel{\text{def}}{=} \lambda X.\, \lambda t.\, \{\, \langle \sigma, \sigma' \rangle \mid \\ \qquad\qquad \pi_t(\sigma) \in X(t) \wedge \sigma \stackrel{t}{\to}_\tau \sigma' \,\} \ . \end{array} \tag{3.10}$$

$B(X)(t)$ simply collects all the transitions in the transition relation of the thread t starting from a local state in $X(t)$.

There is a mutual dependency between equations (3.9) and (3.10), which we solve using a fixpoint. The following theorem, which characterizes reachable local states $\mathcal{Rl}$ in a nested fixpoint form, is proved in [Min12c]:

Theorem 3.2.1.
$\mathcal{Rl} = \mathrm{lfp}\, H$, where
$H \in (\Pi t{:}\mathcal{T}.\mathcal{P}(\Sigma_t)) \to (\Pi t{:}\mathcal{T}.\mathcal{P}(\Sigma_t))$
$H \stackrel{\text{def}}{=} \lambda X.\, \lambda t.\, \mathrm{lfp}\, R_t(B(X))$.

Compared to a rely-guarantee proof $R, G \vdash \{P\}\ stat_t\ \{Q\}$, the reachable local states $\mathcal{Rl}(t)$ correspond to state assertions P and Q, while the interferences $\mathcal{I}(t)$ correspond to rely and guarantee assertions R and G. Proving that a given quintuple is valid amounts to checking that $\forall t \in \mathcal{T} : \mathcal{Rl}(t) \subseteq R_t(\mathcal{I})(\mathcal{Rl}(t))$ and $B(\mathcal{Rl})(t) \subseteq \mathcal{I}(t)$. Our fixpoints are, however, constructive and can infer the optimal assertions instead of simply checking user-provided assertions. Computing $\mathrm{lfp}\, R_t(\mathcal{I})$ corresponds to inferring state assertions P and Q given the interferences, while computing $\mathrm{lfp}\, H$ infers both interferences and state assertions. Thread-modularity is achieved as each function R_t only explores the transition relation generated by the thread t in isolation. It relies on its first argument to know the transitions of the other threads without having to explore them.

Fixpoints can be computed by iteration. Indeed, we apply Thm. 2.2.2 by Cousot and Cousot. In particular, $\mathrm{lfp}\, H = \bigsqcup_{n \in \mathbb{N}} H^n(\lambda t.\, \emptyset)$. As noted above, R_t is similar to R, and computing $\mathrm{lfp}\, R_t(Y)$ used in the definition of H is similar to a classic reachability computation. Hence, the computation of $\mathcal{Rl} = \mathrm{lfp}\, H$ in Thm. 3.2.1 can be understood as an iterative computation that re-analyzes all the threads until the interferences stabilize. The analysis of a single thread is a sequential program analysis, slightly modified to incorporate the effect of interferences.

3.2.3 Abstraction

Although it enjoys the required thread-modularity, the nested fixpoint characterization of $\mathcal{Rl}$ in Thm. 3.2.1 is still very concrete. Indeed, the uncomputable state semantics $\mathcal{R}$ can be recovered from it. To construct an effective static analysis, we need further abstractions. We will abstract the local states and the interferences independently from each other.

Local state abstraction. A set of local states for a thread t lives in $\mathcal{P}(\Sigma_t) = \mathcal{P}((\mathcal{L} \times \mathcal{E}_t) \cup \Omega) \simeq (\mathcal{L} \to \mathcal{P}(\mathcal{E}_t)) \times \mathcal{P}(\Omega)$. Hence, it can be abstracted by associating to each control location in $\mathcal{L}$ a value in some numeric abstract domain $\mathcal{E}^\sharp$, and maintaining a set of errors in extension. However, the number of variables is often a critical parameter in the efficiency of a domain (especially for precise relational domains, such as polyhedra). A faster analysis can be constructed by first removing

the auxiliary variables from $\mathcal{V}_t$ with the following abstraction:

$$\begin{array}{l} \alpha_{aux} \in \mathcal{P}(\mathcal{V}_t \to \mathbb{R}) \to \mathcal{P}(\mathcal{V} \to \mathbb{R}) \\ \alpha_{aux} \stackrel{\text{def}}{=} \lambda R.\, \{\, \rho_{|_{\mathcal{V}}} \mid \rho \in R \,\} \end{array} \tag{3.11}$$

before using a numeric abstract domain. The resulting analysis is still flow-sensitive in that a distinct local invariant is associated to each control location $\ell \in \mathcal{L}$ of the thread. Removing auxiliary variables still allows us to infer a large class of properties (for instance, the example of Fig. 3.2 has been precisely handled without the need for auxiliary variables). However, they are necessary in some cases. In fact, early proof methods did not feature them, and they were introduced subsequently in order to achieve completeness [OG76].

Example 3.2.2. Consider the program $prog = {}^{\ell 1}X \leftarrow X + 1^{\ell 2} \;||\; {}^{\ell 3}X \leftarrow X + 1^{\ell 4}$ composed of two identical threads incrementing X once. The reachable local states on thread 1 at $\ell 2$ with auxiliary variables are described by $(pc_2 = 3 \wedge X = 1) \vee (pc_2 = 4 \wedge X = 2)$, which implies $X \in [1, 2]$. Its image by α_{aux} simply gives $X \in [1, 2]$, which is no longer invariant by the transition set $\{\, \langle \ell 3,\, [X \mapsto x] \rangle \to \langle \ell 4,\, [X \mapsto x+1] \rangle \mid x \in \mathbb{R} \,\}$ generated by $X \leftarrow X + 1$ in the second thread. Hence, $X \in [1, 2]$ cannot be proved to hold at $\ell 2$ after removing pc_2.
End of example.

This example also shows that invariants involving auxiliary variables tend to be disjunctive, hence, non-convex. Classic numeric domains, such as polyhedra and intervals, are not well equipped to handle non-convex sets. When keeping auxiliary variables, one possible solution is to use value partitioning domains, such as decision trees [BCC+10a] that are inspired by binary decision diagrams [Bry86] and mix a discrete, enumerative part (for auxiliary variables) and a more conventional numeric one (for program variables).

Interference abstraction. An interference set $I \in \mathcal{P}(\Sigma \times \Sigma)$ is a relation on states. A pair $\langle \langle \ell,\, \rho \rangle,\, \langle \ell',\, \rho' \rangle \rangle \in I$ modeling a transition can be seen as a mapping r from the variable set $\overline{\mathcal{V}} \stackrel{\text{def}}{=} \{\, V, V' \mid V \in \mathcal{V} \,\} \cup \{\, pc_1, \ldots, pc_N, pc'_1, \ldots, pc'_N \,\}$ to values, where $\forall V \in \mathcal{V} : r(V) = \rho(V) \wedge r(V') = \rho'(V)$ and $\forall t : r(pc_t) = \ell(t) \wedge r(pc'_t) = \ell'(t)$. Hence I can be abstracted as an abstract value in a numeric abstract domain (possibly using value partitioning).

The example in Fig. 3.2 uses a single global interference, which shows that the control flow information is not always useful. A more efficient analysis can thus be achieved using the *flow-insensitive abstraction* α_{flow}:

$$\begin{array}{l} \alpha_{flow} \in \mathcal{P}(\Sigma \times \Sigma) \to \mathcal{P}(\mathcal{E} \times \mathcal{E}) \\ \alpha_{flow} \stackrel{\text{def}}{=} \lambda I.\, \{\, \langle \rho,\, \rho' \rangle \mid \exists \ell, \ell' : \langle \langle \ell,\, \rho \rangle,\, \langle \ell',\, \rho' \rangle \rangle \in I \,\} \;. \end{array} \tag{3.12}$$

Finally, we propose an abstraction of $\mathcal{P}(\mathcal{E} \times \mathcal{E})$ that only remembers which variables have changed and their new value. Hence, an abstract relation is an element $X^\sharp$ of $\mathcal{D}^\sharp_{chg} \stackrel{\text{def}}{=} \mathcal{V} \to \mathcal{P}(\mathbb{R})$ and represents:

$$\begin{array}{l} \gamma_{chg}(X^\sharp) \stackrel{\text{def}}{=} \{\, \langle \rho,\, \rho' \rangle \in \mathcal{E} \times \mathcal{E} \mid \forall V \in \mathcal{V} : \\ \qquad\qquad \rho(V) = \rho'(V) \vee \rho'(V) \in X^\sharp(V) \,\} \end{array} \tag{3.13}$$

and the associated abstraction is:

$$\begin{array}{l} \alpha_{chg}(I) \stackrel{\text{def}}{=} \\ \lambda V.\, \{\, x \in \mathbb{R} \mid \exists \langle \rho,\, \rho' \rangle \in I : \rho(V) \neq x \wedge \rho'(V) = x \,\} \;. \end{array} \tag{3.14}$$

A map $\mathcal{V} \to \mathcal{P}(\mathbb{R})$ can be further abstracted into $\mathcal{V} \to \mathcal{R}^\sharp$ for any numeric domain $\mathcal{R}^\sharp$ abstracting the value of a single variable, such as the interval domain from Ex. 2.2.1.[1] This is an efficient abstraction which, although coarse, can nevertheless infer important information.

Example 3.2.3. Consider again the example from Fig. 3.2. Abstracting the interferences with α_{flow} and α_{chg} yields the map $[X \mapsto [1, 10], Y \mapsto \emptyset]$ for t_1, and $[X \mapsto \emptyset, Y \mapsto [1, 10]]$ for t_2. It successfully captures the fact that t_1 does not modify Y, and that t_2 stores values from $[1, 10]$ into Y and does not modify X, which is sufficient to prove that X and Y are bounded in $[1, 10]$ at all program locations. However, the abstract interference cannot represent the fact that Y is only increased by t_2. As a consequence, it is impossible to infer the program invariant $X \leq Y$, even when abstracting the local state in a relational domain (such as polyhedra) that can represent this invariant.
End of example.

The abstraction α_{chg} is *non-relational* in both the senses that it cannot represent relationships between program variables, and that it cannot represent relationships between the state before and the state after a transition.

3.2.4 Unbounded number of threads

We have assumed, since Sec. 3.1.1, a fixed, finite set of threads $\mathcal{T}$. However, this hypothesis is never used in our derivations and all our formulas apply even if $\mathcal{T}$ is infinite. An infinite number of threads is useful to model, in the absence of dynamic thread creation, programs that exhibit an *unbounded number of threads*: at initialization, we choose non-deterministically a finite subset in the infinite thread pool allowed to run. Our finiteness restriction is only here to guarantee that the construction results in an effective static analyzer. Indeed, when considering an infinite number of threads, we encounter the following issues: firstly, iterating over the threads in Thm. 3.2.1 may not terminate; secondly, the number of reachable local state sets $\mathcal{Rl}(t)$ to abstract is infinite; thirdly, the number of auxiliary variables is infinite and we cannot apply numeric abstractions defined in vector spaces of finite dimensions.

Nevertheless, these issues can be side-step in a restricted but useful case. Firstly, we assume that we have a finite set $\mathcal{T}$ of syntactic threads, but one of them, $t^* \in \mathcal{T}$, has an infinite number of instances running concurrently (this can be extended easily to the case of several infinite threads). Secondly, we assume that we remove auxiliary variables in the abstraction of local states (using α_{aux}) and that we abstract interferences in a flow-insensitive way (using α_{flow}). These conditions ensure that the local states and the interferences have a finite number of variables, and can be abstracted in a numeric abstract domain. They also ensure that, given two instances t^*_1 and t^*_2 of t^*, we have $\mathcal{Rl}(t^*_1) \simeq \mathcal{Rl}(t^*_2)$ and $R_{t^*_1} \simeq R_{t^*_2}$ up to the abstractions α_{aux} and α_{flow}. Hence, it is sufficient to represent only one instance of t^* in $\mathcal{Rl}$ and iterate over one instance of R_{t^*} in Thm. 3.2.1. However, the definition of R_{t^*} is changed slightly

[1] Note that the domain $\mathcal{V} \to \mathcal{R}^\sharp_i$ is not equivalent to the interval domain $\mathcal{D}^\sharp_i$ presented in Sec. 2.4.1. The former abstracts $\mathcal{V} \to \mathcal{P}(\mathbb{R})$ while the latter abstracts $\mathcal{P}(\mathcal{V} \to \mathbb{R})$. As a consequence, least elements $\bot^\sharp$ are coalescent in the later domain and not in the former one.

into:

$$R_{t^*} \stackrel{\text{def}}{=} \lambda Y.\, \lambda X. \\ \pi_{t^*}(I) \cup \{\, \pi_{t^*}(\sigma') \mid \exists \pi_{t^*}(\sigma) \in X : \\ \sigma \xrightarrow{t^*}_{\tau} \sigma' \vee \exists t' : \langle \sigma, \sigma' \rangle \in Y(t') \,\} \tag{3.15}$$

which simply removes the condition $t' \neq t^*$ to take into account "self-interferences," i.e., interferences on an instance of t^* caused by another instance of t^* (note that R_t remains unchanged when $t \neq t^*$).

We have thus reduced the analysis of a concurrent program with unbounded thread instances to the analysis of a concurrent program with only a bounded number of threads. This comes at some cost in precision. The resulting analysis cannot distinguish between the different instances of the same thread nor express properties that depend on the number of running threads: the analysis is uniform. Non-uniform analyses are quite rare (a main example is Feret's occurrence counting analysis for the $\pi-$calculus [Fer01]), and designing a non-uniform analysis in our framework remains a challenging future work.

3.3 Big-step interference analysis

In this section, we apply the general principles and abstractions from the preceding section to construct, on our concurrent language, a thread-modular static analysis in big-step form, similar to the one we developed for sequential programs (Sec. 2.3.5). The construction is described in more details in [Min11, Min12d].

3.3.1 Concrete interference semantics

We start by enriching the concrete semantics of expressions (Fig. 2.2) and statements (Fig. 2.6) with a notion of interference. With in mind the abstractions from Sec. 3.2.3, we model interferences as values written into variables, in a flow-insensitive and non-relational way.

Formally, *interferences* live in $\mathcal{Itf} \stackrel{\text{def}}{=} \mathcal{T} \times \mathcal{V} \times \mathbb{R}$, where a triple $\langle t, X, v \rangle \in \mathcal{Itf}$ means that the thread t can write the value v into the variable X.

Expression semantics. We define $\mathbb{E}_{\mathcal{Itf}}[\![\ expr\ ]\!]_t\langle \rho, I \rangle$, the semantics of expressions with interferences, in Fig. 3.3. Compared to the original semantics of expression $\mathbb{E}[\![\ expr\ ]\!]\rho$ introduced in Fig. 2.2, it takes as argument, in addition to an environment $\rho \in \mathcal{E}$, a set $I \subseteq \mathcal{Itf}$ of interferences. Moreover, the semantics is parametrized by the identifier $t \in \mathcal{T}$ of the thread that evaluates the expression. When reading the value of a variable $X \in \mathcal{V}$, all the interferences $\langle t', X, v \rangle \in I$ from threads $t' \neq t$ are applied, i.e., evaluating X may non-deterministically evaluate either to $\rho(X)$ or to any value v written to X by another thread. The semantics of operators and constants is not changed, apart from propagating I to sub-expressions. Note that different occurrences of the same variable in an expression may evaluate, in the same environment, to different values. Intuitively, evaluating an expression is longer an atomic action: the value of a variable may change due to thread interferences during the evaluation (we will formalize this remark in Sec. 3.5).

Statement semantics. The semantic domain of statements $\mathcal{D}_{\mathcal{Itf}}$ is similar to that of sequential programs from Fig. 2.6, but enriched with interferences. We use: $\mathcal{D}_{\mathcal{Itf}} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega) \times \mathcal{P}(\mathcal{Itf})$, which is a complete lattice ordered by element-wise set inclusion. Given a statement *stat* executed by a thread t, its semantics is the join-morphism $\mathbb{S}_{\mathcal{Itf}}[\![\ stat\ ]\!]_t$ defined in Fig. 3.4. There, the join $\sqcup$ denotes the element-wise set union. Compared to $\mathbb{S}[\![\ stat\ ]\!]$ from Fig. 2.6, $\mathbb{S}_{\mathcal{Itf}}[\![\ stat\ ]\!]_t$ passes down the interference I to $\mathbb{E}_{\mathcal{Itf}}[\![\ e\ ]\!]_t$ every time an expression e needs to be evaluated. Moreover, it outputs its argument interference set enriched with the interferences that are created in thread t by all the assignments it performs.

Program semantics. A first step in the analysis of a concurrent program is to collect the interferences *itf*. The analysis of a single thread computes its interferences assuming *a priori* knowledge of the interferences from the other threads. This circular dependency is solved by computing interferences as a fixpoint:

$$itf \stackrel{\text{def}}{=} \operatorname{lfp} \lambda I.\, \bigcup_{t \in \mathcal{T}} \left[\mathbb{S}_{\mathcal{Itf}}[\![\ stat_t\ ]\!]_t \langle \lambda V.\, 0, \emptyset, I \rangle\right]_{\mathcal{Itf}} \tag{3.16}$$

where $[\cdot]_{\mathcal{Itf}}$ restricts the triple output by $\mathbb{S}_{\mathcal{Itf}}[\![\ stat_t\ ]\!]_t$ to the component in $\mathcal{P}(\mathcal{Itf})$. The set of run-time errors O in the concurrent program can then be extracted by running the analysis again and gathering this time the component in $\mathcal{P}(\Omega)$:

$$O \stackrel{\text{def}}{=} \bigcup_{t \in \mathcal{T}} \left[\mathbb{S}_{\mathcal{Itf}}[\![\ stat_t\ ]\!]_t \langle \lambda V.\, 0, \emptyset, itf \rangle\right]_{\Omega} \ . \tag{3.17}$$

The following soundness theorem is proved in [Min12d]:

Theorem 3.3.1. $\mathcal{R} \cap \Omega \subseteq O$.

The converse inequality does not hold in general. Although it manipulates uncomputable concrete sets of environments and interferences, our semantics is a strict over-approximation of the reachable state semantics because we use a non-relational and flow-insensitive abstraction of interferences and do not use auxiliary variables (see also Exs. 3.3.1 and 3.3.2).

Output environments. In addition to errors and interferences, $\mathbb{S}_{\mathcal{Itf}}[\![\ stat_t\ ]\!]_t$ outputs a set of environments R_t. In the case of sequential programs (Sec. 2.3.5), these corresponded to the program states reachable at the end of the program. However, this is not the case for $\mathbb{S}_{\mathcal{Itf}}[\![\ stat_t\ ]\!]_t$: the environments in R_t only take into account the values written by the thread t. Writes from the other threads, that can nevertheless contribute to the program state, are stored in the interference set *itf*. An overapproximation of the reachable state set at the end of the program can be constructed by combining these two information:

$$X_t \stackrel{\text{def}}{=} \{\, \rho \mid \exists \rho' \in R_t : \forall V \in \mathcal{V} : \\ \rho(V) = \rho'(V) \vee \exists t' \neq t :\ (t', V, \rho(V)) \in itf \,\} \ .$$

Note that each thread $t \in \mathcal{T}$ may give a different overapproximation X_t. It is possible to combine them to gain more precision, which leads to $\cap_{t \in \mathcal{T}} X_t$.

The environments R_t computed by our big-step semantics are thus not actually local invariants, which departs from our formalization of rely-guarantee in Sec. 3.2. The rationale is to avoid adding to the environment redundant information that is already available in interferences. Clearly separating the effect

$$\underline{\mathbb{E}_{\mathcal{I}tf}[\![\ expr\ ]\!]_t \in (\mathcal{E} \times \mathcal{P}(\mathcal{I}tf)) \to (\mathcal{P}(\mathbb{R}) \times \mathcal{P}(\Omega))}$$

$$\mathbb{E}_{\mathcal{I}tf}[\![\ X\ ]\!]_t\langle \rho,\ I\rangle \stackrel{\text{def}}{=} \langle \{\ \rho(X)\ \} \cup \{\ v \mid \exists t' \neq t :\ \langle t',\ X,\ v\rangle \in I\ \},\ \emptyset\rangle$$

$$\mathbb{E}_{\mathcal{I}tf}[\![\ [c_1, c_2]\ ]\!]_t\langle \rho,\ I\rangle \stackrel{\text{def}}{=} \langle \{\ x \in \mathbb{R} \mid c_1 \leq c \leq c_2\ \},\ \emptyset\rangle$$

$$\mathbb{E}_{\mathcal{I}tf}[\![\ -_\omega\ e\ ]\!]_t\langle \rho,\ I\rangle \stackrel{\text{def}}{=} \text{let } \langle V,\ O\rangle = \mathbb{E}_{\mathcal{I}tf}[\![\ e\ ]\!]_t\ \langle \rho,\ I\rangle \text{ in } \langle \{\ -v \mid v \in V\ \},\ O\rangle$$

$$\begin{aligned}\mathbb{E}_{\mathcal{I}tf}[\![\ e_1 \diamond_\omega e_2\ ]\!]_t\langle \rho,\ I\rangle \stackrel{\text{def}}{=}\ & \text{let } \langle V_1,\ O_1\rangle = \mathbb{E}_{\mathcal{I}tf}[\![\ e_1\ ]\!]_t\ \langle \rho,\ I\rangle \text{ in}\\ & \text{let } \langle V_2,\ O_2\rangle = \mathbb{E}_{\mathcal{I}tf}[\![\ e_2\ ]\!]_t\ \langle \rho,\ I\rangle \text{ in}\\ & \langle \{\ v_1 \diamond_\omega v_2 \mid v_1 \in V_1,\ v_2 \in V_2,\ \diamond \neq\ / \vee v_2 \neq 0\ \},\\ & O_1 \cup O_2 \cup \{\ \omega \mid \diamond =\ / \wedge\ 0 \in V_2\ \}\rangle\end{aligned}$$

Figure 3.3: Semantics of expressions with interference.

$$\underline{\mathbb{S}_{\mathcal{I}tf}[\![\ stat\ ]\!]_t \in \mathcal{D}_{\mathcal{I}tf} \longrightarrow \mathcal{D}_{\mathcal{I}tf}}$$

$$\mathbb{S}_{\mathcal{I}tf}[\![\ X \leftarrow e\ ]\!]_t\langle R,\ O,\ I\rangle \stackrel{\text{def}}{=} \langle \emptyset,\ O,\ I\rangle \sqcup \bigsqcup_{\rho \in R} \begin{array}{l}\text{let } \langle V,\ O'\rangle = \mathbb{E}_{\mathcal{I}tf}[\![\ e\ ]\!]_t\ \langle \rho,\ I\rangle \text{ in}\\ \langle \{\ \rho[X \mapsto v] \mid v \in V\ \},\ O',\ \{\ \langle t,\ X,\ v\rangle \mid v \in V\ \}\rangle\end{array}$$

$$\mathbb{S}_{\mathcal{I}tf}[\![\ e \bowtie 0\ ]\!]_t\langle R,\ O,\ I\rangle \stackrel{\text{def}}{=} \langle \emptyset,\ O,\ I\rangle \sqcup \bigsqcup_{\rho \in R} \begin{array}{l}\text{let } \langle V,\ O'\rangle = \mathbb{E}_{\mathcal{I}tf}[\![\ e\ ]\!]_t\ \langle \rho,\ I\rangle \text{ in}\\ \langle \{\ \rho \mid \exists v \in V :\ v \bowtie 0\ \},\ O',\ \emptyset\rangle\end{array}$$

$$\mathbb{S}_{\mathcal{I}tf}[\![\ \textbf{if } e \bowtie 0 \textbf{ then } s\ ]\!]_t\mathcal{X} \stackrel{\text{def}}{=} (\mathbb{S}_{\mathcal{I}tf}[\![\ s\ ]\!]_t \circ \mathbb{S}_{\mathcal{I}tf}[\![\ e \bowtie 0\ ]\!]_t)\mathcal{X} \sqcup \mathbb{S}_{\mathcal{I}tf}[\![\ e \not\bowtie 0\ ]\!]_t\mathcal{X}$$

$$\mathbb{S}_{\mathcal{I}tf}[\![\ \textbf{while } e \bowtie 0 \textbf{ do } s\ ]\!]_t\mathcal{X} \stackrel{\text{def}}{=} \mathbb{S}_{\mathcal{I}tf}[\![\ e \not\bowtie 0\ ]\!]_t\ (\text{lfp } \lambda\mathcal{Y}.\ \mathcal{X} \sqcup (\mathbb{S}_{\mathcal{I}tf}[\![\ s\ ]\!]_t \circ \mathbb{S}_{\mathcal{I}tf}[\![\ e \bowtie 0\ ]\!]_t)\mathcal{Y})$$

$$\mathbb{S}_{\mathcal{I}tf}[\![\ s_1;\ s_2\ ]\!]_t \stackrel{\text{def}}{=} \mathbb{S}_{\mathcal{I}tf}[\![\ s_2\ ]\!]_t \circ \mathbb{S}_{\mathcal{I}tf}[\![\ s_1\ ]\!]_t$$

Figure 3.4: Big-step semantics with interference.

of writes from the current thread and from the other threads will also prove useful when considering mutual exclusion constraints, in Sec. 3.4.

Example 3.3.1. We exemplify our semantics on the program of Fig. 3.1 by computing the fixpoint *itf* (3.16) by iteration. To be concise, we only show the value of I at each iteration, and the environment set at $\ell 1$ and $\ell 4$ (corresponding to loop invariants). Starting from $I = \emptyset$, t_1 does not update X as $X < Y$ is always false, while t_2 increments Y up to 10, which gives, after an iteration:

$$\begin{array}{l}\ell 1 : X = Y = 0\\ \ell 4 : X = 0 \wedge Y \in [0, 10]\\ I = \{\ \langle t_2,\ Y,\ i\rangle \mid i \in [1, 10]\ \}\ .\end{array}$$

As I has increased, we perform the analysis again and, this time, $X < Y$ can be satisfied in t_1, which causes X to be incremented. We get:

$$\begin{array}{l}\ell 1 : X \in [0, 10] \wedge Y = 0\\ \ell 4 : X = 0 \wedge Y \in [0, 10]\\ I = \{\ \langle t_1,\ X,\ i\rangle, \langle t_2,\ Y,\ i\rangle \mid i \in [1, 10]\ \}\ .\end{array}$$

Performing another analysis iteration returns the same I, and the analysis stops. Invariants at $\ell 1$ can be constructed by combining the local environments $X \in [0, 10] \wedge Y = 0$ with the interferences from t_2 in I to get: $X \in [0, 10] \wedge Y \in [0, 10]$. As expected, the invariant $X \leq Y$ is not inferred, because interferences do not carry any relational information.
End of example.

Example 3.3.2. Consider again the program $prog = {}^{\ell 1}X \leftarrow X + 1^{\ell 2}\ ||\ {}^{\ell 3}X \leftarrow X + 1^{\ell 4}$ from Ex. 3.2.2. The fixpoint iteration for *itf* (3.16) will compute an infinite increasing sequence, even though the program features no loop: at step n, we get $I = \{\ \langle t_1,\ X,\ i\rangle, \langle t_2,\ X,\ i\rangle \mid 0 \leq i < n\ \}$. Due to the flow-insensitive abstraction of interferences, we are not able to infer that each thread can increment X at most once.
End of example.

3.3.2 Abstract interference semantics

In order to construct an effective static analysis, it remains to abstract the semantic domain $\mathcal{D}_{\mathcal{I}tf} = \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Omega) \times \mathcal{P}(\mathcal{I}tf)$ into a computable abstract domain $\mathcal{D}^\sharp_{\mathcal{I}tf}$ with sound abstractions $\mathbb{S}^\sharp_{\mathcal{I}tf}[\![\ stat\ ]\!]_t$ of $\mathbb{S}_{\mathcal{I}tf}[\![\ stat\ ]\!]_t$. This is not very difficult as $\mathbb{S}_{\mathcal{I}tf}[\![\ stat\ ]\!]_t$ is very close to the semantics $\mathbb{S}[\![\ stat\ ]\!]$ of sequential programs introduced in Fig. 2.6, and for which we already designed abstractions $\mathbb{S}^\sharp[\![\ stat\ ]\!]$ in Sec. 2.3.6.

Firstly, any numeric domain $\mathcal{E}^\sharp$ can be used to abstract $\mathcal{P}(\mathcal{E})$. For interferences, we note that $\mathcal{P}(\mathcal{I}tf) = \mathcal{P}(\mathcal{T} \times \mathcal{V} \times \mathbb{R}) \simeq (\mathcal{T} \times \mathcal{V}) \to \mathcal{P}(\mathbb{R})$. Abstracting $\mathcal{P}(\mathcal{I}tf)$ can be reduced to the problem of abstracting $\mathcal{P}(\mathbb{R})$. This situation is similar to the abstraction of flow-insensitive interferences described in Sec. 3.2.3. We thus assume that we are given a numeric domain $\mathcal{R}^\sharp$ for one variable, and denote by $\mathcal{I}tf^\sharp$ its point-wise lifting into functions in $\mathcal{I}tf^\sharp \stackrel{\text{def}}{=} (\mathcal{T} \times \mathcal{V}) \to \mathcal{R}^\sharp$: an abstract interference $I^\sharp \in \mathcal{I}tf^\sharp$ maps each thread and each variable to an abstract set of reals. Finally, we state:

$$\mathcal{D}^\sharp_{\mathcal{I}tf} \stackrel{\text{def}}{=} \mathcal{E}^\sharp \times \mathcal{P}(\Omega) \times \mathcal{I}tf^\sharp\ . \tag{3.18}$$

Secondly, we design $\mathbb{S}^\sharp_{\mathcal{I}tf}[\![\ stat\ ]\!]_t$ for conditionals, loops, and sequences by induction, independently from the abstract domain, exactly as $\mathbb{S}^\sharp[\![\ stat\ ]\!]$ in Fig. 2.7. Only the base case of assignments and tests needs some adaptation, which we describe informally (we refer the interested reader to [Min12d]

for a full formal presentation). Tests $\mathbb{S}^\sharp_{\mathcal{I}tf}[\![e \bowtie 0]\!]_t\langle R^\sharp, O, I^\sharp\rangle$ are reduced to the case of an interference-free abstraction with a slight change of expression $\mathbb{S}^\sharp[\![e' \bowtie 0]\!]\langle R^\sharp, O\rangle$ as follows:

- for each variable V appearing in e, we collect its interferences from other threads $V^\sharp \stackrel{\text{def}}{=} \bigcup^\sharp_{t' \neq t} I^\sharp\langle t', V\rangle$,
- if $V^\sharp \neq \bot^\sharp$, we then compute a range $[a_V, b_V]$ that contains both $\gamma_{\mathcal{R}}(V^\sharp)$ and the range of V in $\gamma_{\mathcal{E}}(R^\sharp)$,
- finally, we construct e' by replacing in e all the occurrences of V with $[a_V, b_V]$ (if $V^\sharp = \bot^\sharp$, V is left intact).

The case of assignments $\mathbb{S}^\sharp_{\mathcal{I}tf}[\![V \leftarrow e]\!]_t$ is similar, and only slightly complicated by the need to enrich the abstract interferences with the effect of the assignment. This can be achieved by replacing, in $I^\sharp$, $I^\sharp\langle t, V\rangle$ with its join with an abstraction in $\mathcal{R}^\sharp$ of the value of V after the assignment.

Finally, when abstracting the interference fixpoint itf, we must take care to ensure the convergence of the iterates. We use an interference widening $\triangledown_{\mathcal{I}tf}$, which is simply the widening $\triangledown_{\mathcal{R}}$ on $\mathcal{R}^\sharp$ applied element-wise to each pair $\langle t, V\rangle \in \mathcal{T} \times \mathcal{V}$. The abstract interferences $itf^\sharp$ are then computed as in (3.16):

$$itf^\sharp \stackrel{\text{def}}{=} \lim \lambda I^\sharp.\, I^\sharp \triangledown_{\mathcal{I}tf} \bigcup^\sharp_{t\in\mathcal{T}} \left[\mathbb{S}^\sharp_{\mathcal{I}tf}[\![stat_t]\!]_t\langle E^\sharp_0, \emptyset, I^\sharp\rangle\right]_{\mathcal{I}tf} . \tag{3.19}$$

The result is an effective static analysis in big-step form which is thread-modular, parametrized by a choice of numeric abstract domains, and can reuse existing big-step static analyses of sequential domains with minimal change. Moreover, although thread interferences are abstracted in a flow-insensitive and non-relational way, the analysis of each thread is fully flow-sensitive and can use relational numeric domains $\mathcal{E}^\sharp$ to abstract memory states.

Example 3.3.3. The analysis of the program in Fig. 3.1 using the interval abstraction for both $\mathcal{E}^\sharp$ and $\mathcal{R}^\sharp$ gives the exact same result as the concrete interference semantics (Ex. 3.3.1). In particular, the sequence of abstract interference iterates with widening (3.19) gives (omitting mappings to $\bot^\sharp$):

$$\begin{aligned} I_1 &= [] \\ I_2 &= [\langle t_2, Y\rangle \mapsto [1, 10]] \\ I_3 &= [\langle t_1, X\rangle \mapsto [1, 10], \langle t_2, Y\rangle \mapsto [1, 10]] \end{aligned}$$

at which point it is stable.
End of example.

Unbounded thread instances. The analysis assumes a finite, fixed number of threads, but it is not difficult to adapt it to handle threads with an unbounded number of instances. Following the remark in Sec. 3.2.4, we construct a uniform analysis where a thread t^* has an unbounded number of instances by taking into account self-interferences for t^*. This is achieved by omitting the condition $t' \neq t$ in the definition of $V^\sharp$ when applying interferences to expressions; it becomes simply: $V^\sharp \stackrel{\text{def}}{=} \bigcup^\sharp_{t'\in\mathcal{T}} I^\sharp\langle t', V\rangle$.

3.4 Scheduling

The model of executions considered up to now allows arbitrary interleavings. In practice, however, the scheduling of threads can be controlled to some extent. This can be achieved by executing synchronisation primitives offered in the language (such as locks) or by controlling directly some parameters of the scheduler (such as thread priorities). Note that our interference analysis considers non-deterministic scheduling, and so, is sound in the context of restricted scheduling. We now show how scheduling restrictions can be taken into account to achieve a more precise analysis.

As there exist numerous synchronization schemes and scheduling policies, we focus on two simple but useful cases: mutual exclusion locks (Sec. 3.4.1) and real-time scheduling with fixed priorities (Sec. 3.4.2). Moreover, for the sake of concision, we present our semantics informally, and refer the reader to [Min12d] for the detailed formalization.

3.4.1 Mutexes

Mutual exclusion locks (thereby referred to as "*mutexes*") are a standard low-level synchronization primitive offered by most concurrent languages and concurrency libraries (such as POSIX Threads [IT95]). The core property of mutexes is that each mutex cannot be acquired by more than one thread at a time: a thread trying to lock a mutex already locked by another thread will wait until the mutex is available; it will not resume its execution before it can lock the mutex. Mutexes are useful to delimit *critical sections*, i.e., section of the program that only one thread can enter at a time. We assume the existence of a fixed, finite set $\mathcal{M}$ of mutexes and add statements to lock and unlock them:

$$stat ::= \mathbf{lock}(m) \mid \mathbf{unlock}(m) \quad m \in \mathcal{M} . \tag{3.20}$$

In the context of an interference semantics, we can then take advantage of the mutual exclusion property of mutexes to restrict the effect of interferences according to which mutexes each thread holds.

Interferences. We illustrate how mutual exclusion restricts interferences in Fig. 3.5. We denote respectively as R and W reads from and writes into a shared variable X; for the sake of presentation, we model only the effect of thread 1 on thread 2. In Fig. 3.5.(a), all the accesses are protected by the mutex m and thread 1 writes twice into X while holding m. When thread 2 locks m and reads X, it can see the second value written by thread 1, but never the first one, which is necessarily overwritten before m is acquired. Moreover, after thread 2 locks m and overwrites X while holding m, it can only read back the value it has written, unaffected by the interferences from thread 1. This kind of interferences, that carries a small amount of flow-sensitivity, will be called "*synchronized*."

Figure 3.5.(b) describes a case where the accesses are not all protected by the mutex. In addition to the synchronized interference from Fig. 3.5.(a) (and not repeated here) any write by thread 1 will influence all the reads by thread 2 occurring when thread 2 does not hold m, and any write by thread 1 occurring when thread 1 does not hold m influences all the reads by thread 2. These interferences, where the read / write pairs are not protected by a common mutex, are called "*non synchronized*."

Partitioning. To model these interferences, we track, in our concrete interference semantics, the exact set of mutexes locked by the current thread, and associate interferences with the mutexes locked when the corresponding write was performed. This

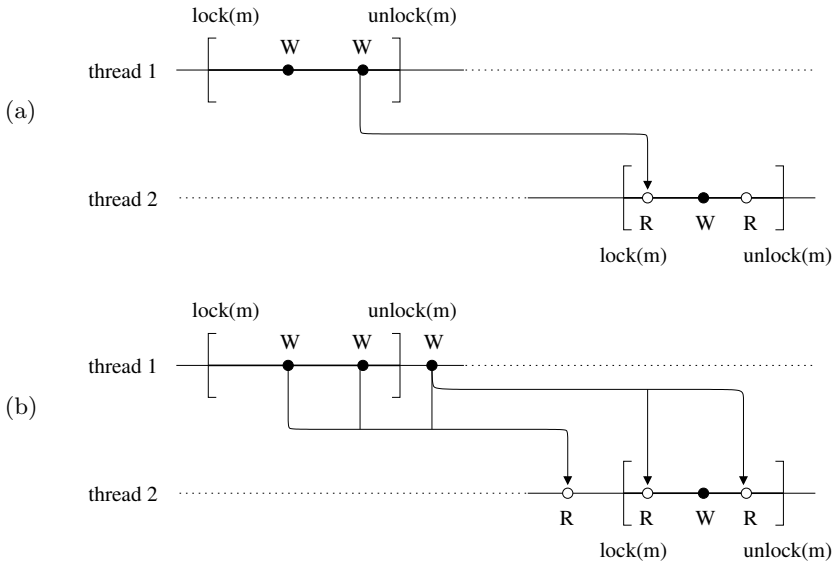


Figure 3.5: Synchronized (a) and non synchronized (b) interferences in the presence of a mutex.

t_1, t_1'	t_2, t_2'
while $0 = 0$ **do**	**while** $0 = 0$ **do**
$\quad {}^{\ell 1}$**lock**(m);	$\quad {}^{\ell 2}$**lock**(m);
$\quad$ **if** $X > 0$ **then**	$\quad X \leftarrow X + 1$;
$\quad\quad X \leftarrow X - 1$	$\quad$ **if** $X > 10$ **then**
$\quad$ **endif**;	$\quad\quad X \leftarrow 10$
$\quad$ **unlock**(m)	$\quad$ **endif**;
done	$\quad$ **unlock**(m)
	done

Figure 3.6: Abstract consumers / producers.

is achieved through *partitioning*: environments $\mathcal{P}(\mathcal{E})$ and interferences $\mathcal{P}(\mathcal{I}tf)$ are replaced, respectively, with maps $\mathcal{S} \to \mathcal{P}(\mathcal{E})$ and $\mathcal{S} \to \mathcal{P}(\mathcal{I}tf)$, where $\mathcal{S} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{M})$.

The semantics of expressions with interferences (Fig. 3.3) is changed as follows to take into account non synchronized interferences:

$$\begin{array}{l}\mathbb{E}_{\mathcal{I}tf}[\![\, X \,]\!]_t\langle M,\, \rho,\, I\rangle \stackrel{\text{def}}{=} \\ \quad \langle \{\, \rho(X) \,\} \;\cup\; \{\, v \mid \exists t' \neq t, M' \in \mathcal{S} : \\ \qquad\qquad \langle t',\, X,\, v\rangle \in I(M') \wedge M \cap M' = \emptyset \,\},\, \emptyset\rangle\end{array}$$

where $\langle M,\, \rho\rangle \in \mathcal{S} \times \mathcal{E}$ denotes the state in which the expression is evaluated, as a set M of mutexes held and an environment ρ. The new condition $M \cap M' = \emptyset$ models the absence of mutual exclusion.

Synchronized interferences are handled by collecting, at each **unlock**(m) instruction, the current value of all the variables modified since the last **lock**(m) instruction, which is held in a special interference partition attached to m. Then, when another thread performs a **lock**(m) instruction, the interferences are imported into the environment.

Example 3.4.1. Figure 3.6 presents a classic producer / consumer program, abstracted away so that we only keep track, in X, of the number of resources available. The (identical) threads t_1 and t_1' consume resources ($X \leftarrow X - 1$), if available ($X > 0$), while threads t_2 and t_2' produce resources ($X \leftarrow X + 1$) up to a limit ($X \leq 10$). All the accesses to X are protected by a mutex m. As a consequence, the statement **if** $X > 0$ **then** $X \leftarrow X - 1$ **endif** is free from non synchronized interference. In particular, it is free from interferences from another consumer: it is not possible for any thread to modify X between the test ensuring that $X > 0$ and its subsequent decrementation, which is key to prove that X stays positive. Likewise, we can prove that $X \leq 10$. The synchronized interferences associate $[X \mapsto [0, 9]]$ to m in t_1 and t_1', and $[X \mapsto [1, 10]]$ to m in t_2 and t_2'. These value sets are imported in the environments when locking m, respectively at $\ell 2$ and $\ell 1$.
End of example.

We presented interference partitioning only at the level of the concrete interference semantics. It is straightforward to derive a computable abstract semantics parametrized by abstract domains, as in Sec. 3.3.2 (see also [Min12d]).

Data-race detection. *Data-races* occur when two threads can access the same variable, one access at least is a write, and the accesses are not protected by some common mutex. In our semantics, data-races correspond to non synchronized interferences. Hence, it is straightforward to extent our interference static analysis to detect all data-races in a sound way.

Deadlock detection. A *deadlock* occurs when there exists a subset of threads such that each thread in the subset is waiting for a lock held by another thread of the subset to be unlocked. Hence, the threads in the subset are blocked indefinitely.

Example 3.4.2. The program:

$$\begin{array}{l}\mathit{prog} \stackrel{\text{def}}{=} \\ \quad \textbf{lock}(m_1);\, \textbf{lock}(m_2);\, \textbf{unlock}(m_2);\, \textbf{unlock}(m_1) \;||\; \\ \quad \textbf{lock}(m_2);\, \textbf{lock}(m_1)\; ;\textbf{unlock}(m_1);\, \textbf{unlock}(m_2)\end{array}$$

presents one of the simplest case of deadlock: if the first thread locks m_1 and then the second thread locks m_2, then neither can advance further.
End of example.

t_h	t_l
$L \leftarrow$ **islocked**(m);	**lock**(m);
if $L = 0$ **then**	$Z \leftarrow Y$;
$\quad Y \leftarrow Y + 1$;	$Y \leftarrow 0$;
$\quad$ **yeild**	**unlock**(m)
endif	

Figure 3.7: Priority-based critical sections.

Deadlocks can be easily detected in our semantics. It is sufficient to collect, for each **lock**(m) instruction in each thread t, the set M of mutexes t already holds just before issuing the instruction, which we denote as a triple $\langle t, m, M\rangle$. This information is readily available in our concrete and abstract semantics. Then, we check for the existence of a set of collected triples $\langle t_1, m_1, M_1\rangle, \ldots, \langle t_k, m_k, M_k\rangle$ such that: $\forall i \neq j : t_i \neq t_j \wedge M_i \cap M_j = \emptyset$, and $\forall i : \exists j : m_i \in M_j$.[2] This results in a sound over-approximation of the set of possible deadlocks. In general, the over-approximation is strict and introduces spurious deadlocks, as it can consider sets of triples that may never be reachable simultaneously in any program execution.

3.4.2 Real-time scheduling

Real-time operating systems are a flavor of operating systems that offer more guarantees in terms of determinism and execution times than general-purpose ones. They are thus used in most embedded applications (in avionics, for instance, with the ARINC 653 standard [Aer] considered by our analyzer AstréeA). We are not interested here in the timing guarantees (physical time is, after all, not tracked in our semantics), but in another property of real-time systems: the strict interpretation of thread priorities by the scheduler. Each thread is given a *priority*, and a lower level priority thread can never preempt a higher level priority thread unless it is blocked.

The analysis presented so far is, of course, sound with respect to any scheduler, including a real-time one. In this section, we show that a more precise analysis can be achieved by using properties that are specific to real-time schedulers. More precisely, we assume that each thread is given a distinct and fixed priority, and that a single thread executes at a time (i.e., there is no true parallelism, but only time-sharing on a single execution unit). Then, the scheduler ensures that the unblocked thread of highest priority is the only one to run. Here, blocking means: either waiting for a mutex to be unlocked by another thread, or waiting for an external event to occur. To account for this second case, we add a new statement **yeild** which models waiting for a non-deterministic amount of time: by yielding, a high priority thread allows lower priority threads to run, but reserves the right to interrupt them at any point and resume its own execution. Despite a strict policy on thread priorities, a real-time scheduler still allows a large amount of non-determinism.

Motivation. Our study of real-time programs is motivated by the example in Fig. 3.7. It implements a critical section protected by a mutex m, but without the need for the high priority thread t_h to actually lock the mutex m. Instead, t_h tests, with the $L \leftarrow$ **islocked**(m) statement (which stores 1 into L if m is locked, and 0 otherwise), whether the lower priority thread, t_l, has locked m. If it has not, t_h can enter its critical section, confident that t_l cannot interrupt it and enter its own critical section. The critical section ends when t_h performs a **yeild** to enable preemption by t_l. This example cannot be analyzed precisely without handling thread priorities.

Partitioning. In order to benefit from a real-time scheduler, we enrich the partitioning mechanism introduced for locks in Sec. 3.4.1. Each statement $L \leftarrow$ **islocked**(m) will create two partitions: one where m is assumed to be locked by another thread and 1 is stored into L, and another where m is assumed to be unlocked and 0 is stored into L. This partitioning allows representing relations between the value of variables and an abstraction of the scheduling state. The partitions are merged when a **yeild** instruction is encountered, as it becomes possible for the lower priority thread to run and invalidate our assumption about the status of the mutex m. As for mutexes, we do not detail the resulting concrete interference semantics nor its straightforward abstraction; these can be found in [Min12d].

3.5 Weakly consistent memories

Up to now in this chapter, we have assumed a straightforward execution model for concurrent programs, stating that: a program execution is an interleaving of the execution of instructions from the threads, that expression evaluations and assignments are atomic, and that a value written by a thread into the memory is immediately available for the next executing thread to read. This attractive model, based on Lamport's notion of *sequential consistency* [Lam79], is, unfortunately, no longer realistic. Concurrent programs running on current hardware may exhibit behaviors that are not sequentially consistent. This issue is now widely recognized and many recent works in semantics and verification take relaxed models into account; this includes the fields of language specification [MPA05], program testing [AMSS11], model checking [ABBM10], theorem proving [ŠA08], and abstract interpretation [Fer08].

In this section, we present the relaxed execution model that we introduced in [Min11, Min12d], and study its static analysis. More precisely, we state that the interference-based analysis we use in Secs. 3.3 and 3.4 is sound with respect to our relaxed model.

3.5.1 Non-consistent behaviors

We first present some example symptoms and causes of non-sequentially consistent behaviors that motivate our model.

Non-consistent memories. In modern architectures, read and write operations do not act directly and instantaneously on the shared memory, but through a hierarchy of caches and store buffers. As observed by Lamport already in the late 70s on the case of distributed memories [Lam78], this can result in behaviors that are not sequentially consistent.

Example 3.5.1. Figure 3.8 presents a simple mutual exclusion algorithm. To ensure that both threads cannot be simultaneously in their critical section, each thread signals its intent to enter it by raising a flag, and then tests the other thread's flag. This is an extremely simplified version of Dekker's algorithm [Dij68]. However, if the assignment performed by a thread is

[2] This check includes the case of a single thread locking the same mutex twice, which is considered here to produce a deadlock.

t_1	t_2
flag1 $\leftarrow 1$;	*flag2* $\leftarrow 1$;
if *flag2* $= 0$ **then**	**if** *flag1* $= 0$ **then**
critical section	*critical section*
endif	**endif**

Figure 3.8: Mutual exclusion algorithm.

t_1	t_2
if *flag2* $= 0$ **then**	**if** *flag1* $= 0$ **then**
flag1 $\leftarrow 1$;	*flag2* $\leftarrow 1$;
critical section	*critical section*
endif	**endif**

Figure 3.9: Reordering independent statements in Fig. 3.8.

propagated asynchronously and takes too long to be acknowledged, it is possible for the other thread to read an outdated 0 flag value and enter its critical section, although the first thread is still executing its own critical section.
End of example.

Optimizations. Another cause for the lack of consistency is the various program transformations and optimizations (such as out of order execution) that are performed by compilers and modern processors. Their validity is generally based only an analysis of a single thread of execution and does not take concurrency issues into account.

Example 3.5.2. Consider again the program in Fig. 3.8. Then, a dependency analysis on thread t_1 in isolation shows that the assignment and the test are independent, so, a compiler can decide to switch their order. The same holds for t_2. The program effectively executed, shown in Fig. 3.9, no longer enforces mutual exclusion.
End of example.

Example 3.5.3. Figure 3.10 presents another transformation that inserts a spurious write into Y, which seems innocuous as Y is not used by t_1 until Y is assigned again. However, this spurious value can be observed by t_2 and stored into X. As a result, when t_1 terminates, Y holds the value 42. The chosen variable Y and value 42 are arbitrary, and so, allowing arbitrary spurious writes makes the program completely unpredictable. This kind of transformation is called "out-of-thin-air," as it introduces unjustified values, and we will not allow it in our model.
End of example.

Atomicity. Finally, we note that the granularity of atomic actions, i.e., the set of points where a thread can be interrupted by the execution of another thread, matters. Consider, for instance, the program $prog = X \leftarrow X + 1 \;||\; X \leftarrow X + 1$. Assuming (as we did) that assignments of arbitrary expressions are atomic, $X = 2$ always holds at the end of the program. However, if they are not, then each thread may read the same value 0 from X before storing 1 into X, so that the program can also end with X equal to 1.

t_1	t_2		t_1	t_2
			$Y \leftarrow 42$;	
$R_1 \leftarrow X$;	$R_2 \leftarrow Y$;	$\rightarrow$	$R_1 \leftarrow X$;	$R_2 \leftarrow Y$;
$Y \leftarrow R_1$	$X \leftarrow R_2$		$Y \leftarrow R_1$	$X \leftarrow R_2$

Figure 3.10: Illegal program transformation.

Variable protection. Protecting all the accesses to shared variables by using mutual exclusion locks (Sec. 3.4) avoids these issues by enforcing memory barriers and locally disabling compiler optimisations: the (highly desirable) "*data-race-freedom*" property states that, in the absence of data-race, the semantics follows strictly sequential consistency. Additionally, Reynolds [Rey04] suggests that all unprotected accesses should be considered as fatal errors, so that a valid program only exhibits sequential consistent executions. These rules on the compiler and the program are attractive as they reduce the verification problem to checking that programs are correct in the sequentially consistent model plus checking that they do not have any data-race. However, we aim at checking programs with "benign" data-daces so that, in addition to detecting data-races, we must continue the analysis with a realistic semantics for them.

3.5.2 Formal model

We now propose a formal model of executions that takes into account a large class of non sequentially consistent behaviors.

Weakly-consistent models. Extensions of Lamport's sequentially consistent execution model, so called *weakly-consistent memory models*, have been studied originally for hardware. We refer the reader to [AG96] for a tutorial. Precise formal models of popular architectures are now available (for instance, the x86-TSO model by Sewell et al. [SSO$^+$10] formalizing Intel architectures). The use of weakly consistent memory models in programming language semantics, that additionally model the effect of optimizing compilation, were pioneered by Pugh [Pug99] and culminated in the Java memory model of Manson et al. [MPA05, GJSB05]. Models in this family are defined implicitly, as the solution of a complex process where each value read must be justified by a series of transformed execution traces. We choose instead a generative model based on an explicit set of local control path transformations, which is reminiscent of the approach by Saraswat et al. [SJMvP07]. It makes it easy to check whether a given compiler or processor obeys this model.

Control paths. To account for transformations that alter program block-structures, we start by converting programs into sets of linear *control paths*, which are sequences of assignments $X \leftarrow e$ and tests $e \bowtie 0$. The set of control paths $path(stat)$ in a statement $stat$ is defined by structural induction as:

$$\begin{aligned}
&path(X \leftarrow e) \stackrel{\text{def}}{=} \{\, X \leftarrow e \,\} \\
&path(s_1;\, s_2) \stackrel{\text{def}}{=} path(s_1) \cdot path(s_2) \\
&path(\textbf{if } e \bowtie 0 \textbf{ then } s \textbf{ endif}) \stackrel{\text{def}}{=} \\
&\qquad (\{e \bowtie 0\} \cdot path(s)) \cup \{e \not\bowtie 0\} \\
&path(\textbf{while } e \bowtie 0 \textbf{ do } s \textbf{ done}) \stackrel{\text{def}}{=} \\
&\qquad (\{e \bowtie 0\} \cdot path(s))^* \cdot \{e \not\bowtie 0\} \;.
\end{aligned} \tag{3.21}$$

Paths are all finite but, when *stat* contains a loop, $path(stat)$ is an infinite set. Note that, because $\mathbb{S}_{\mathcal{I}\!f}[\![\ stat\ ]\!]_t$ is a join-morphism, it is equal to the join over all control paths in *stat*:

$$\begin{array}{l}\mathbb{S}_{\mathcal{I}\!f}[\![\ stat\ ]\!]_t\mathcal{X} = \mathbb{\Pi}_{\mathcal{I}\!f}[\![\ path(stat)\ ]\!]_t\mathcal{X} \\ \text{where } \mathbb{\Pi}_{\mathcal{I}\!f}[\![\,\Pi\,]\!]_t\,\mathcal{X} \stackrel{\text{def}}{=} \bigsqcup_{s_1\cdot\ldots\cdot s_n\in\Pi} \mathbb{S}_{\mathcal{I}\!f}[\![\,s_1;\ldots;s_n\,]\!]_t\mathcal{X}\ .\end{array} \tag{3.22}$$

This fact, which is well-known for distributive data-flow analyses [Kil73], was proved for big-step semantics in [Min12d].

Path transformations. We now propose an example set of local *path transformations*, which we denote as $p \rightsquigarrow p'$:

1. reordering assignments: $X_1 \leftarrow e_1 \cdot X_2 \leftarrow e_2 \rightsquigarrow X_2 \leftarrow e_2 \cdot X_1 \leftarrow e_1$;
2. reordering tests: $e_1 \bowtie_1 0 \cdot e_2 \bowtie_2 0 \rightsquigarrow e_2 \bowtie_2 0 \cdot e_1 \bowtie_1 0$;
3. reordering tests before assignments: $X_1 \leftarrow e_1 \cdot e_2 \bowtie 0 \rightsquigarrow e_2 \bowtie 0 \cdot X_1 \leftarrow e_1$;
4. reordering assignments before tests: $e_1 \bowtie 0 \cdot X_2 \leftarrow e_2 \rightsquigarrow X_2 \leftarrow e_2 \cdot e_1 \bowtie 0$, when X_2 is local to the thread;
5. propagating assignments: $X \leftarrow e \cdot s \rightsquigarrow X \leftarrow e \cdot s[e/X]$, when variables in e are local to the thread and e is deterministic;
6. eliminating common sub-expressions: $s_1\cdot\ldots\cdot s_n \rightsquigarrow X \leftarrow e \cdot s_1[X/e] \cdot \ldots \cdot s_n[X/e]$, when X does not occur in the program.

These transformations are only valid under some conditions: assigned variables should not appear in other expressions, expressions must not block nor evaluate to an error, and modified statements should only involve assignments and tests (not synchronization statements).

These simple rules allow modeling large classes of classic program transformations as well as distributed memories. Store latency can be simulated using rules 5 and 1. Changing the atomicity of operations by breaking a statement into several ones is possible with rules 5 and 6. Rules 1–4 allow peephole optimization. Transformations that do not change the set of control paths, such as loop unrolling, are naturally supported. As a concrete example, the realistic x86-TSO model [SSO$^+$10] can be entirely simulated with these transformations, and so, an analysis sound for our model will also be sound for x86-TSO (but it may include behaviors not allowed by x86-TSO, which results in a loss of precision). The rules, however, do not allow "out-of-thin air" transformations (Fig. 3.10). This list is not exhaustive; we refer the reader to [Min11] for more examples.

Program semantics. We then close the $\rightsquigarrow$ relation by context ($p \rightsquigarrow p' \implies a\cdot p\cdot b \rightsquigarrow a\cdot p'\cdot b$), transitivity ($p_1 \rightsquigarrow p_2 \wedge p_2 \rightsquigarrow p_3 \implies p_1 \rightsquigarrow p_3$), and reflexivity ($p \rightsquigarrow p$), and state that the set of paths Π' is a valid transformation of *stat* if it contains all its path, possibly transformed: $\forall p \in path(stat) : \exists p' \in \Pi' : p \rightsquigarrow p'$. The semantics of the transformed program is then simply the join over all paths in Π': $\mathbb{\Pi}_{\mathcal{I}\!f}[\![\,\Pi'\,]\!]_t$.

We proved in [Min12d] that transformed programs do not exhibit more errors nor interferences than the original one:

Theorem 3.5.1.
$\forall\mathcal{X} : \left[\mathbb{\Pi}_{\mathcal{I}\!f}[\![\,\Pi'\,]\!]_t\mathcal{X}\right]_{\Omega,\mathcal{I}\!f} \sqsubseteq \left[\mathbb{\Pi}_{\mathcal{I}\!f}[\![\ path(stat)\ ]\!]_t\mathcal{X}\right]_{\Omega,\mathcal{I}\!f}$.

t_1	t_2
while $0 = 0$ **do**	**while** $0 = 0$ **do**
lock(m);	**lock**(m);
if $X > 0$ **then**	**if** $X < 10$ **then**
$X \leftarrow X - 1$;	$X \leftarrow X + 1$;
$Y \leftarrow Y - 1$	$Y \leftarrow Y + 1$
endif;	**endif**;
unlock(m)	**unlock**(m)
done	**done**

Figure 3.11: Imprecisely analyzed program due to the lack of relational interferences.

Hence, our interference analysis is sound with respect to transformed programs. A result similar to Thm. 3.5.1 was found simultaneously by Alglave et al. [AKL$^+$11].

Limitations. Our set of allowed transformations is not exhaustive; it would be interesting to characterize more precisely under which transformations Thm. 3.5.1 holds. Moreover, it is also possible to change our interference-semantics so that it holds under more transformations. For instance, our framework imposes atomic memory writes, but this restriction can be lifted by generating interferences that expose partially assigned values. Dually, it would be interesting to restrict our model to a less permissive one (such as x86-TSO [SSO$^+$10]), and then define a more precise interference semantics that benefits from the restricted set of possible transformations. Note that our choice of an interference semantics was not initially motivated by the modeling of weakly consistent memories (although this is an important side effect), but rather by the construction of an effective and efficient static analyzer.

3.6 Discussion

In this chapter, we have constructed a big-step interference-based thread-modular static analysis by abstracting a semantics expressed in rely-guarantee form. Although the original rely-guarantee semantics is complete, one of our first step to construct an effective analysis was, in Sec. 3.3, to abstract interferences in an incomplete flow-insensitive and non-relational way. Our experimental results, which we will detail in Sec. 6.3, show that such an analysis has nevertheless a good precision; yet, it is not sufficient to prove completely the absence of runtime error in the analyzed codes. We now show, on small program fragments, examples of imprecise analyses due to our initial abstraction of interferences.

Example 3.6.1. Consider the program in Fig. 3.11. It is similar to the producer/consumer example of Fig. 3.6, but additionally maintains a copy of the resource count in Y. Our analysis finds, as in Ex. 3.4.1, that $X \in [0, 10]$. However, it finds no bound on Y. In order to prove that $Y \in [0, 10]$, we would need to infer that $X = Y$ holds at lock boundaries, which would thus require a relational abstraction of well synchronized interferences. This example is also related to Ex. 2.4.2 which motivated the need for relational domains in (sequential) program analyses.
End of example.

Example 3.6.2. Figure 3.12 presents a slightly more complex variant of Ex. 3.2.2. Here, each thread increments and decrements X in a loop, once per loop iteration. As no more than

t_1	t_2
while $0 = 0$ **do**	**while** $0 = 0$ **do**
lock(m);	**lock**(m);
$^{\ell 1}X \leftarrow X + 1$;	$^{\ell 3}X \leftarrow X + 1$;
unlock(m);	**unlock**(m);
lock(m);	**lock**(m);
$^{\ell 2}X \leftarrow X - 1$;	$^{\ell 4}X \leftarrow X - 1$;
unlock(m)	**unlock**(m)
done	**done**

Figure 3.12: Imprecisely analyzed program due to the lack of flow-sensitive interferences.

two incrementations occur without a decrementation in any interleaving of thread instructions, and no more than two decrementations without an incrementation, we have $X \in [-2, 2]$. Our analysis cannot infer such a complex property and finds no bound on X. One solution would be to refine the analysis with flow-sensitive information by exploiting auxiliary variables. Indeed, we could express that, when the thread t_1 is at location $\ell 1$ and the thread t_2 is at location $\ell 3$, then a thread interference can only increment X from 0 to 1, and similarly at other locations. This example shows the importance of limiting the interferences to only the transitions appearing in actual program traces; the full transition system of the program would instead state that X can be incremented from c to $c + 1$ for any value c.
End of example.

Chronologically, we first proposed, in [Min11], the analysis with flow-insensitive and non-relational interferences described in Sec. 3.3, and only later [Min12c] re-formalized it as an abstraction of a complete rely-guarantee semantics (Sec. 3.2). A natural future work consists in developing further the connection with rely-guarantee, developing interference abstractions that are, at least partially, flow-sensitive and relational, and incorporating them into our generic big-step analyzer construction.

Related work. There exists a large literature on the use of formal methods to verify parallel programs; we can only present here a shallow overview and present mainly recent results. For further information, we refer the reader to the comprehensive, if dated, survey by Rinard [Rin01].

We already mentioned proof methods, as our method is inspired from Jones' popular rely-guarantee method [Jon81]. We refer the reader to [dRdBH$^+$01] for a survey of such techniques. The connection between proof methods and abstract interpretation has not been much investigated since the work by Cousot and Cousot in [CC84, Cou85], with the notable exception of Malkis [Mal10]; all these works focus on Owicki–Gries–Lamport methods.

Model checking also has a long history of verifying parallel systems, including recently on weak memory models (for instance in [ABBM10]). The state explosion problem, that plagues explicit-state model checking methods, is particularly acute on concurrent programs due to the larger amount of states and interleavings to consider. Some solutions have been proposed, such as symbolic model checking [McM93], which is a general model checking method, or partial order reduction methods [God94], which target specifically concurrent programs. Due to the emphasis on completeness, these methods remain costly. Another way to address the state explosion problem, bounded model checking [BCCZ99], consists in performing a partial exploration. A variant proposed in the context of concurrent programing is context-bounded model checking [QR05]. These methods are not sound as they may miss errors. By contrast, we abstract the problem sufficiently so that no interleaving needs to be considered, at the cost of completeness, while never sacrificing soundness.

We now focus on related work in static analysis. Fully flow-insensitive analyses (such as Steensgaard's popular points-to analysis [Ste96]) can be used as-is on concurrent programs, as they consider arbitrary interleavings of all program instructions, but their precision is not sufficient for program verification. We are aware of a few static analyses that treat threads in a flow-sensitive way, as we do. They use, similarly to us, a notion of interference and achieve thread-modularity. One example is the pointer and escape analysis for Java by Sălcianu and Rinard [SR01], where interferences also model method calls. Another one is the recent static analysis of C programs with POSIX Threads by Carré and Hymans [CH09], with a slightly different focus as it includes dynamic thread creation but not synchronization mechanisms. Static analysis in weak consistency models has also gathered recent attention: we can cite Ferrara's work [Fer08] on the Java memory model, and the "repair-loop" technique by Alglave and al. [AMSS11] which resembles our interference fixpoint. Although these methods handle each thread in a flow-sensitive way, their interactions are abstracted, similarly to our analysis, in a flow-insensitive way. Goubault et al. propose a different kind of analysis [GH05] based on geometric principles in order to abstract thread interactions in a flow-sensitive way. This abstraction focuses on locks and synchronization properties (deadlocks and mutual exclusion); it is not thread-modular and considers only a finite number of program steps.

Chapter 4

Affine abstractions

By construction, static analyses by abstract interpretation are parametrized by a choice of abstract domains and, in particular, numeric domains able to abstract the numeric computations that are pervasive in computer programs. There exists a growing library of numeric abstract domains, but the need exists always to design new ones: either to infer new classes of properties, or to explore new trade-offs between cost and precision, or even to propose new algorithms to handle well-known classes of abstract properties. We present, in this chapter and the next, a few novel domains. These domains are not specific to the analysis of concurrent programs and, while they can indeed be used as parameters in the analysis construction from Chap. 3 and some of them are effectively integrated in our AstréeA prototype analyzer (Sec. 6.2), they are of general use. The present chapter focuses on variations of the polyhedra domain and presents more fundamental constructions, while the next chapter constructs more pragmatic domains geared towards specific applications, namely the analysis of C datatypes as considered in the Astrée and AstréeA analyzers.

Since its introduction by Cousot and Halbwachs in the late 1970s [CH78], the polyhedra abstract domain has been widely used in static analysis. However, its underlying algorithmic, based on the double description method and Chernikova's algorithm on arbitrary precision rational coefficients, has remained largely unchanged, until the mid 2000s when Simon and King proposed to switch to a constraint-only representation [SK05].

Together with Patrick Cousot, we suggested to Liqian Chen, then a PhD student of Ji Wang at the National University of Defense Technology (Changsha, China) visiting the ENS, to further advance the design of polyhedral domains. This chapter reports on the results we achieved; it is a collaborative work with Liqian Chen, Patrick Cousot, and Ji Wang.

On the semantic level, our work consists in changing the nature of the coefficients appearing in the affine constraints: we replace arbitrary precision rationals with floating-point numbers (Sec. 4.1) and with intervals with rational or float bounds (Sec. 4.2). We thus study restrictions and extensions of the expressiveness of polyhedra. On the algorithmic level, changing the nature of coefficients radically changes the way abstract operations are performed. These changes require us to enrich the classic polyhedra algorithms with new ones, often borrowing from recent results in constraint programming and mathematical programming: we use in particular guaranteed linear programming [NS04], interval linear programming [CR00], and solvers for linear complementary problems [MP95].

Our results have been published as conference articles, as well as in Liqian Chen's PhD [CMC08, CMWC09, CMWC10, CMWC11, Che10]. Moreover, the domains have been implemented as prototypes and tested in the Apron library, a general framework for numeric abstract domains, which we describe in Sec. 6.1.

4.1 Floating-point polyhedra

Our first work consists in exploring the use of floating-point numbers in order to improve the scalability of polyhedra.

4.1.1 Motivation

Classic polyhedra libraries following the early work by Cousot and Halbwachs [CH78] (such as Apron [JM09]) scale up to only a few variables: an experimental study conducted by Duong in his PhD [NQ10] reports a significant number of time outs and out of memory errors on polyhedra, starting from as few as seven dimensions. A first issue, the explosion of the number of generators output by Chernikova's algorithm, can be avoided by abandoning the double description method and using only constraints [SK05]. Another cause of inefficiency is the use of exact rational arithmetic: this may cause coefficients to grow up to an unbounded size in theory. In practice, exponential blow-ups are not uncommon, even for programs featuring only variables with a small range, as observed in [NQ10].

A simple and practical solution consists in discarding constraints when their coefficients grow too large (e.g., when a numerator or denominator cannot be represented in a machine integer), thus trading precision for efficiency. It is always sound to discard constraints, but may result in missed properties. As machine integers, floating-point numbers benefit from a constant memory and fast, hardware-assisted operations, but additionally allow representing a much larger range of values. Rounding errors will result in a gradual loss of precision in constraints, which is more gentle than abruptly removing them. The main challenge is to ensure that, despite rounding errors, the domain stays sound, i.e., rounding can only enlarge polyhedra. We stress on the fact that simply replacing rationals with floats in existing algorithms does not result in a sound outcome.

4.1.2 Representation

We build on the constraint-only presentation of polyhedra from Simon et al. [SK05] recalled in Sec. 2.4.2, but use floating-point coefficients in $\mathbb{F}$ (Sec. 2.4.4). Hence, a *floating-point polyhedron* on n variables is represented as a pair $\langle \mathbf{A}, \vec{B} \rangle$ composed of a matrix $\mathbf{A} \in \mathbb{F}^{m \times n}$ and a vector $\vec{B} \in \mathbb{F}^m$. The polyhedron still represents a set of real points in the vector space $\mathbb{R}^n$, and its concretization γ_p is unchanged: $\gamma_p(\langle \mathbf{A}, \vec{B} \rangle) \stackrel{\text{def}}{=} \{ \vec{V} \in \mathbb{R}^n \mid \mathbf{A} \times \vec{V} \leq \vec{B} \}$, where $\mathbf{A} \times \vec{V}$ is evaluated using real arithmetic. As

before, we also denote polyhedra as sets of affine constraints $\mathcal{C} = \{ \sum_{i=1}^{n} A_{1i}V_i \leq B_1, \ldots, \sum_{i=1}^{n} A_{mi}V_i \leq B_m \}$ when more convenient.

Arithmetic. Following Sec. 2.4.4, we distinguish exact operations on reals from float operations with rounding by using plain operators $+$, $-$, $\times$, / for the former and circled ones $\oplus_r$, $\ominus_r$, $\otimes_r$, $\oslash_r$ for the letter, tagged with a rounding direction $r \in \{+\infty, -\infty\}$. We also use interval arithmetic with float bounds: $\oplus_i^\sharp$, $\ominus_i^\sharp$, $\otimes_i^\sharp$, $\oslash_i^\sharp$ (2.21), which we extend to operations on vectors, matrices, affine expressions, and affine constraints whose coefficients are intervals with float bounds. Likewise, $\odot_i^\sharp$ denotes the dot product of two vectors of float intervals. As float arithmetic and, by extension, float interval arithmetic, does not enjoy the distributivity and associativity of reals, the ordering of additions and multiplications matters. In our case, we do not impose any order and note simply that the results obtained with different orders, while possibly different, are all sound.

4.1.3 Core algorithms

Recall from Sec. 2.4.2 that the constraint-only presentation of the polyhedra domain relies on two main algorithms: linear programming and projection. We show how to adapt soundly these two algorithms using float operations only.

Linear programming. Recall that solving the linear programming problem LP (2.12) given a polyhedron $\langle \mathbf{A}, \vec{B} \rangle$ and a vector $\vec{C}$ consists in computing:

$$LP(\langle \mathbf{A}, \vec{B} \rangle, \vec{C}) \stackrel{\text{def}}{=} \min \{ \vec{C} \cdot \vec{V} \mid \mathbf{A} \times \vec{V} \leq \vec{B} \} \ .$$

Generally $LP(\langle \mathbf{A}, \vec{B} \rangle, \vec{C})$ is not representable as a float, even if both $\langle \mathbf{A}, \vec{B} \rangle$ and $\vec{C}$ are, and we will settle for upper and lower bounds. We consider here only the problem of computing a lower bound in $\mathbb{F}$, denoted as $LP_{\mathbb{F}}(\langle \mathbf{A}, \vec{B} \rangle, \vec{C})$, i.e. we require:

$$\forall \vec{V} : \mathbf{A} \times \vec{V} \leq \vec{B} \implies \vec{C} \cdot \vec{V} \geq LP_{\mathbb{F}}(\langle \mathbf{A}, \vec{B} \rangle, \vec{C})$$

being understood that computing an upper bound is similar.

It is interesting to note that most linear programming implementations compute with floats, for the sake of efficiency, and thus output an approximation of the result. This includes modern interior point methods [Kar84] which proceed by successive approximations, but also many implementations of the Simplex algorithm [Sch86] (although an exact Simplex implementations based on arbitrary precision rational arithmetic is possible). There is no guarantee that the computed approximate result is a lower bound. When an exact result is required, a "*purification scheme*" is often employed to construct it from an approximate one. For instance, in the case of Simplex, the algorithm explores bases of the constraint system (i.e., subsets of n constraints) to find an optimal feasible solution. Thus, one method (used for instance in [SSM05]) is to perform most of the search using floats, which outputs a basis that may not be optimal nor feasible, and then bootstrap an exact Simplex solver using arbitrary precision rationals with this basis, in the hope that only a few extra exploration steps are necessary.

Purification methods are not adequate for us as we are more interested in efficiency than in exactness, and we wish to perform the entire algorithm using solely floats. We thus use recent advances in the field of *rigorous linear programming*: we use a method by Neumaier and Shcherbina [NS04] that consists in post-processing the approximate result into a lower approximation. More precisely, the method starts with an approximate solution of the dual problem:

$$LP^*(\langle \mathbf{A}, \vec{B} \rangle, \vec{C}) \stackrel{\text{def}}{=} \max \{ \vec{B} \cdot \vec{W} \mid \mathbf{A}^t \times \vec{W} = \vec{C} \wedge \vec{W} \leq \vec{0} \} \tag{4.1}$$

given as a vector $\vec{W}$ that approximates the optimum. We consider now the vector $\vec{r}$ that evaluates "how far" $\vec{W}$ is from actually satisfying the dual constraint system $\mathbf{A}^t \times \vec{W} = \vec{C}$:

$$\vec{r} \stackrel{\text{def}}{=} \mathbf{A}^t \times \vec{W} - \vec{C} \ .$$

Sound bounds for $\vec{r}$ can be computed using interval arithmetic:

$$[\underline{\vec{r}}, \overline{\vec{r}}] \stackrel{\text{def}}{=} \mathbf{A}^t \otimes_i^\sharp \vec{W} \ominus_i^\sharp \vec{C} \ .$$

Finally, we assume that we are given a bounding box $[\underline{\vec{V}}, \overline{\vec{V}}]$ of the polyhedron in the form of a lower and an upper bound vector, so that we know that the (exact) optimal solution $\vec{V}$ of the primal linear programming problem satisfies: $\underline{\vec{V}} \leq \vec{V} \leq \overline{\vec{V}}$. As $\vec{W} \leq \vec{0}$ and $\mathbf{A} \times \vec{V} \leq \vec{B}$, we have $\vec{W}^t \times \mathbf{A} \times \vec{V} \geq \overline{\vec{W}} \cdot \vec{B}$. Thus, $\vec{V} \cdot \vec{C} = \vec{V} \cdot (\mathbf{A}^t \times \vec{W} - \vec{r}) = \vec{W}^t \times \mathbf{A} \times \vec{V} - \vec{r} \cdot \vec{V} \geq \vec{W} \cdot \vec{B} - \vec{r} \cdot \vec{V}$. Hence, a lower bound of $LP(\langle \mathbf{A}, \vec{B} \rangle, \vec{C})$ can be computed by interval arithmetic, using our interval approximations of $\vec{r}$ and the bounding box $[\underline{\vec{V}}, \overline{\vec{V}}]$:

$$LP_{\mathbb{F}}(\langle \mathbf{A}, \vec{B} \rangle, \vec{C}) \stackrel{\text{def}}{=} \min(\vec{W} \odot_i^\sharp \vec{B} \ominus_i^\sharp [\underline{\vec{r}}, \overline{\vec{r}}] \odot_i^\sharp [\underline{\vec{V}}, \overline{\vec{V}}]). \tag{4.2}$$

Note that our interval computations are performed using the float intervals domain from Sec. 2.4.4, hence the computation is performed using solely floats.

Fourier–Motzkin's elimination. Projecting (or eliminating) a variable V_k on a set of constraints $\mathcal{C}$ can be performed by Fourier–Motzkin's elimination algorithm $FM(\mathcal{C}, V_k)$ (2.14). It consists in combining all possible pairs of constraints where the coefficients of V_k have opposite signs (and keeping constraints where V_k does not appear). More precisely, given $c^+ \stackrel{\text{def}}{=} (\vec{A}^+ \cdot \vec{V} \leq b^+) \in \mathcal{C}$ and $c^- \stackrel{\text{def}}{=} (\vec{A}^- \cdot \vec{V} \leq b^-) \in \mathcal{C}$ such that $A_k^+ > 0$ and $A_k^- < 0$, we add the constraint: $c \stackrel{\text{def}}{=} A_k^+ c^- + (-A_k^-) c^+$. Unfortunately, c is generally not representable in floats. Our solution is to combine c^+ and c^- using float interval arithmetic. Note however that, when computing $(A_k^+ \otimes_i^\sharp c^-) \oplus_i^\sharp ((-A_k^-) \otimes_i^\sharp c^+)$, the coefficient of V_k may be an interval not reduced to zero in the result, due to rounding errors; hence, V_k is not eliminated. Instead of combining c^+ and c^- by weighted addition, we combine them by simple addition after normalizing the coefficient of V_k to 1 by division:

$$c \stackrel{\text{def}}{=} (c^+ \oslash_i^\sharp A_k^+) \oplus_i^\sharp (c^- \oslash_i^\sharp (\ominus_i^\sharp A_k^-)) \ . \tag{4.3}$$

We note that the (interval) coefficient of V_k in c is given by the formula: $(A_k^+ \oslash_i^\sharp A_k^+) \oplus_i^\sharp (A_k^- \oslash_i^\sharp (\ominus_i^\sharp A_k^-))$, which evaluates to $[0, 0]$ as self-divisions as well as adding 1 to -1 are all exact operations in float. We have effectively eliminated V_k. Finally, we use the scalar linearization slin (2.17) to replace the interval coefficients of variables with scalar ones, yielding a constraint of the form $\sum_{j \neq k} A_j V_j \leq [b, c]$, which is equivalent to $\sum_{j \neq k} A_j V_j \leq c$. The resulting constraint is affine, does not feature V_k, and is sound in the sense that it is implied by c^+ and c^-.

We denote as $FM_{\mathbb{F}}(\mathcal{C}, V_k)$ the outcome after applying this method to each pair of original constraints c^+ and c^- in $\mathcal{C}$. Then, $FM_{\mathbb{F}}(\mathcal{C}, V_k)$ over-approximates $FM(\mathcal{C}, V_k)$, and can be computed using only floats.

4.1.4 Abstract operators

We now review the polyhedra abstract operators from Sec. 2.4.2 and show that, despite the approximations in $FM_{\mathbb{F}}$ and $LP_{\mathbb{F}}$, they are sound.

Comparison. Firstly, float linear programming can be used to check for polyhedra inclusion $\sqsubseteq_p^\sharp$. Indeed, substituting $LP_{\mathbb{F}}$ for LP in (2.13) gives:

$$\mathcal{C} \sqsubseteq_p^\sharp \{\, \vec{A} \cdot \vec{V} \leq b \,\} \Longleftarrow LP_{\mathbb{F}}(\mathcal{C}, -\vec{A}) + b \geq 0$$
$$\mathcal{C}_1 \sqsubseteq_p^\sharp \mathcal{C}_2 \iff^{\text{def}} \forall c \in \mathcal{C}_2 : \mathcal{C}_1 \sqsubseteq_p^\sharp \{c\}$$

Note that, because $LP_{\mathbb{F}}$ only computes a lower bound, we do not have the equivalence: it can be used to prove that a polyhedron definitively entails a constraint (and so, that a polyhedron is definitely included in another), but not that it does not entail it, making our abstract inclusion a sound semi-test for $\sqsubseteq_p^\sharp$. Likewise, entailment can be used to remove redundant constraints, whereby the approximation causes only constraints that are actually redundant to be removed but may fail to remove some redundant ones.

Tests and assignments. Tests and assignments are handled as in the case of rational polyhedra (Sec. 2.4.2). Affine tests $\mathbb{S}_p^\sharp[\![\, \vec{A} \cdot \vec{V} + b \leq 0 \,]\!]\mathcal{C}$ are handled by simply adding a constraint to $\mathcal{C}$, which remains an exact abstraction. The nondeterministic assignment $\mathbb{S}_p^\sharp[\![\, V_k \leftarrow [-\infty, +\infty] \,]\!]$ is modeled by a projection $FM_{\mathbb{F}}(\mathcal{C}, V_k)$ which, unlike the rational projection $FM(\mathcal{C}, V_k)$, may incur a slight loss of precision. Then, arbitrary affine assignments can be reduced, as before, to tests and projections using a temporary variable $\mathbb{S}_p^\sharp[\![\, V_k \leftarrow \vec{A} \cdot \vec{V} + b \,]\!] \stackrel{\text{def}}{=} [V_{n+1}/V_k] \circ \mathbb{S}_p^\sharp[\![\, V_k \leftarrow [-\infty, +\infty] \,]\!] \circ \mathbb{S}_p^\sharp[\![\, V_{n+1} - \vec{A} \cdot \vec{V} - b = 0 \,]\!]$. This operator is no longer exact because the projection is not exact.

Join and widening. As shown in (2.15), computing a convex hull $\mathcal{C}_1 \cup_p^\sharp \mathcal{C}_2$ can be reduced to projecting some variables; we can thus approximate it through $FM_{\mathbb{F}}$. We observed in [CMC08] that, due to over-approximations, the result sometimes fails to include some constraints from one polyhedron which are entailed by the other one, and are thus obviously satisfied by the join. To solve this imprecision, we tighten the resulting polyhedron by adding any constraint $c \in \mathcal{C}_1 \cup \mathcal{C}_2$ such that $\mathcal{C}_1 \sqsubseteq_p^\sharp \{c\} \wedge \mathcal{C}_2 \sqsubseteq_p^\sharp \{c\}$ can be proved using $LP_{\mathbb{F}}$. Our widening $\mathcal{C}_1 \triangledown_p \mathcal{C}_2$ simply keeps the constraints in $\mathcal{C}_1$ that are entailed by $\mathcal{C}_2$.[1]

Bounding box. Note that our definition of $LP_{\mathbb{F}}$ implicitly assumes that a bounding box of the polyhedron is available. This is actually also the case for $FM_{\mathbb{F}}$ (it is needed for the scalar linearization slin that gets rids of interval coefficients). In practice, it is useful to maintain such a bounding box at all time. Sometimes, the bounding box of the result of an operation can be computed solely based on the bounding boxes of the arguments (this is the case for the convex hull, for instance). When this is not the case, the bounding box can be recovered by applying $LP_{\mathbb{F}}$ on the basis vectors $\vec{e}_i$.

[1] The refined widening from Fig. 2.12, which also considers constraints in $\mathcal{C}_2$, cannot be used as it is well-defined only for completely non-redundant polyhedra, and this cannot be ensured using our approximate $LP_{\mathbb{F}}$.

while $0 = 0$ **do**
  $X \leftarrow [-128, 128]$;
  $D \leftarrow [1, 16]$;
  $S \leftarrow Y$;
  $R \leftarrow X \ominus_r S$;
  $Y \leftarrow X$;
  if $R \oplus_r D \leq 0$ **then** $Y \leftarrow S \ominus_r D$ **endif**;
  if $D \ominus_r R \leq 0$ **then** $Y \leftarrow S \oplus_r D$ **endif**
done

Figure 4.1: Floating-point rate limiter.

4.1.5 Experimental results

A proof-of-concept implementation was designed by L. Chen and interfaced with Apron, a general library of numeric abstract domains (Sec. 6.1). The implementation uses the GLPK floating-point simplex library [Mak00], on top of which the rigorous linear programming algorithm $LP_{\mathbb{F}}$ is constructed.

The domain was tested on a few simple examples and compared to NewPolka, Apron's built-in polyhedra library that uses the double description method and arbitrary-precision rationals. Experiments were conducted using the Interproc static analyzer [LAJ11] bundled with Apron; it analyzes simple programs in a toy numeric language. We refer the interested reader to [CMC08] for the detailed experiments and only reproduce here a synthesis of the results. A first test considered the analysis of the integer program examples from Apron (such as: factorial, bubble sort, heap sort, Ackermann's function); our domain inferred the exact same invariants as NewPolka but performed less efficiently (up to five times slower, with an average analysis time of 38ms). However, a second test based on small floating-point programs showed that our domain performed more efficiently (up to ten times faster, with an average analysis time of 195ms) and found similar invariants (up to rounding of coefficients). These analyses use the floating-point linearization of Sec. 2.4.4 to soundly model float operations in the program. An example of such analysis is given below:

Example 4.1.1. Consider the program in Fig. 4.1, which is extracted from [Min06b, CMC08] and inspired from an actual program. It implements a rate limiter: at each loop iteration, it fetches an input X in $[-128, 128]$ from a sensor and a maximal slope D in $[1, 16]$, and computes an output value in Y that tries to follow X but is limited to change at maximal rate D (i.e., $|S - Y| \leq D$ where S is the last value output). Our analysis finds, as output bound: $|Y| \leq 128.000047684$, which is actually optimal assuming a worse-case rounding. This example requires relational information and is thus out of the reach of the interval domain.
End of example.

The results are encouraging for an early implementation. In particular, the domain really shines when it comes to analyzing programs featuring floating point numbers. The reason

is that modeling float operations with exact rationals quickly results in large coefficients and becomes impractical, while float polyhedra are immune to this problem.

4.1.6 Discussion

This section has presented a polyhedra abstract domain programmed purely in floats, with an emphasis on ensuring its soundness, which is a prime requirement for program validation. It also has an interesting theoretical significance: it provides the first sound implementation of a relational analysis for float programs implemented fully with floats.

We now discuss briefly two other aspects: efficiency and precision.

As a result of rounding errors, most operations that were exact or optimal on polyhedra with rational arithmetic are no longer exact nor optimal. In practice, our domain includes several heuristics to limit the precision loss, including: bound tightening using propagation algorithms, a reduced product with intervals (which are less prone to rounding errors as they use simpler algorithms), and a careful implementation of the slin operator (e.g., by replacing an interval with a suitably rounded value instead of its midpoint); we refer the interested reader to [CMC08] for more information. An important remark, however, is that programmers expect float programs to suffer from computation drift due to rounding, and generally include bound checks as a safety measure. Such checks are abstracted exactly and tremendously help the analysis, compensating for the drift in the abstract semantics as well as in the concrete one. We believe that, when analyzing programs written in such a defensive way, rounding errors in the analyzer do not significantly degrade the result of the analysis.

The main bottleneck in efficiency is the large number of calls to the linear programming algorithm, in particular triggered by the need to remove the large amount of redundant constraints generated by Fourier–Motzkin's eliminations in joins. This problem was already observed for rational polyhedra based on constraints [SK05]. While [SK05, HLL92, Imb93] propose some solutions, which can be directly applied to our float domain, they are not sufficient to scale up. We believe that more work on constraint-only polyhedra is required in this direction to improve the scalability.

4.2 Interval polyhedra

Our second work stemmed from the first one, by observing the importance of intervals when abstracting floats or abstracting with floats. For instance, the value of a real expression cannot always be represented as a float, but it can always be enclosed in a float interval. Moreover, affine forms with interval coefficients play an important role in modeling float expressions (Sec. 2.4.4). They also appear internally in our floating-point Fourier–Motzkin's elimination (Sec. 4.1.3), only to be removed by slin. This leads naturally to the design of polyhedra domains where coefficients can also be intervals. A first construction, in Sec. 4.2.1, arises naturally from that of the preceding section by allowing intervals with float bounds as coefficients instead of plain float coefficients. A second one, in Sec. 4.2.2, returns to exact rationals and a double description method, while keeping interval coefficients. A third one, in Sec. 4.2.3, consists in restricting the expressiveness of interval polyhedra to interval affine equalities.

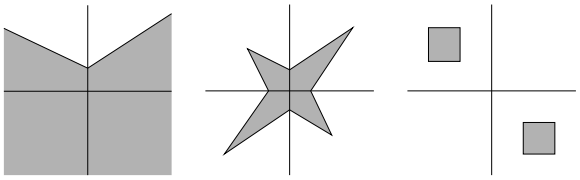


Figure 4.2: Interval polyhedra examples.

4.2.1 Float interval polyhedra

Representation. We extend polyhedra to represent *affine interval constraints*, of the form: $\sum_j [\underline{a}_j, \overline{a}_j] V_j \leq b_j$. A constraint is satisfied by a vector $\vec{V}$ if $\sum_j a_j V_j \leq b_j$ holds for some choice of $a_j \in [\underline{a}_j, \overline{a}_j]$. A *float interval polyhedron* is then defined by a matrix of intervals with float bounds, conveniently represented as a matrix $\underline{\mathbf{A}} \in \mathbb{F}^{m\times n}$ of lower bounds and a matrix $\overline{\mathbf{A}} \in \mathbb{F}^{m\times n}$ of upper bounds, and by a vector of floats $\vec{B} \in \mathbb{F}^m$. Then $\langle [\underline{\mathbf{A}}, \overline{\mathbf{A}}], \vec{B} \rangle$ represents (extending $\leq$ element-wise):

$$\begin{aligned}&\gamma_{ip}(\langle [\underline{\mathbf{A}}, \overline{\mathbf{A}}], \vec{B} \rangle) \stackrel{\text{def}}{=} \bigcup \{ \gamma_p(\langle \mathbf{A}, \vec{B} \rangle) \mid \underline{\mathbf{A}} \leq \mathbf{A} \leq \overline{\mathbf{A}} \} \\ &= \{ \vec{V} \in \mathbb{R}^n \mid \exists \mathbf{A} \in \mathbb{R}^{m\times n} : \underline{\mathbf{A}} \leq \mathbf{A} \leq \overline{\mathbf{A}} \wedge \mathbf{A} \times \vec{V} \leq \vec{B} \} \ .\end{aligned} \tag{4.4}$$

Interval polyhedra can represent all the classic (float) polyhedra, and are actually much more expressive. In particular, they can represent non-convex and even unconnected sets, as illustrated in Fig. 4.2 and the example below.

Example 4.2.1. Figure 4.2.(a) shows the set of points satisfying the constraint $[-1, 1]x + 2y \leq 2$. When $x \geq 0$, the constraint reduces to $-x+2y \leq 2$ while, when $x \leq 0$, it reduces to $x+2y \leq 2$; hence, the result is not convex. Figure 4.2.(b) is defined by: $[-1, 1]x + 2y = [-2, 2] \wedge 2x + [-2, 1]y = [-2, 2]$, i.e., the conjunction of four constraints similar to that of Fig. 4.2.(a). Finally, 4.2.(c) is generated by: $[-1, 1]x = [-1, 1]y = 1 \wedge x, y \in [-2, 2] \wedge x + y \in [-1, 1]$ and is unconnected.
End of example.

An important remark is that, when the sign of each variable is fixed, an interval affine constraint $\sum_j [\underline{a}_j, \overline{a}_j] V_j \leq b_j$ can be reduced to an affine constraint using one bound from each interval, i.e., $\sum_j a_j V_j \leq b_j$ where $\forall j : a_j \in \{\underline{a}_j, \overline{a}_j\}$. As a consequence, in each *orthan*, an interval polyhedron gives a regular convex polyhedron. However, unlike disjunctive completions [CC79b], not all finite disjunctions of polyhedra are interval polyhedra. As we show shortly, the special form of interval polyhedra allows deriving more efficient algorithms than for arbitrary disjunctions.

Interval linear programming. Given an interval polyhedron $\langle [\underline{\mathbf{A}}, \overline{\mathbf{A}}], \vec{B} \rangle$ and a vector $\vec{C}$, the *interval linear programming* problem generalizes linear programming (2.12) as follows:

$$\begin{aligned}&ILP(\langle [\underline{\mathbf{A}}, \overline{\mathbf{A}}], \vec{B} \rangle, \vec{C}) \stackrel{\text{def}}{=} \\ &\quad \min \{ \vec{C} \cdot \vec{V} \mid \underline{\mathbf{A}} \leq \mathbf{A} \leq \overline{\mathbf{A}} \wedge \mathbf{A} \times \vec{V} \leq \vec{B} \} \ .\end{aligned} \tag{4.5}$$

From a theoretical point of view, interval linear programming is much harder than linear programming (the former is NP-complete [Roh06] while the later is polynomial). However,

techniques that perform well in practice have been proposed recently, including smart orthan enumeration methods avoiding the need to solve exponentially many linear programing problems [CR00], or iterative methods [Jan04]. These methods can be adapted to compute a lower bound of the optimum (4.5) using only floats. We refer the reader to [CR00, Jan04, Roh06] for more information on the relevant algorithms and will not discuss them here.

Given such a lower approximation, it becomes possible to check constraint entailment, polyhedra inclusion, and remove redundant constraints in a sound way, as in Sec. 4.1.4.

Projection. To implement variable elimination, used in the semantics of assignments, a simple idea is to adapt Fourier–Motzkin's algorithm to affine interval constraints. It is sufficient to explain how, given a variable V_k and two constraints $c^+ \stackrel{\text{def}}{=} (\sum_j [\underline{a}_j^+, \overline{a}_j^+] V_j \leq b^+)$ and $c^- \stackrel{\text{def}}{=} (\sum_j [\underline{a}_j^-, \overline{a}_j^-] V_j \leq b^-)$, we can combine c^+ and c^- to derive a new constraint implied by them where V_k does not occur. As in Fourier–Motzkin, we only consider pairs of constraints where the coefficient of V_k has a different sign: $\underline{a}_k^+ > 0$ and $\overline{a}_k^- < 0$. An important remark is that it is possible to ensure that the coefficient of V_k is exactly 1 in c^+ and -1 in c^-, by dividing, with interval arithmetic, the constraints by, respectively, $[\underline{a}_k^+, \overline{a}_k^+]$ and $[-\overline{a}_k^-, -\underline{a}_k^-]$. Indeed, we have for c^+ (the result is similar for c^-):

$$\begin{array}{ll} & \sum_j [\underline{a}_j^+, \overline{a}_j^+] V_j \leq b^+ \\ \iff & \exists a \in [\underline{a}_k^+, \overline{a}_k^+] : a V_k + \sum_{j \neq k} [\underline{a}_j^+, \overline{a}_j^+] V_j \leq b^+ \\ \iff & \exists a \in [\underline{a}_k^+, \overline{a}_k^+] : V_k + \sum_{j \neq k} ([\underline{a}_j^+, \overline{a}_j^+]/a) V_j \leq b^+/a \\ \implies & V_k + \sum_{j \neq k} ([\underline{a}_j^+, \overline{a}_j^+] /_i^\sharp [\underline{a}_k^+, \overline{a}_k^+]) V_j \leq b^+ /_i^\sharp [\underline{a}_k^+, \overline{a}_k^+] \ . \end{array} \tag{4.6}$$

We can then add the normalized constraints which gives, similarly to (4.3):

$$c \stackrel{\text{def}}{=} (c^+ \oslash_i^\sharp [\underline{a}_k^+, \overline{a}_k^+]) \oplus_i^\sharp (c^- \oslash_i^\sharp [-\overline{a}_k^-, -\underline{a}_k^-]) \ . \tag{4.7}$$

This operation is then performed for each pair of constraints $\langle c^+, c^- \rangle$ with opposed sign. We note that (4.6) is not an equivalence, so, our operator over-approximates the exact projection. Moreover, our operator can be performed soundly using floats only, which induces an extra loss of precision due to rounding. However, unlike Fourier–Motzkin's algorithm for float polyhedra, we do not need to apply slin to remove interval coefficients, which removes one cause of imprecision.

Join. The optimal abstraction of the join of two polyhedra can be modeled as their convex hull; it can be implemented exactly in rationals using Benoy et al.'s algorithm [BKM05] and easily approximated in float polyhedra (Sec. 4.1.4). However, this algorithm does not extend to interval polyhedra. Thus, in [CMWC09], we suggested a simple join $\cup_{ip}^\sharp$ that combines constraints pairwise, exploiting the ability of interval coefficients to be joined using the classic interval join $\cup_i^\sharp$. More precisely, for each pair of constraints $(\sum_j [\underline{a}_j^1, \overline{a}_j^1] V_j \leq b^1) \in \mathcal{C}_1$ and $(\sum_j [\underline{a}_j^2, \overline{a}_j^2] V_j \leq b^2) \in \mathcal{C}_2$, we add in $\mathcal{C}_1 \cup_{ip}^\sharp \mathcal{C}_2$ the constraint:

$$\sum_j ([\underline{a}_j^1, \overline{a}_j^1] \cup_i^\sharp [\underline{a}_j^2, \overline{a}_j^2]) V_j \leq \max(b^1, b^2) \ . \tag{4.8}$$

As for the float join of Sec. 4.1.4, it is worthwhile to refine the result of the join by adding the constraints from $\mathcal{C}_1 \cup \mathcal{C}_2$ that are satisfied by both $\mathcal{C}_1$ and $\mathcal{C}_2$. This algorithm safely over-approximates the join, but is not guaranteed to be optimal (this problem will be addressed in Sec. 4.2.2).

Abstract operations. With the exception of the join, which is handled as above, all the other abstract operations are handled as the operations in the rational and float constraint-based polyhedra: tests correspond to adding a constraint, assignments can be reduced to tests and projections, and inclusion checking and widening can be reduced to entailment checking. We refer the reader to [CMWC09] for a verbose presentation of these operators.

Application. The float interval polyhedra domain was implemented by Liqian Chen in Apron as a proof-of-concept and compared with the float polyhedra domain from the previous section on the same benchmark (see Sec. 4.1.5). We refer again the reader to [CMWC09] for the detailed experimental results and present here only a qualitative synthesis: while interval float polyhedra and float polyhedra give similar results in terms of precision and cost, which one is more precise or more efficient varies with the analyzed program.

On the one hand, interval polyhedra are more expressive and employ more complex algorithms (such as interval linear programming), which would imply that they are more precise and more costly. On the other hand, they use a weak join, unlike float polyhedra which try to over-approximate the exact join; hence, interval polyhedra may be less precise in some circumstances. Moreover, the weak join is less dependant on linear programming; as linear programming accounts, in both domains, for a large part of the cost, the weak join may improve the domain efficiency in some cases. As an example, in the analysis of the float rate limiter of Ex. 4.1.1, interval polyhedra managed to be more precise (inferring non-convex invariants) while being twice faster.

4.2.2 Exact interval polyhedra

While practical, the interval polyhedra domain presented in the last section suffers from imprecise projection and join operators; this is not only due to the use of floats, but also to fundamental algorithmic issues: even when computed with exact arithmetic, our algorithms do not compute optimal abstractions. In this section, we show that optimal operators can be constructed for interval polyhedra by returning to the original double description method, the rationale being that joins and projections are straightforward to compute on the generator representation. As we now seek optimality at the expense of efficiency, we consider rational bounds and exact arithmetic (avoiding soundness and precision issues due to rounding).

Constraint representation. As in Sec. 4.2.1, a *rational interval polyhedron* is then defined by an interval matrix $[\underline{\mathbf{A}}, \overline{\mathbf{A}}]$ and a vector $\vec{B}$, but now $\underline{\mathbf{A}}, \overline{\mathbf{A}} \in \mathbb{Q}^{m \times n}$ and $\vec{B} \in \mathbb{Q}^m$. The concretization γ_{ip} remains the same.

As stated before, an interval affine constraint reduces to a regular affine constraint when the sign of each variable is fixed. This leads to an equivalent formulation of an interval polyhedron using only affine constraints but twice as many variables. For each variable V_k, we denote respectively as V_k^+ and V_k^- its positive and its negative parts: $V_k^+ \stackrel{\text{def}}{=} \max(V_k, 0)$ and $V_k^- \stackrel{\text{def}}{=} \max(-V_k, 0)$, so that $V_k^+, V_k^- \geq 0$ and $V_k = V_k^+ - V_k^-$. Additionally, for each k, only one of V_k^+ and V_k^- is non-zero, which we note as: $\vec{V}^+ \cdot \vec{V}^- = 0$. This non-linear constraint is called the *complementary condition*. Hence, we represent an

interval polyhedra in $\mathbb{R}^n$ as a set of complementary vectors in $\mathbb{R}^{2n}$ obeying constraints encoded in a matrix $\mathbf{A} \in \mathbb{Q}^{m\times 2n}$ and a vector $\vec{B} \in \mathbb{Q}^m$. We have:

$$\begin{aligned}\gamma_x(\langle \mathbf{A},\, \vec{B}\rangle) \stackrel{\text{def}}{=}\\ \{\, \langle \vec{V}^+,\, \vec{V}^-\rangle \in \mathbb{R}^{2n} \mid \mathbf{A} \times \begin{bmatrix} \vec{V}^+ \\ \vec{V}^- \end{bmatrix} \leq \vec{B}, \\ \vec{V}^+, \vec{V}^- \geq \vec{0},\, \vec{V}^+ \cdot \vec{V}^- = 0 \,\}\ .\end{aligned} \tag{4.9}$$

The use of interval coefficients, which was justified on floats by the imprecision caused by rounding errors, might not seem as useful when considering exact computations. However, we note that $|V_k| = V_k^+ + V_k^-$; hence, $\gamma_x(\langle \mathbf{A},\, \vec{B}\rangle)$ can also represent constraints involving the absolute value of the variables. Such relations occur in many programs, and inferring them is useful. Due to this change of focus, this domain is also called the *linear absolute value relation domain* [CMWC11] (although its expressiveness is the same as interval polyhedra).

Example 4.2.2. The constraints $x+2|x| \geq 10$ and $[-1,3]x \geq 10$ are equivalent. Both can be represented as $3x^+ + x^- \geq 10$ with the extra conditions: $x^+, x^- \geq 0$, $x^+x^- = 0$.
End of example.

We refer the reader to [CPS92, MP95] for more information on complementary linear constraint systems, which are well-studied in the literature, and to [Roh06, CMWC11] for more information on the links between these systems, affine interval constraints, and affine constraints with absolute values.

A generator representation. The main result underlying the construction of our domain is that we can derive a generator representation for a polyhedron with complementary conditions $\gamma_x(\langle \mathbf{A},\, \vec{B}\rangle)$ from the generator representation of the polyhedron $\gamma_p(\langle \mathbf{A},\, \vec{B}\rangle)$ considered without complementary condition (this result is proved in [CMWC11]). More precisely, assume that we are given such a generator representation: $\mathbf{P}$ and $\mathbf{R}$ such that $\gamma_p([\mathbf{P}, \mathbf{R}]) = \gamma_p(\langle \mathbf{A},\, \vec{B}\rangle)$ (2.11). Then, to represent $\gamma_x(\langle \mathbf{A},\, \vec{B}\rangle)$, it is sufficient to consider the complementary generators, i.e., the sub-matrices $\mathbf{P}'$ and $\mathbf{R}'$ of columns of $\mathbf{P}$ and $\mathbf{R}$ that obey the complementary condition (i.e., columns $[\vec{C}^{+\,t}\, \vec{C}^{-\,t}]^t$ such that $\vec{C}^+ \cdot \vec{C}^- = 0$). We then consider all the maximal subfamilies $\langle \mathbf{P}'',\, \mathbf{R}''\rangle$ of generators from $\langle \mathbf{P}',\, \mathbf{R}'\rangle$ that satisfy:

$$\forall \begin{bmatrix} \vec{V}^+ \\ \vec{V}^- \end{bmatrix} \in \gamma_p(\langle \mathbf{P}'',\, \mathbf{R}''\rangle) : \vec{V}^+ \cdot \vec{V}^- = 0$$

i.e., each family corresponds to a polyhedron of vectors satisfying the complementary condition, which we call a *complimentary polyhedron*. Recall that an interval polyhedron is, in general, composed of several convex polyhedra (at most one per orthan). Intuitively, each complementary polyhedron in $\mathbb{R}^{2n}$ corresponds to a convex polyhedron in $\mathbb{R}^n$ in this decomposition.

Representation conversion. The previous result suggests a simple algorithm to convert constraints to generators: first, apply Chernikova's algorithm as in the classic double description method, and then filter out generators that do not obey the complementary condition, and finally group them into families corresponding to a complementary polyhedron each. This last step is very costly: it can be reduced to the problem of finding maximal subgraphs of a directed graph, which is NP-complete [GJ79]. Moreover, it makes the decomposition of an interval polyhedron into its convex parts explicit. Fortunately, this is not necessary: none of the abstract operations explicitly require this decomposition; it is sufficient to manipulate unordered lists of complementary generators and leave the decomposition into convex parts implicit. Thus, from an algorithmic point of view, the operators are not equivalent to that of a disjunctive completion [CC79b], that would manipulate explicit lists of convex polyhedra. We note additionally that the polyhedron $\langle \mathbf{A},\, \vec{B}\rangle$ may contain many non-complementary generators, and that it is wasteful to enumerate them all before filtering them. To improve the efficiency, we can exploit the incremental nature of Chernikova's algorithm to filter them as early as possible during the construction of the generator set, the same way redundant generators are removed as soon as possible by LeVerge's modification to Chernikova's algorithm [LeV92]. Our algorithm is detailed fully in [CMWC11].

Abstract operations. The operators used on regular rational polyhedra in the double description representation, presented in Sec. 2.4.2, can be reused as is, assuming that the relevant representation is always available: tests consist in adding constraints, projections and joins in adding generators, assignments are reduced to projections and tests, entailment (and so inclusion checking and widening) is reduced to checking that generators satisfy a constraint. A point of note is that the projection is no longer exact. Indeed, interval polyhedra are not closed under projection, and so, no exact abstraction can be devised; the same is also true of assignments. However, the projection, the assignment of affine expressions, and the join operators are all optimal: they always output the smallest interval polyhedron encompassing the concrete result of the operation. We refer the reader to [CMWC11] for a more detailed presentation of these operators.

Application. The domain was implemented and tested in the Apron library, and compared to both regular rational polyhedra based on the double description method, and float interval polyhedra from the preceding section. As expected, the domain is less efficient but more precise than the two others: it can discover invariants out of their reach. In addition to its ability to infer relations involving absolute values, the domain surprised us by inferring exactly, in some cases, disjunctive invariants after joining program branches. This suggests that the domain can be an alternative to generic techniques, such as disjunctive completion [CC79b] and trace partitioning [RM07], to infer disjunctive properties.

In addition to its potential applications, this domain is interesting from a theoretical point of view as all its operators are optimal. Similarly to classic polyhedra (and unlike float polyhedra and float interval polyhedra) it is a perfectly semantic domain, where the approximation is only caused by the choice of abstract properties and by not the algorithms implementing the abstract operators.

4.2.3 Interval affine equalities

In the last two sections, we discussed extensions of the polyhedra domain to interval coefficients. We now discuss the extension of another domain: the *affine equality domain*, initially proposed by Karr [Kar76]. Affine equalities are less expressive

than affine inequalities, but they are also much more efficient: the domain is based on a quadratic memory representation and a cubic-time Gauss elimination algorithm (while polyhedra are unbounded in theory and exponential in practice [NQ10]). Our goal is to try and extend affine equalities to interval coefficients, thereby achieving an expressiveness between regular affine equalities and interval polyhedra, while not sacrificing performance.

Our construction is presented on exact rationals, but can be easily adapted to float arithmetic.

Representation. An abstract element in the *interval affine equality domain* is composed of an interval matrix represented as an upper bound matrix $\overline{\mathbf{A}}$ in $(\mathbb{Q} \cup \{+\infty\})^{m\times n}$ and a lower bound matrix $\underline{\mathbf{A}}$ in $(\mathbb{Q} \cup \{-\infty\})^{m\times n}$, and an interval vector represented as an upper bound vector $\overline{\vec{B}}$ in $(\mathbb{Q} \cup \{+\infty\})^m$ and a lower bound vector $\underline{\vec{B}}$ in $(\mathbb{Q} \cup \{-\infty\})^m$. Similarly to (4.4), the concretization is:

$$\begin{aligned}\gamma_{il}(\langle[\underline{\mathbf{A}}, \overline{\mathbf{A}}], [\underline{\vec{B}}, \overline{\vec{B}}]\rangle) \stackrel{\text{def}}{=}\\ \{\, \vec{V} \in \mathbb{R}^n \mid \exists \mathbf{A} \in \mathbb{R}^{m\times n} : \underline{\mathbf{A}} \le \mathbf{A} \le \overline{\mathbf{A}},\\ \exists \vec{B} \in \mathbb{R}^m : \underline{\vec{B}} \le \vec{B} \le \overline{\vec{B}},\, \mathbf{A} \times \vec{V} = \vec{B} \,\}\ .\end{aligned} \tag{4.10}$$

Note that we allow bounded as well as unbounded intervals. Unbounded intervals in the constant part allow representing inequalities: for instance, $x = [1, +\infty]$ simply means $x \ge 1$. Bounded interval coefficients for variables allow representing non-convex and unconnected sets, similarly to interval polyhedra: in each orthan, the system is equivalent to a regular affine inequality system of the form $\underline{\vec{B}} \le \mathbf{A} \times \vec{V} \le \overline{\vec{B}}$. Unbounded interval coefficients for variables extend the expressiveness to allow strict sign constraints: for instance, $[-\infty, +\infty]x = 1$ is equivalent to $x \ne 0$.

An arbitrary interval affine inequality can be encoded as an interval affine equality. Thus, allowing arbitrary interval matrices $[\underline{\mathbf{A}}, \overline{\mathbf{A}}]$ would result in a domain slightly more expressive (due to the extension to unbounded intervals) than interval polyhedra, and so, at least as costly. To ensure that our algorithms are cubic in the worst case, we restrict $[\underline{\mathbf{A}}, \overline{\mathbf{A}}]$ to be in *row echelon form*:

$$\begin{aligned}\forall i : \exists j : [\underline{A}_{ij}, \overline{A}_{ij}] \ne [0,0] \wedge \forall j' < j : [\underline{A}_{ij'}, \overline{A}_{ij'}] = [0,0]\\ \wedge \forall i' > i : [\underline{A}_{i'j}, \overline{A}_{i'j}] = [0,0]\end{aligned}$$

i.e., each row of $[\underline{\mathbf{A}}, \overline{\mathbf{A}}]$ contains a unique leading variable (first column with non-zero coefficient), which does not appear in subsequent rows. From an efficiency point of view, the choice of a row echelon form is motivated by noting that a matrix in row echelon form has at most $n = |\mathcal{V}|$ rows, which ensures that abstract elements have a quadratic worst-case memory cost (compare this to the unbounded size of interval polyhedra). Concerning expressiveness, the rationale is that arbitrary conjunctions of (non-interval) affine equalities can always be put into an equivalent row echelon form, without loss of precision, and so, we ensure that our domain is at least as expressive as the affine equality domain.

Constraint addition. Adding a constraint in a polyhedral domain is a simple syntactic operation, because a polyhedron can maintain an arbitrary number of constraints. The operation is more complex in the interval affine domain as we must ensure that the system remains in row echelon form. Assuming that we are given a system in row echelon form and an interval constraint to add $c \stackrel{\text{def}}{=} (\sum_{j\ge k}[\underline{a}_j, \overline{a}_j]V_j = [\underline{b}, \overline{b}])$, with leading variable V_k, we proceed as follows:

- if the system has no constraint where V_k appears in leading position, we add c to the system and stop;
- if the system has a constraint c' where V_k appears in leading position, then:
  - if c is more "precise than" c' (for a notion of precision presented below), we replace c' with c in the system;
  - we combine c and c' to get a constraint c'' with leading variable V_l with $l > k$;
  - and we recursively add c'' to the system.

This algorithm terminates because the index of the leading variable of the constraint to add increases strictly. It remains to explain what is meant by "c is more precise than c'" and by "combine c and c'". Generally, the sets of points defined by two different constraints are incomparable. Thus, we rely on synthetic heuristics to assess the relative precision of constraints. An example of such heuristic consists in choosing the constraint with the smallest width, where the width of a constraint $\sum_j [\underline{a}_j, \overline{a}_j]V_j = [\underline{b}, \overline{b}]$ with respect to a bounding box $V_j \in [\underline{V}_j, \overline{V}_j]$ is defined as the constraint evaluated on the bounding box using interval arithmetic: $\sum_j [\underline{a}_j, \overline{a}_j] \times^\sharp_i [\underline{V}_j, \overline{V}_j] -^\sharp_i [\underline{b}, \overline{b}]$. We refer the reader to [CMWC11] for more details on this heuristic and several alternate ones. Likewise, there are several ways to combine two constraints c and c' that lead to the elimination of V_k. For instance, when the coefficient $[\underline{a}_k, \overline{a}_k]$ and $[\underline{a}'_k, \overline{a}'_k]$ of V_k in c and c' have a different sign (i.e., $\underline{a}_k > 0$ and $\overline{a}'_k < 0$), we can apply the same technique we used in the interval version of Fourier–Motzkin's elimination (4.7) and compute:

$$(c \oslash^\sharp_i [\underline{a}_k, \overline{a}_k]) \oplus^\sharp_i (c' \oslash^\sharp_i [-\overline{a}'_k, -\underline{a}'_k])\ . \tag{4.11}$$

This technique and other ones, as well as their respective merit, are also discussed in [CMWC11]. Note that constraint addition is an approximate operation, which is in contrast to the large majority of abstract domains (including all the polyhedral domains we presented before; a notable exception in the literature is the zonotope domain [GGP10], which is a restriction of polyhedra).

An important point of note is that, when the constraints in the system and the constraint to add are all affine without interval, our algorithm reduces to adding the constraint and applying Gaussian elimination to re-normalize the system: constraint addition is thus exact in this case.

Given an arbitrary set of constraints, it is possible to construct a system in row echelon form by adding the constraints one by one with this algorithm. While in the case of affine equality constraints the construction of the row echelon form is a normalization process keeping the semantic intact, for interval affine equalities this process incurs an actual abstraction.

Projection. Eliminating a variable V_k in the regular affine equality domain consists in using a row where V_k appears to eliminate the occurrences of V_k in other rows. After this process, there remains at most one constraint where V_k occurs, which is removed. We extend this algorithm to interval affine constraints, and eliminate V_k using the constraint combination technique introduced for constraint addition. Note that this projection is generally not exact nor optimal; however, as before, if the constraint system is actually affine, it reduces to the projection on the affine equality domain, which is exact.

Assignments are then modeled using projections and tests, as usual.

Comparison. Exact entailment checking, and so exact comparison of abstract elements, can be achieved by interval linear programming (4.5), similarly to the case of the interval polyhedra from Sec. 4.2.1. However, this is a costly operation. In order to be consistent with our goal to construct a less precise but cheaper abstract domain, we propose a coarser comparison algorithm.

Given two constraints $c \stackrel{\text{def}}{=} (\sum_{j\geq k}[\underline{a}_j, \overline{a}_j]V_j = [\underline{b}, \overline{b}])$ and $c' \stackrel{\text{def}}{=} (\sum_{j\geq k}[\underline{a}'_j, \overline{a}'_j]V_j = [\underline{b}', \overline{b}'])$, we denote as $c \sqsubseteq^\sharp_{il} c'$ the element-wise inclusion of coefficients:

$$\forall j : [\underline{a}_j, \overline{a}_j] \subseteq [\underline{a}'_j, \overline{a}'_j] \wedge [\underline{b}, \overline{b}] \subseteq [\underline{b}', \overline{b}'] \ . \tag{4.12}$$

Then, if $c \sqsubseteq^\sharp_{il} c'$, any point satisfying c also satisfies c'. This is extended to sets of constraints as:

$$\mathcal{C}_1 \sqsubseteq^\sharp_{il} \mathcal{C}_2 \iff \forall c_2 \in \mathcal{C}_2 : \exists c_1 \in \mathcal{C}_1 : c_1 \sqsubseteq^\sharp_{il} c_2 \tag{4.13}$$

which leads to a cubic-time algorithm. Then $\sqsubseteq^\sharp_{il}$ implies the inclusion of the concretizations of abstract elements. The converse implication, however, does not hold.

Join. Similarly to the weak join of float interval polyhedra (4.8), we can model the join by element-wise interval joins. For each pair of constraints $c \in \mathcal{C}$ and $c' \in \mathcal{C}'$ with the same leading variable, we generate the constraint:

$$\sum_j ([\underline{a}_j, \overline{a}_j] \cup^\sharp_i [\underline{a}'_j, \overline{a}'_j])V_j = [\underline{b}, \overline{b}] \cup^\sharp_i [\underline{b}', \overline{b}']$$

which is implied by both $\mathcal{C}$ and $\mathcal{C}'$ (constraints with leading variable appearing in only one system are discarded). The conjunction of all these constraints is in row echelon form and over-approximates the join. More precise joins are possible, such as adapting the join by Benoy et al. [BKM05] used in float polyhedra (Sec. 4.1.3). We refer the reader to [CMWC11] for a description of more advanced joins.

Widening. On all the domains presented before, the widening always proceeds by filtering constraints in order to keep only the stable ones. This requires an entailment check, which is a precise or even exact operation in those domains, but is rather coarsely approximated on interval affine equalities. Instead of relying on entailment checking, we construct a widening based on the point-wise extension of an interval widening. More precisely, given any interval widening $\triangledown_i$ (such as the classic widening from Fig. 2.10), each pair of constraints $c \in \mathcal{C}$ and $c' \in \mathcal{C}'$ with the same leading variable is replaced with:

$$\sum_j ([\underline{a}_j, \overline{a}_j] \triangledown_i [\underline{a}'_j, \overline{a}'_j])V_j = [\underline{b}, \overline{b}] \triangledown_i [\underline{b}', \overline{b}']$$

which indeed over-approximates the join and ensures the convergence in finite time. More refined widenings are proposed in [CMWC11], such as a widening that tries to keep affine constraints intact when they can be proved to be stable, in the hope of discovering affine invariants as precise as the (non-interval) affine equality domain.

Application. As the previous domains, the interval affine equality domain has been implemented in Apron and tested on some small programs. The domain is actually implemented using float bounds instead of exact rationals; the soundness is guaranteed by simply rounding upper bounds towards $+\infty$ and lower bounds towards $-\infty$, as in the interval domain. When compared to an implementation of the affine equality domain with exact rationals, our domain was shown to be consistently more precise, and similar in cost (from twice slower to twice faster). The precision improvement is explained by the improved expressiveness (in particular, the ability to represent inequalities). When compared to NewPolka (Apron's built-in polyhedra domain using exact rationals) our domain proved to have a similar speed for programs with few variables, but additionally scaled up to programs with 32 or more variables with a reasonable analysis time (less than 30s) while NewPolka consistently timed-out after 1h. Concerning the precision, our domain sometimes fails to discover constraints inferred by polyhedra, but it is often able to infer more; this can be explained by a combination of extended expressiveness (in particular, the ability to represent non-convex sets) and non-optimal abstract operations. Comparing our domain to the float interval polyhedra domain shows similar results, although the differences are less stressed because float interval polyhedra scale up better and are more expressive than exact rational polyhedra. Because both float interval affine equalities and float interval polyhedra use their own, incomparable, approximated abstract operators, each one can generally infer constraints not inferred by the other. We refer the reader to [CMWC11] for a more detailed analysis of the experimental results.

4.2.4 Discussion

In this section, we have proposed several new numeric abstract domains that extend the expressiveness of polyhedra and affine equalities by using interval coefficients. They enrich the ever growing library of available domains. Our work remains, for the moment, mostly fundamental. Future work includes the construction of more robust implementations to test them in the large, analyzing actual programs in realistic programming languages. Some of our domains make deliberate choices to sacrifice precision and improve efficiency and, in the absence of a best abstraction, rely on local heuristics to make decisions. This is the case in particular for interval affine equalities, which are sensitive to the chosen ordering of variables and measure of constraint precision. This is also the case for all the domains that use the slin linearization function. New heuristics should be developed once the limits of the existing ones in actual analyses become apparent. Concerning optimal and costly domains, such as exact interval polyhedra, analyzing actual programs will also be useful to develop restrictions that scale better while maintaining a high precision where it matters in practice (for instance, using packing techniques [BCC$^+$10a]).

Related work. The search for new numeric domains is a very active field of abstract interpretation. We now compare the abstract domains we presented to other, related proposals.

Domains based on interval affine constraints can represent natively non-convex and unconnected sets, which is quite rare. Representing non-convex invariants is generally achieved using a generic domain lifting, such as arbitrary disjunctive completions [CC79b] or disjunctions guided by the history of com-

putations as in trace partitioning [RM07]. Arbitrary disjunctions often suffer from scalability issues, triggering the need for heuristics (such as periodically replacing elements with their approximate join). Moreover, widenings for disjunctive completions are difficult to design [BHZ04]. Instead of representing disjunctions of conjunctions of convex constraints, our domains achieve non-convexity by a direct conjunction of non-convex constraints. Another example of such domain is the max-plus polyhedra domain by Allamigeon et al. [AGG08].

As the interval domain is often too imprecise and polyhedra are not scalable, much effort has been devoted to the design of domains in-between those two in terms of cost and precision. During my PhD, I participated in the design of some weakly-relational domains, such as octagons [Min06b] that restrict polyhedra to constraints of the form $\pm V_i \pm V_j \leq c$. Other proposals include: Two-Variable-Per-Inequality by Simon et al. [SKH02], octahedra by Clarisó [CC04], pentagons [LF10]. Our interval affine equality domain is in-between the affine equality domain by Karr [Kar76] and polyhedra; another instance of such domain is the sub-polyhedra domain proposed by Laviron [LL09], which is less expressive as intervals are only allowed in the constant term. Another, less expressive proposal to add bound information to affine equalities is to construct a partially reduced product (Sec. 2.2) between intervals and affine equalities, as proposed by Feret [Fer01].

We were inspired to use linear programming by the work by Simon et al. on polyhedra [SK05]. Linear programming is also used by Sankaranarayanan et al. to construct template polyhedra [SSM05], a flexible restriction of polyhedra that generalizes octagons and octahedra.

We advocate the use of float coefficients to analyze programs using float. The float domains we proposed are based on constraints only, lacking a generator representation. Dually, Ghorbal et al. propose to analyze float programs with zonotopes [GGP09], a restriction of polyhedra based on a generator-only representation. Zonotopes do not enjoy a simple and exact modeling of tests; similarly to our interval affine equalities, constraint addition must be over-approximated [GGP10]. It remains, in future work, to bridge the gap between those domains and construct a float double description method, complete with a sound float version of Chernikova's algorithm.

Chapter 5

Abstracting C data-types

The idealized programming language that we introduced in Sec. 2.3.1, then extended to parallel software in Chap. 3, and analyzed using classic or novel abstract domains (Chap. 4), offers only one data-type: mathematical reals in $\mathbb{R}$. Even when we extended the language to floating-point numbers, in Sec. 2.4.4, we considered them as a subset of reals and abstracted their operations using real operators. Considering a real semantics allowed us to view invariants as subsets of a vector space $\mathbb{R}^n$ and exploit classic results in linear arithmetic to construct our abstract domains (for instance, Chap. 4 used many results from linear programming). However, actual programming languages feature more complex and numerous data-types, which we must handle to construct a sound and useful static value analysis.

In this chapter, we consider more realistic data-types, inspired from the C programming language, with their associated expressions and operator semantics. Firstly, we consider machine integers with wrap-around, in Sec. 5.1. Secondly, we discuss, in Sec. 5.2, the abstraction of structured types, such as arrays or structures, but also union types and pointer operations that allow navigating within structured objects. Finally, we go back to floating-point numbers in Sec. 5.3 with a more detailed handling than in Sec. 2.4.4. In each case, we start by defining a very precise concrete semantics of the considered data-type and its operations. These semantics go beyond viewing a type as a set of values, and also take into account their binary encoding in memory. We are then able to give a proper semantics to operations that rely on it. For instance, the binary representation of integers is useful to model bit-wise operations in C, while the binary representation of structures gives a semantics to "type-punning" constructions. Finally, we provide abstractions able to exploit the knowledge of this encoding. Our motivation is the precise static analysis of C programs that rely on such knowledge for their correctness.

The work presented here has been published in [Min12a] and [Min06a]. It stemmed from our experience developing the Astrée C static analyzer (Sec. 6.2) and its extension AstréeA (Sec. 6.3). While initial versions of Astrée [BCC+03] used a straightforward modeling of data-types, it became obvious when trying to extend the class of programs it supported that many C programs are not portable and feature operations dependent on the machine representation of data-types, hence the need for the semantics and the abstractions presented here.

5.1 Machine integers

We start here by extending our language with machine integer data-types and their associated operations.

5.1.1 Extended language

Syntax. To support machine integers, we slightly modify the language syntax of Fig. 2.1. Firstly, we introduce *machine integer types*, with the following grammar:

$$\begin{array}{rcl} \textit{int-type} ::= & (\textbf{signed} \mid \textbf{unsigned}) \\ & (\textbf{char} \mid \textbf{short} \mid \textbf{int} \mid \textbf{long}) \end{array} \tag{5.1}$$

Types can vary in size (from **char** to **long**) and can be signed or unsigned. In expressions, unary $\circ$ and binary $\diamond$ operators are changed to match the C ones:

$$\begin{array}{rcl} \circ & ::= & \texttt{-} \mid \texttt{\~{}} \mid \texttt{(}\ \textit{int-type}\ \texttt{)} \\ \diamond & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{\^{}} \mid \texttt{>>} \mid \texttt{<<} \end{array} \tag{5.2}$$

In addition to the operators based on classic mathematical integers (`+`, `-`, `*`, `/`, and the remainder `%`), there exists bit-level operations: bit-wise negation `~`, and `&`, or `|`, and exclusive or `^`, as well as bit shifts `>>` and `<<`. Finally, we add a *cast operator* `(` *int-type* `)`, which converts an expression to a given integer type.

Type representation. Each type denotes a set of possible values, but also a representation for these values as bit strings in the memory. The actual size of each integer type depends on the architecture and the compiler, so, we assume that we are given a function $\textit{sizeof} \in \textit{int-type} \to \mathbb{N}$ providing the size of types in units of 8-bit bytes. Unsigned integers are represented using a pure *binary representation*: $b_{n-1} \cdots b_0 \in \{0,1\}^n$ represents $\sum_{i=0}^{n-1} 2^i b_i$. Signed integers use a *two's complement representation*: $b_{n-1} \cdots b_0 \in \{0,1\}^n$ represents $-2^{n-1} b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$, which reduces to the unsigned representation for positive numbers ($b_{n-1} = 0$), and to the complement to 2^n for negative numbers ($b_{n-1} = 1$).[1] The range of a type is then:

$$\textit{range}(t) \stackrel{\text{def}}{=} \begin{cases} [0, 2^{8 \times \textit{sizeof}(t)} - 1] & \text{if } t \text{ is unsigned} \\ [-2^{8 \times \textit{sizeof}(t) - 1}, 2^{8 \times \textit{sizeof}(t) - 1} - 1] & \text{if } t \text{ is signed} \end{cases}. \tag{5.3}$$

Typing. We assume that each variable $V \in \mathcal{V}$ has a user-provided type $\textit{type}(V) \in \textit{int-type}$. Then, expressions can be given a type $\textit{type}(\textit{expr}) \in \textit{int-type}$ by structural induction. The C typing rules are somewhat complex, in order to account for binary operations with arguments of heterogeneous types and the preference towards a native integer type (denoted as

[1] In theory, the C standard [ISO07] allows other representations, but our choice corresponds to the vast majority of architectures.

int). As a consequence, the value of a sub-expression is often converted to another type when used as argument to an operator. In the following, we assume that all these implicit conversions have been materialized explicitly by cast operators `(`*int-type*`)`. Without detailing these rules (see [ISO07]), we illustrate their subtlety on one often overlooked rule: *integer promotion*. It states that values of a type smaller than **int** are converted to **int**. This rule causes values with the same binary representation but different types to behave differently. For instance, the unsigned byte 255 and the signed byte -1 have the same representation; however, $255 \gg 1 = 127$ while $(-1) \gg 1 = -1$. Integer promotion is in fact value preserving [AI99], as opposed to representation preserving.

Concrete semantics. At the hardware level, integers are bit strings of fixed length. One way to define the semantics is thus at the bit level (so called "bit blasting" [BK11]). This works well when representing value sets explicitly, for instance with *binary decision diagrams* [Bry86], but it is not adapted to abstraction in numeric domains. We provide, instead, a semantics expressed using classic integer arithmetic. The choice of using integers in $\mathbb{Z}$ rather than bit strings is justified when the intent of the programmer is to use modular integers as mathematical integers most of the time. Hence, arithmetic operators, such as `+` and `*` can be modeled easily as long as no overflow occurs, while we accept that overflow behaviors and less frequently used operators, such as `<<` and `&`, are more complex to model, at the risk of being less precisely abstracted. We oppose this to binary decision diagram representations that can model easily `&` but do not scale up well for some arithmetic operators such as `*`. At the language level, it is also justified by the value preserving property of integer promotion.

The effect of a machine integer operation can be modeled in two steps, similarly to the way floating-point operations are performed (Sec. 2.4.4): it first computes an exact integer in $\mathbb{Z}$, and then maps it to the range of the result type by modular wrap-around. For instance, we get, for binary operators:

$$
\begin{aligned}
&\mathbb{E}[\![\, e_1 \diamond_\omega\ e_2 \,]\!]\rho \overset{\text{def}}{=} \\
&\quad \text{let } \langle V_1,\, O_1\rangle = \mathbb{E}[\![\, e_1 \,]\!]\,\rho \text{ in} \\
&\quad \text{let } \langle V_2,\, O_2\rangle = \mathbb{E}[\![\, e_2 \,]\!]\,\rho \text{ in} \\
&\quad \langle \{\, \mathit{wrap}(v_1 \diamond v_2, [l,h]) \mid v_i \in V_i, \diamond \notin \{/, \%\} \vee v_2 \neq 0 \,\}, \\
&\quad\ O_1 \cup O_2 \cup \{\, \omega \ \text{ if } \diamond \in \{/, \%\} \wedge 0 \in V_2 \,\} \cup \\
&\quad\ \{\, \omega \ \text{ if } \exists v_1 \in V_1, v_2 \in V_2 :\ v_1 \diamond v_2 \notin [l,h] \,\}\rangle \\
&\text{where } [l,h] \overset{\text{def}}{=} \mathit{range}(\mathit{type}(e_1 \diamond_\omega\ e_2))
\end{aligned}
$$

where the *wrap* function models *wrap-around*:

$$
\begin{aligned}
&\mathit{wrap}(v,[l,h]) \overset{\text{def}}{=} \\
&\quad \min\{\, v' \mid v' \geq l \wedge \exists k \in \mathbb{Z} :\ v = v' + k(h-l+1) \,\}\ .
\end{aligned}
\tag{5.4}
$$

In case of an overflow, our semantics generates an error ω and outputs the modular result, so that we can alert the user of the wrap-around behavior while continuing the analysis in case the wrap-around is intended.

It remains to define the semantics of the operators. The arithmetic operators `+`, `-`, `*`, `/`, `%` have their usual meaning in $\mathbb{Z}$ (with `/` rounding towards 0 and $a \,\%\, b \overset{\text{def}}{=} a - (a/b)*b$). To provide a semantics for bit-level operators on $\mathbb{Z}$, independently from the machine representation of a given type, we use $2-$adic integers, i.e., infinite strings of 0 and 1. The $2-$*adic representation* $p(x) \in \{0,1\}^\omega$ of an integer $x \in \mathbb{Z}$ generalizes the two's complement representation:

$$
\begin{array}{lcl}
\sim x & \overset{\text{def}}{=} & p^{-1}(\neg p(x)) = -x-1 \\
x \,\&\, y & \overset{\text{def}}{=} & p^{-1}(p(x) \wedge p(y)) \\
x \mid y & \overset{\text{def}}{=} & p^{-1}(p(x) \vee p(y)) \\
x \,\hat{}\, y & \overset{\text{def}}{=} & p^{-1}(p(x) \oplus p(y)) \\
x \ll y & \overset{\text{def}}{=} & \lfloor x \times 2^y \rfloor \\
x \gg y & \overset{\text{def}}{=} & \lfloor x \times 2^{-y} \rfloor
\end{array}
$$

Figure 5.1: Definition of bit-wise operators.

$$
(p(x))_i \overset{\text{def}}{=} \begin{cases} \lfloor x/2^i \rfloor \mod 2 & \text{if } x \geq 0 \\ \neg(p(-x-1))_i & \text{if } x < 0\ . \end{cases}
\tag{5.5}
$$

Note that p is one-to-one on its image $\{\, p(z) \mid z \in \mathbb{Z} \,\}$, which is exactly the set of infinite strings that are stable, i.e., either always 0 or always 1 after a certain index. The semantics of operators can then be defined as shown in Fig. 5.1, using the element-wise and operator $\wedge$, or $\vee$, exclusive or $\oplus$, and complement $\neg$. Note that the operations defined in Fig. 5.1 are directly available in existing arbitrary precision integer libraries (such as GMP [GNUa]).

5.1.2 Adapting classic domains

Our machine integer semantics is completely defined in terms of mathematical integers. Thus, we consider abstract domains abstracting integer-valued environments, with concrete domain $\mathcal{D} \overset{\text{def}}{=} \mathcal{P}(\mathcal{V} \to \mathbb{Z})$. We first present existing techniques to adapt the intervals and polyhedra domains to integers, and show their shortcomings.

Intervals. The interval domain from Sec. 2.4.1 abstracting $\mathcal{P}(\mathcal{V} \to \mathbb{R})$ can be easily adapted to machine integers. As intervals enjoy a Galois connection (Fig. 2.9), we can construct optimal abstractions of operators. We will not present them in detail (in particular, $+_i^\sharp$, $-_i^\sharp$, $*_i^\sharp$ are defined as in Fig. 2.11) but focus on the wrap-around effect. The best abstraction of *wrap* is then:

$$
\mathit{wrap}_i^\sharp([l,h],[l',h']) = \begin{cases} [\mathit{wrap}(l,[l',h']), \mathit{wrap}(h,[l',h'])] \\ \qquad \text{if } (l' + (h'-l'+1)\mathbb{Z}) \cap [l+1,h] = \emptyset \\ [l',h'] \quad \text{otherwise} \end{cases}
\tag{5.6}
$$

which returns the full interval $[l',h']$ when $[l,h]$ crosses a boundary in $l' + (h'-l'+1)\mathbb{Z}$. This is the case when the set $\{\, \mathit{wrap}(v,[l',h']) \mid v \in [l,h] \,\}$ is not convex. This case results in a great a loss of precision, as shown below.

Example 5.1.1. Consider computing:

$$
X \leftarrow \textbf{(signed char)}(\textbf{(unsigned char)}\ X + \\ \textbf{(unsigned char)}\ Y)
$$

where X and Y have type **signed char** and range in $[-1,1]$.

Firstly, the cast to **unsigned char** computes, in the concrete, the set $\{\, 0,1,255 \,\}$. The interval abstraction of this set is computed as $\mathit{wrap}_i^\sharp([-1,1],[0,255]) = [0,255]$, which is optimal but not exact. Secondly, an addition is performed to give an **int** result (due to integer promotion), which gives $[0,255] +_i^\sharp [0,255] = [0,510]$. Thirdly, the result is cast back

to **signed char**, which gives $wrap_i^\sharp([0,510],[-128,127])$, that is, $[-128,127]$. By comparison, the concrete result would be $[-2,2]$. We note that the concrete result is exactly representable as an interval, and that each interval operation is optimal but, as the combination of optimal abstractions is not optimal, the accumulation of imprecision gives a very coarse interval result.

This example might seem unrealistic, and yet such code patterns are used in actual industrial code generators, such as TargetLink [dSp] where they are known as "*compute-through-overflow*."
End of example.

Polyhedra. The polyhedra domain (Sec. 2.4.2) and, more generally, most relational domains, are based on field properties of reals that are not true of integers. However, with a little care, polyhedra can be used to soundly abstract sets of points with integer coordinates, in $\mathcal{P}(\mathbb{Z}^n)$. The principle is to keep a syntactic representation based on constraints or generators with coefficients in $\mathbb{Q}$, but change its meaning: the concretization γ_p (2.11) is replaced with $\gamma_{\mathbb{Z}p}$ that keeps only integer points:

$$\gamma_{\mathbb{Z}p}(P) \stackrel{\text{def}}{=} \gamma(P) \cap \mathbb{Z}^n \ . \tag{5.7}$$

Then, all the operators described in Sec. 2.4.2 implemented in rationals are also sound with respect to $\gamma_{\mathbb{Z}p}$. However, some of them that were exact or optimal for γ_p are no longer exact or optimal for $\gamma_{\mathbb{Z}p}$.

Example 5.1.2. Consider the polyhedron P in $\mathbb{R}^2$ defined by the constraint $x = 2y$. In integers, it represents $\gamma_{\mathbb{Z}p}(P) = \{\,(x,y) \in \mathbb{Z}^2 \mid x = 2y\,\}$. Then, projecting y in the concrete gives: $\mathbb{S}[\![\, y \leftarrow [-\infty,+\infty] \,]\!](\gamma_{\mathbb{Z}p}(P)) = (2\mathbb{Z})\times\mathbb{Z}$, which cannot be represented exactly as a polyhedron. Applying the polyhedron projection, we get $\gamma_{\mathbb{Z}p}(\mathbb{S}_p^\sharp[\![\, y \leftarrow [-\infty,+\infty] \,]\!](P)) = \mathbb{Z}^2$. Hence, the projection, which is exact with respect to γ_p, is no longer exact with respect to $\gamma_{\mathbb{Z}p}$.
End of example.

Note that various methods exist to exploit the restriction to integer coordinates in order to improve the precision. They range from simple low-cost integer tightening of constraints, as used for instance in Apron [JM09], to costly integer linear programming methods, as in the Omega test [Pug92], but we will not discuss them further.

We now discuss the problem of modeling machine integer operations. Most polyhedra libraries only feature abstract operators to model affine assignments and tests, as these enjoy exact abstractions (on rational at least). Nevertheless, we proposed in Sec. 2.4.3 a technique to abstract non-affine expressions into (interval) affine ones, which can be extended to support bit-level operators as well: any application of `~`, `&`, `|`, `^`, `>>`, `<<` is replaced with an interval obtained by evaluating the sub-expression using interval arithmetic.

For the cast operation, one simple solution is to test whether the argument overflows the range of the type. If it does not, then the semantics of the cast is the identity, and the cast can be safely ignored. If it does overflow, a wrap-around occurs, and we use the interval domain to compute its result. Simon and King proposed a more precise wrap-around operator for polyhedra in [SK07]. It works by cutting polyhedra into pieces at wrap-around thresholds, folding each piece, and joining them. This method is able to infer affine relations preserved or induced by wrap-around. However, neither using the interval domain nor Simon and King's solution is precise enough to handle Ex. 5.1.1. Indeed, that example requires representing (at least locally) a non-convex set, which is not possible with polyhedra, whatever abstraction of wrap-around is used.

5.1.3 Modular intervals

We now propose a very simple variation on intervals in order to model $wrap_i^\sharp$ more precisely and handle Ex. 5.1.1. Following the ideas of Masdupuy [Mas93], we add a modular component to intervals. The *modular interval domain* is defined as:

$$\begin{array}{l} \mathcal{D}_m^\sharp \stackrel{\text{def}}{=} \{\,[l,h] + k\mathbb{Z} \mid l,h \in \mathbb{Z} \cup \{\pm\infty\}, k \in \mathbb{N}\,\} \\ \gamma_m([l,h]+k\mathbb{Z}) \stackrel{\text{def}}{=} \{\,x + ky \mid l \le x \le h,\ y \in \mathbb{Z}\,\} \ . \end{array} \tag{5.8}$$

Unlike intervals, this domain does not feature a best abstraction function α_m. We construct our abstract operators $\circ_m^\sharp$ as in Fig. 5.2, by mixing standard interval operations $\circ_i^\sharp$ and simple coset identities [Gra89]. For the arithmetic operators $+_m^\sharp$, $-_m^\sharp$, and $*_m^\sharp$ as well as the join $\cup_m^\sharp$ and the widening $\triangledown_m$, it is possible to derive a meaningful modular component form the component of the arguments. For other operators (such as $/_m^\sharp$), we revert to classic interval arithmetic with no modular component (i.e., $k = 0$). The most interesting operator is $wrap_m^\sharp([l,h]+k\mathbb{Z},[l',h'])$. When the modular interval folded by wrap-around to $[l',h']$ is an interval, it is directly returned. Otherwise, we return the argument $[l,h]+k\mathbb{Z}$ with an extra modular component $(h'-l'+1)\mathbb{Z}$ modeling the possible wrap-around. The ability, in the later case, to keep the bounds l and h of the original argument intact is key to precisely analyze "compute-through-overflow" programs, as shown below:

Example 5.1.3. We analyze Ex. 5.1.1 again, using modular intervals. Firstly, casting $[-1,1]$ to **unsigned char** gives:

$$wrap_m^\sharp([-1,1],[0,255]) = [-1,1] + 256\mathbb{Z} \ .$$

Hence, we represent exactly the fact that the result equals $[-1,1]$ wrapped-around. Unlike intervals, we cannot represent the fact that it is bounded in $[0,255]$, but, as it will turn out, this is not necessary. Then, adding twice this abstract element simply gives $[-2,2] + 256\mathbb{Z}$. Finally, the cast back to **signed char** gives:

$$wrap_m^\sharp([-2,2]+256\mathbb{Z},[-128,127]) = [-2,2]$$

which is the exact concrete result. It is sufficient to know that the final result is bounded in $[-128,127]$, due to the last cast, to recover the interval $[-2,2]$ from the modular interval $[-2,2]+256\mathbb{Z}$.
End of example.

Compared to the domain proposed originally by Masdupuy [Mas93], our domain is slightly less expressive. Indeed, the former infers interval congruences of the form $\theta\cdot[l,u]\langle m\rangle$ while, in our case, θ is fixed to 1. The main difference lies in the intended use and the design of abstract operators. Masdupuy's domain is designed to infer sets of array indexes encountered in loops, while our domain focuses on abstracting precisely wrap-around, and so, revolves around the definition of $wrap_m^\sharp$.

$$
\begin{array}{l}
-^\sharp_m([l,h]+k\mathbb{Z}) \stackrel{\text{def}}{=} [-h,-l]+k\mathbb{Z} \\
\sim^\sharp_m([l,h]+k\mathbb{Z}) \stackrel{\text{def}}{=} [-h-1,-l-1]+k\mathbb{Z} \\
([l_1,h_1]+k_1\mathbb{Z}) \circ^\sharp_m ([l_2,h_2]+k_2\mathbb{Z}) \stackrel{\text{def}}{=} \\
\quad \begin{cases} ([l_1,h_1] \circ^\sharp_i [l_2,h_2]) + \gcd(k_1,k_2)\mathbb{Z} & \text{if } \circ \in \{\, +, -, *, \cup, \triangledown \,\} \\ ([l_1,h_1] \circ^\sharp_i [l_2,h_2]) + 0\mathbb{Z} & \text{if } k_1 = k_2 = 0 \text{ and } \circ \in \{\, /, \%, \&, |, \hat{}, >>, << \,\} \\ [-\infty,+\infty] + 0\mathbb{Z} & \text{otherwise} \end{cases} \\
wrap^\sharp_m([l,h]+k\mathbb{Z},[l',h']) \stackrel{\text{def}}{=} \\
\quad \text{let } k' = \gcd(k,\, h'-l'+1) \text{ in} \\
\quad \begin{cases} [wrap(l,[l',h']),\, wrap(h,[l',h'])] + 0\mathbb{Z} & \text{if } (l'+k'\mathbb{Z}) \cap [l+1,h] = \emptyset \\ [l,h]+k'\mathbb{Z} & \text{otherwise} \end{cases}
\end{array}
$$

where gcd is the greatest common divisor, extended to $\gcd(0,x) = \gcd(x,0) = x$.

Figure 5.2: Abstract operators in the modular interval domain.

$$
\begin{array}{rcl}
[c_1,c_2]^\sharp_b & \stackrel{\text{def}}{=} & (\sim c_1, c_1) \qquad \textit{(when } c_1 = c_2\textit{)} \\
\sim^\sharp_b(z,o) & \stackrel{\text{def}}{=} & (o,z) \\
(z_1,o_1)\ \&^\sharp_b\ (z_2,o_2) & \stackrel{\text{def}}{=} & (z_1 \mid z_2,\ o_1 \,\&\, o_2) \\
(z_1,o_1)\ |^\sharp_b\ (z_2,o_2) & \stackrel{\text{def}}{=} & (z_1 \,\&\, z_2,\ o_1 \mid o_2) \\
(z_1,o_1)\ \hat{}^\sharp_b\ (z_2,o_2) & \stackrel{\text{def}}{=} & ((z_1 \,\&\, z_2) \mid (o_1 \,\&\, o_2),\ (z_1 \,\&\, o_2) \mid (o_1 \,\&\, z_2)) \\
(z_1,o_1)\ <<^\sharp_b\ (z_2,o_2) & \stackrel{\text{def}}{=} & ((z_1 << n) \mid ((1 << n) - 1),\ o_1 << n), \text{ when } \exists n \geq 0 :\ (z_2,o_2) = [n,n]^\sharp_b \\
(z_1,o_1)\ >>^\sharp_b\ (z_2,o_2) & \stackrel{\text{def}}{=} & (z_1 >> n,\ o_1 >> n), \text{ when } \exists n \geq 0 :\ (z_2,o_2) = [n,n]^\sharp_b \\
(z_1,o_1)\ \cup^\sharp_b\ (z_2,o_2) & \stackrel{\text{def}}{=} & (z_1 \mid z_2,\ o_1 \mid o_2) \\
(z_1,o_1)\ \triangledown^\sharp_b\ (z_2,o_2) & \stackrel{\text{def}}{=} & (z_1 \triangledown z_2,\ o_1 \triangledown o_2), \text{ with } x \triangledown y \stackrel{\text{def}}{=} \text{ if } x = x \mid y \text{ then } x \text{ else } -1 \\
wrap^\sharp_b((z,o),[0,2^n-1]) & \stackrel{\text{def}}{=} & (z \mid (-2^n),\ o \,\&\, (2^n-1)) \\
wrap^\sharp_b((z,o),[-2^n,2^n-1]) & \stackrel{\text{def}}{=} & ((z \,\&\, (2^n-1)) \mid (-2^n(p(z))_n),\ (o \,\&\, (2^n-1)) \mid (-2^n(p(o))_n))
\end{array}
$$

Figure 5.3: Abstract operators in the bit-field domain.

5.1.4 Bit-field domain

While it is precise on wrap-around, the modular interval domain is not well-adapted to bit-level operations, such as masking bits. For instance, $V \,\&\, 5$ always gives a result in $\{0,1,4,5\}$, which cannot be represented as an interval nor a modular interval. (We will see in Sec. 5.3.3 more realistic examples using bit masks, in the context of bit-level float manipulations.) A very natural solution is to complement interval domains with a domain that tracks the value of each bit independently. Such a non-relational bit-field domain has been proposed by Monniaux [Mon07] and Regehr et al. [RD06]. These domains abstract a concrete semantics of integers viewed as bit strings of fixed length. We now show that a bit-field domain can be constructed on a $\mathbb{Z}-$based semantics, lifting the restriction to fixed-length bit strings.

The *bit-field domain* $\mathcal{D}^\sharp_b \stackrel{\text{def}}{=} \mathbb{Z}\times\mathbb{Z}$ associates to each variable two integers, z and o, that represent the bit masks for bits that can be set respectively to 0 and to 1:

$$
\begin{array}{l}
\gamma_b(z,o) \stackrel{\text{def}}{=} \{\, b \mid \forall i \geq 0 :\ (\neg p(b)_i) \wedge p(z)_i \text{ or } p(b)_i \wedge p(o)_i \,\} \\
\alpha_b(S) \stackrel{\text{def}}{=} (\vee \{\, \neg p(b) \mid b \in S \,\},\ \vee \{\, p(b) \mid b \in S \,\})\ .
\end{array}
\tag{5.9}
$$

We have a Galois connection which allows defining optimal abstract operators. The most interesting ones are presented in Fig. 5.3. Note that we only handle the case of constants when they are singletons, and limit shifts to the case where the right argument is a positive singleton. In other cases, we can return the greatest element $\top^\sharp_b \stackrel{\text{def}}{=} (-1,-1)$ that represents $\mathcal{P}(\mathbb{Z})$. Wrapping around an unsigned interval $[0,2^n-1]$ is modeled by masking high bits, while wrapping around a signed interval $[-2^n,2^n-1]$ additionally performs a sign extension. As widening, we simply set all the bits in a bit mask (setting its value to -1) if it is not stable. Note that this domain is extremely easy to implement as the abstract operations can always be expressed in terms of concrete operations in $\mathbb{Z}$ (Fig. 5.1).

5.1.5 Discussion

The modular interval and bit-field domains were described in [Min12a], and developed as part of our work extending the Astrée static analyzer to larger classes of programs. Our goal was to analyze low-level programs that manipulate individual bits in integers, as well as automatically generated code (in the spirit of Ex. 5.1.3). The proposed domains are not intended to replace existing ones (such as plain intervals or relational domains), but to supplement them, through a reduced product, to gain some precision in very specific cases. Moreover, each domain is effectively tailored to specific coding practices. This fits very well the design by refinement of a specialized static analyzer, such as Astrée and AstréeA, which will be discussed in Chap. 6. The added domains are lightweight; indeed, they are non-relational and have a linear time cost. Our experience [Min12a] shows that adding them does not degrade the per-

formance of the analysis. They are moreover easy to design and to implement. This justifies the use of many specialized small domains instead of fewer large all-purpose ones. Future work includes the design of new support domains adapted to other programming idioms, should the need arise to improve the analysis on newly considered programs.

5.2 Structured types

Additionally to numeric types (such as machine integers and floats), the C language supports aggregate types (such as arrays and structures) as well as pointers. In this section, we assume that we are given abstract domains that support machine integers and floats (based on Secs. 5.1 and 2.4.4), and show how to extend them to a C-like type system (Sec. 5.2.1). After presenting a classic semantics for well-structured variable accesses (Sec. 5.2.2), we show its limitations and introduce a new, lower-level semantics (Sec. 5.2.3). That semantics models in precise details the memory representation of types in order to analyze precisely programs that rely on it.

5.2.1 Extended types

We extend our integer types (5.1) with the following type algebra, accounting for the main features of C:

$$\begin{array}{lcll} \textit{type} & ::= & \textit{scalar-type} & \\ & | & \textit{type}\,[n] & \textit{(arrays, } n \in \mathbb{N}\textit{)} \\ & | & \textbf{struct}\,\{\textit{type}_1, \ldots, \textit{type}_n\} & \textit{(structures)} \\ & | & \textbf{union}\,\{\textit{type}_1, \ldots, \textit{type}_n\} & \textit{(unions)} \end{array}$$
$$\textit{scalar-type} ::= \textit{int-type} \mid \textit{float-type} \mid \textbf{ptr}$$
$$\textit{float-type} ::= \textbf{float} \mid \textbf{double} \qquad (5.10)$$

Integers, floats, and pointers are collectively referred to as *scalar types*. Note that we use a single type, **ptr**, to represent all pointers, disregarding the kind of objects they are pointing to (it can be assimilated to C's `void*` type). Our language also allows structured types: arrays of fixed size, structures, and unions. Structures are aggregates storing a fixed collection of fields of heterogeneous types. Unions are also collections of fields, but all the fields occupy the same place in memory, so that only one of them can be stored at a given time (this results in a gain in memory, but can also be used for special effects as detailed in Sec. 5.2.3).

Machine integers and floats are both subsets of $\mathbb{R}$. Given a set of variables $\mathcal{V}$ of scalar type, we thus use as concrete domain $\mathcal{D}_{\mathcal{V}} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{V} \to \mathbb{R})$. We assume that we are given a family of abstract domains $\mathcal{D}^\sharp_{\mathcal{V}}$ abstracting $\mathcal{D}_{\mathcal{V}}$ for each $\mathcal{V}$, and we will lift this family to aggregate and pointer types.

5.2.2 Well-structured semantics

We first consider the simple case where only arrays and structures are used, but not pointers nor unions, which leads to a straightforward memory model. Most field-sensitive analyses follow this model, and this was also the case of early versions of the Astrée analyzer [BCC+03]. We present briefly this model mainly to show its limitations and motivate for the lower-level model we present next.

Cells. In order to retrieve a purely numeric semantics, we decompose recursively and statically aggregate variables into collections of scalar-typed fields. To distinguish between the original set of variables of type in *type* and the derived variables of type in *scalar-type*, we call the latter "*cells*." Given a variable $V \in \mathcal{V}$, each cell is a sequence of the form $V \cdot p$ where $p \in \mathbb{N}^*$ is a sequence of integers denoting structure field and array element selectors; p ranges in $\mathit{sel}(\mathit{type}(V))$ defined as follows, based on the type $\mathit{type}(V)$ of V:

$$\begin{array}{l} \mathit{sel}(t) \stackrel{\text{def}}{=} \varepsilon \quad \text{if } t \in \textit{scalar-type} \\ \mathit{sel}(t[n]) \stackrel{\text{def}}{=} \{\, i \cdot c \mid c \in \mathit{sel}(t),\, i \in [0, n-1] \,\} \\ \mathit{sel}(\textbf{struct}\,\{t_1, \ldots, t_n\}) \stackrel{\text{def}}{=} \{\, i \cdot c \mid c \in \mathit{sel}(t_i),\, i \in [1, n] \,\}\ . \end{array} \qquad (5.11)$$

We denote by *cell* the set of all cells in $\mathcal{V}$:

$$\mathit{cell} \stackrel{\text{def}}{=} \bigcup \{\, V \cdot p \mid V \in \mathcal{V},\, p \in \mathit{sel}(\mathit{type}(V)) \,\}\ . \qquad (5.12)$$

A set of memory states is then abstracted in a numeric domain $\mathcal{D}^\sharp_{\mathit{cell}}$ abstracting $\mathcal{P}(\mathit{cell} \to \mathbb{R})$.

Operators. The syntax of expressions is enriched in order to gain access to aggregate objects: plain variable accesses are replaced with accesses to lvalues *lval* (short for "left-value" as these also appear on the left of assignments) that are sequences of array and structure field accesses and denote assignable memory parts. The new syntax is shown in Fig. 5.4. The concrete semantics of assignments $\mathbb{S}[\![\ \mathit{lval} \leftarrow \mathit{expr}\ ]\!]$ and tests $\mathbb{S}[\![\ \mathit{expr} \bowtie 0\ ]\!]$ is modeled in two steps: lvalues are first replaced with cell sets (dynamic cell resolution), and then lvalue-less expressions are fed to the numeric domain $\mathcal{D}^\sharp_{\mathit{cell}}$. The first step involves evaluating (possibly recursively) expressions that appear as array indexes into value sets. The second step involves a slight generalization of expressions and statements semantics (Figs. 2.2, 2.6) to cell sets. Formally, we have:

$$\begin{array}{l} \mathbb{E}[\![\,\{\, X_1, \ldots, X_n \,\}\,]\!]\rho \stackrel{\text{def}}{=} \langle \{\, \rho(X_i) \mid i \in [1, n] \,\}, \emptyset \rangle \\ \mathbb{S}[\![\,\{\, X_1, \ldots, X_n \,\} \leftarrow e\,]\!]\langle R, O\rangle \stackrel{\text{def}}{=} \bigsqcup_i \mathbb{S}[\![\, X_i \leftarrow e\,]\!]\langle R, O\rangle\ . \end{array} \qquad (5.13)$$

An abstract semantics can be derived easily. The first step, index expression evaluation, can be performed, for instance, in the interval domain $\mathcal{D}^\sharp_i$, which additionally allows detecting array overflows. The second step involves extending the semantics of expressions and statements from Fig. 2.11 as follows:

$$\begin{array}{l} \mathbb{E}^\sharp_i[\![\,\{\, X_1, \ldots, X_n \,\}\,]\!]R^\sharp \stackrel{\text{def}}{=} \langle \bigcup^\sharp_i R^\sharp(X_i), \emptyset \rangle \\ \mathbb{S}^\sharp_i[\![\,\{\, X_1, \ldots, X_n \,\} \leftarrow e\,]\!]\langle R, O\rangle \stackrel{\text{def}}{=} \\ \qquad \text{let } \langle I, O'\rangle = \mathbb{E}^\sharp_i[\![\, e\,]\!]\, R^\sharp \text{ in} \\ \qquad \begin{cases} \langle R^\sharp[X_1 \mapsto I],\, O \cup O'\rangle & \text{if } n = 1 \\ \langle R^\sharp[\forall i :\, X_i \mapsto R^\sharp(X_i) \cup^\sharp_i I],\, O \cup O'\rangle & \text{otherwise}\ . \end{cases} \end{array} \qquad (5.14)$$

$$\begin{array}{lcll} \textit{expr} & ::= & \textit{lval} & \textit{(left-value)} \\ & | & [c_1, c_2] & \textit{(constant)} \\ & | & \circ_\omega\ \textit{expr} & \textit{(unary operator)} \\ & | & \textit{expr} \diamond_\omega \textit{expr} & \textit{(binary operator)} \\ \textit{lval} & ::= & V & \textit{(variable access, } V \in \mathcal{V}\textit{)} \\ & | & \textit{lval}\,.n & \textit{(field access, } n \in \mathbb{N}\textit{)} \\ & | & \textit{lval}\,[\,\textit{expr}\,]_\omega & \textit{(array access)} \\ \textit{stat} & ::= & \textit{lval} \leftarrow \textit{expr} & \textit{(assignment)} \end{array}$$

Figure 5.4: Syntax of expressions with structured types.

```
typedef unsigned char uint8;
typedef unsigned short uint16;
union {
    struct { uint8 al, ah, bl, bh; } b;
    struct { uint16 ax, bx; } w;
} regs;
ℓ1
regs.w.ax = 0x1234;ℓ2
if (!regs.b.ah) ℓ3 regs.b.bl = regs.b.al; ℓ4
else regs.b.bh = regs.b.al;ℓ5
ℓ6regs.b.al = 0xab;ℓ7
// here regs.w.ax == 0x12ab
```

Figure 5.5: Non-portable use of C unions.

This definition resorts to *weak updates* when assigning into a non-singleton set of cells. The case of relational domains, such as polyhedra and octagons, is only slightly more complicated. We refer the reader to the work of Gopal et al. on array summarization [GDD+04], which provides example implementations for weak updates.

Extensions. The C standard [ISO07] defines the semantics of union types only in the case where a single field is active in a given environment: the program can only read back from a union using the same field used for the preceding write. Other accesses are undefined. It is possible to extend the previous semantics to support unions if we limit ourselves to the usage allowed by the standard. A union is modeled as a structure, plus an extra information tracking which field is active. We can then abstract a set of environments as a numeric abstract element, plus a map from unions to the set of possible active elements.

Moreover, the only well-defined use of pointers in the C standard corresponds to navigating within arrays. Such pointer uses can be modeled in our semantics by associating to each pointer a numeric variable tracking its index, and keeping a map from pointers to the sets of arrays they can point to.

Remark. Another standard way to support pointers in analyses is to resolve all pointer dereferences in a separate pass before the numeric analysis, using one of the many existing points-to analyses (we refer the reader to the survey by Hind [Hin01] for more information on points-to analyses). However, there is experimental evidence [PH99] that combined points-to and numeric analyses are more precise than separate ones. In abstract interpretation, this is related to the well-known fact that a reduced product of abstract domains is more precise than their product without reduction (Sec. 2.2). Additionally, an analysis expressed as a combined pointer and numeric analysis benefits directly from numeric domains to abstract pointer arithmetic. *End of remark.*

Limitations. While attractive due to their simplicity, these extensions are of limited use in practice as many C programs abuse unions and pointers. Such programs rely on behaviors that are unspecified or undefined by the standard, but nevertheless ensured by a specific compiler on a given architecture, and yet, cannot be handled easily in a well-structured memory abstraction. Figure 5.5 presents a simple C program that mixes accesses to different fields of a union. It relies on the fact that, when run on the correct architecture (here, an ia32 architecture), after writing 0x1234 into the field `ax`, the high order byte of its representation, 0x12, can be retrieved from `ah`. Figure 5.6 achieves a similar result, but uses pointer casts. By converting a pointer on 16-bit integers to a pointer on 8-bit integers and dereferencing it, the individual bytes of its representation can be accessed. This bypassing of the language type system is often referred to as "*type punning*." These two examples show that statically decomposing the memory into disjoint parts assigned to independent cells is not always possible: one must consider possibly overlapping views of the memory. In the first example, the set of possible views can be derived from the static type information, i.e., from the type of the union fields. In the second example, the type of the variable does not give any insight on the possible ways it may be used: it is intricately tied to the dynamic value of the pointers when the cast is executed.

```
uint16 ax, bx;
ax = 0x1234;
if (*(((uint8*)&ax)+1))
    *((uint8*)&bx) = *((uint8*)&ax);
else
    *(((uint8*)&bx)+1) = *((uint8*)&ax);
*((uint8*)&ax) = 0xab;
```

Figure 5.6: Non-portable use of pointer casts in C.

$$sizeof(\mathbf{int}) \stackrel{\text{def}}{=} sizeof(\mathbf{long}) \stackrel{\text{def}}{=} sizeof(\mathbf{ptr}) \stackrel{\text{def}}{=} 4$$
$$sizeof(\mathbf{char}) \stackrel{\text{def}}{=} 1 \qquad sizeof(\mathbf{short}) \stackrel{\text{def}}{=} 2$$
$$sizeof(\mathbf{float}) \stackrel{\text{def}}{=} 4 \qquad sizeof(\mathbf{double}) \stackrel{\text{def}}{=} 8$$

if $t \in$ *scalar-type*, then $alignof(t) \stackrel{\text{def}}{=} sizeof(t)$

if $t = t'[n]$, then
$\quad alignof(t) \stackrel{\text{def}}{=} alignof(t')$
$\quad sizeof(t) \stackrel{\text{def}}{=} n \times sizeof(t')$

if $t = \mathbf{struct}\ \{\ t_1, \dots\ t_n\}$, then
$\quad alignof(t) \stackrel{\text{def}}{=} \text{lcm}\ \{\ alignof(t_i) \mid i \in [1,n]\ \}$
$\quad offset(t,1) \stackrel{\text{def}}{=} 0$
$\quad offset(t,i+1) \stackrel{\text{def}}{=} align(offset(t,i) + sizeof(t_i), t_{i+1})$
$\quad sizeof(t) \stackrel{\text{def}}{=} align(offset(t,n) + sizeof(t_n), t)$

if $t = \mathbf{union}\ \{\ t_1, \dots\ t_n\}$, then
$\quad alignof(t) \stackrel{\text{def}}{=} \text{lcm}\ \{\ alignof(t_i) \mid i \in [1,n]\ \}$
$\quad offset(t,i) \stackrel{\text{def}}{=} 0$
$\quad sizeof(t) \stackrel{\text{def}}{=} align(\max\ \{\ sizeof(t_i) \mid i \in [1,n]\ \}, t)$

where $align(o,t) \stackrel{\text{def}}{=} \min\ \{\ x \in (alignof(t))\mathbb{Z} \mid x \geq o\ \}$
and lcm is the least common multiple.

Figure 5.7: System V ABI for a 32-bit architecture.

5.2.3 Low-level semantics

We now present a semantics that supports union types and pointers, and is able to model precisely the examples in Figs. 5.5 and 5.6. Our concrete semantics is based on a low-level byte-based representation of C variables.

expr	::=	*lval*	*(left-value)*
	|	$\&V$	*(variable address, $V \in \mathcal{V}$)*
	|	$[c_1, c_2]$	*(constant)*
	|	$\circ_\omega$ *expr*	*(unary operator)*
	|	*expr* $\diamond_\omega$ *expr*	*(binary operator)*
lval	::=	$*_{\textit{scalar-type},\omega}$ *expr*	*(dereference)*
stat	::=	*lval* $\leftarrow$ *expr*	*(expression assignment)*

Figure 5.8: Syntax of low-level expressions.

Layout. A first step is to make explicit all the assumptions on the layout of data in memory that a program may rely on:
- the byte size $\textit{sizeof}(t)$ of each type $t \in \textit{type}$;
- the offset $\textit{offset}(t, i)$ of the $i-$th field of a structure of type t, i.e., the position in bytes of the first byte of the field relative to the first byte of the structure;
- the alignment $\textit{alignof}(t)$ of a type, which imposes constraints on the offset of structure fields with this type; alignment is enforced by inserting padding bytes between fields and at the end of structures;
- the ordering of bytes in scalar types (either least significant or most significant first).

Hence, the analysis is parametrized by these choices, and will only be sound for the chosen parameter instance. These choices are generally documented in an ABI (Application Binary Interfaces). Figure 5.7 provides an example definition: it uses a general algorithm defined in the System V ABI [ATSCOI97] to derive the sizes, alignments, and offsets in a systematic way by induction on types, based on the size and alignment of scalar types. The example definition in Fig. 5.7 matches a standard 32-bit architecture.

Expressions. Our expressions, presented in Fig. 5.8, now support pointers and *pointer arithmetic* (overloading the operators $+, - \in \diamond$). Most of the syntax of lvalues has disappeared. Indeed, they can be encoded in terms of pointer arithmetic and dereferences. More precisely, an lvalue l of type t appearing in an expression is replaced with $*_{t,\omega}(\&l)$, and the & operator is "pushed inside" using the following rules:

$$\begin{aligned} &\&(l.f) \;\rightsquigarrow\; \&l + \textit{offset}(t, f) \\ &\&(l[e]_\omega) \;\rightsquigarrow\; \&l + \textit{sizeof}(t') \times e \quad \text{where } t = t'[n] \;. \end{aligned}$$

Note that pointer arithmetic is expressed as offset arithmetic, at the byte level. Note also that our expressions can only return a scalar value, and we support only assignments of scalars. The only operation supported on non-scalar objects in C is the copy assignment, which we can statically convert into a set of scalar assignments (either field by field or byte by byte).

Pointers. In our language, valid pointers can only be constructed by taking the address of a variable and performing pointer arithmetic. We restrict our analysis to the case of flat memory models, in which addresses, and thus pointers, are plain integers.[2] However, no assumption can be made on the base address of variables, which can change each time the variable is recreated (for local variables) or the program is run again. Hence, we model *pointer values* as (semi-)symbolic addresses of the form $\langle V, i\rangle \in \mathcal{V} \times \mathbb{Z}$, which indicates an offset of i bytes from the first byte of V. This choice is quite standard for C analyses that take pointer arithmetic into account (see for instance [WL95] for an early example). We must add a special pointer value, **NULL**, to model C's `NULL` pointer. Moreover, we add a special value, **invalid**, denoting pointers that are not obtained by taking the address of any variable (e.g., constructed by converting from an integer or a float value) and can thus point anywhere. The set of pointer values is then:

$$\mathcal{P}tr \stackrel{\text{def}}{=} (\mathcal{V} \times \mathbb{Z}) \cup \{\mathbf{NULL}, \mathbf{invalid}\} \;. \tag{5.15}$$

An important subset of $\mathcal{P}tr$ is the subset $\mathcal{A}ddr$ of pointers to *addressable memory* bytes:

$$\begin{aligned} \mathcal{A}ddr &\stackrel{\text{def}}{=} \bigcup \{\, \langle V, o\rangle \mid V \in \mathcal{V} \wedge o \in [0, \textit{sizeof}(\textit{type}(V)) - 1] \,\} \\ &\subseteq \mathcal{P}tr \end{aligned} \tag{5.16}$$

where $\textit{type}(V)$ denotes the type of the variable $V \in \mathcal{V}$. Depending on the actual base address of variables, dereferencing a pointer outside $\mathcal{A}ddr$ may actually access a valid memory region (inside another variable) or cause a non-deterministic error. In order to enforce the (desirable) property that the program semantics does not depend on the base addresses and to always consider the worst possible scenario, we consider that it is a run-time error to access bytes outside $\mathcal{A}ddr$. However, we allow constructing pointers pointing outside $\mathcal{A}ddr$, as long as they are not dereferenced. It also becomes possible, in our semantics, to start from a pointer to a field of a structure or union and construct, by pointer arithmetic, a pointer to another field of the same variable and dereference it. Note that all these operations are undefined in the C standard [ISO07]; our semantics is thus laxer. While this means that more programs can be given a semantics, it also means that the analysis will report fewer kinds of errors than the well-structured semantics (i.e., violations of the standard that we now accept). In our experience, the choice of the concrete semantics is a trade-off and can vary depending on the kind of programs and properties one wishes to analyze.

Pointer arithmetic is straightforward and reduces to integer arithmetic on offsets. For instance, adding an integer i to a pointer p gives:

$$p +_p i \stackrel{\text{def}}{=} \begin{cases} \langle V, o + i\rangle & \text{if } p = \langle V, o\rangle \in \mathcal{V} \times \mathbb{Z} \\ \mathbf{NULL} & \text{if } p = \mathbf{NULL} \wedge i = 0 \\ \mathbf{invalid} & \text{otherwise} \;. \end{cases} \tag{5.17}$$

We must however be careful that physically distinct symbolic pointers may represent the same address and compare equal. In fact, only distinct pointers to addressable bytes as well as **NULL** are guaranteed to be distinct. For instance, if V and W are 4-byte integers, then the pointers $\langle V, 4\rangle$ and $\langle W, 0\rangle$ may be equal if V and W are allocated at contiguous addresses. This leads to the following definition of equality $=_p$:

$$\begin{aligned} p =_p p' \stackrel{\text{def}}{=} &\{\, \mathit{true} \mid p = p' \vee \{p, p'\} \not\subseteq \mathcal{A}ddr \cup \{\mathbf{NULL}\} \,\} \cup \\ &\{\, \mathit{false} \mid p \neq p' \vee \{p, p'\} \not\subseteq \mathcal{A}ddr \cup \{\mathbf{NULL}\} \,\} \end{aligned} \tag{5.18}$$

which returns $\{\mathit{true}, \mathit{false}\}$ when comparing $\langle V, 4\rangle$ and $\langle W, 0\rangle$ as, depending on the addresses chosen by the compiler, they may compare equal or not.

[2] We thus ignore here the case of segmented architectures.

Byte-based memory model

Our modeling of memory accesses as performing a byte-based address computation and then dereferencing some data at this location suggests modeling also the contents of the memory at the byte level. In the computer, each byte has a value in $[0, 255]$ but, to account for our symbolic pointers, we enrich byte values with pairs $\langle p,\, i\rangle \in \mathcal{P}tr \times \mathbb{N}$ denoting the $i-$th byte in the memory representation of the pointer value p. Hence *byte values* are:

$$\mathbb{B} \stackrel{\text{def}}{=} [0, 255] \cup (\mathcal{P}tr \times \mathbb{N}) \ . \tag{5.19}$$

An environment is now an element of $\mathcal{E} \stackrel{\text{def}}{=} \mathcal{A}ddr \to \mathbb{B}$.

Expressions manipulate *scalar values*, which may be numeric (machine integers or floats) or pointer values. We denote the set of values as $\mathbb{V}$:

$$\mathbb{V} \stackrel{\text{def}}{=} \mathbb{R} \cup \mathcal{P}tr \ . \tag{5.20}$$

The final component required to define our most concrete semantics is a *representation function* $benc_t$ that converts a scalar value of a given type t into a sequence of $sizeof(t)$ byte values, and the conversion back $bdec_t$. The conversion is parametrized by the type, which defines a binary representation of scalars (for instance, the scalar 1 has a different byte encoding when seen as an integer and as a float). An example definition, corresponding to a 32-bit Intel (i.e., little endian) architecture, is presented in Fig. 5.9 (the case of floats is omitted here; it will be handled in Sec. 5.3). Note that the mapping between byte and scalar values is not unique; hence, the functions $benc_t$ and $bdec_t$ are non-deterministic (they output a set of possible values). For instance, when decoding with integer type some bytes representing symbolic pointers, the whole range of integers is returned.

The functions $benc_t$ and $bdec_t$ are used in Fig. 5.10 to give a meaning to pointer dereferences: bytes are fetched and decoded with $bdec_t$ when reading from the memory in an expression $\mathbb{E}[\![*_{t,\omega}\ e]\!]$, while values computed by expressions are encoded to bytes with $benc_t$ when written into the memory in an assignment $\mathbb{S}[\![*_{t,\omega}\ e_1 \leftarrow e_2]\!]$. The semantics also reports illegal memory accesses (i.e., dereferencing non-addressable bytes) as errors at location ω.

Example 5.2.1. The semantics gives its intended meaning to our examples from Figs. 5.5 and 5.6. As another example, consider a variable V of type **unsigned int**. Then, writing a value v into V and then reading it back with $*_{\textbf{signed int}}\ \&V$ reduces to evaluating $(bdec_{\textbf{signed int}} \circ benc_{\textbf{unsigned int}})(v)$. Given our definitions of $benc_t$ and $bdec_t$, it turns out to have the exact same semantics as a regular integer cast: **(signed int)**v.
End of example.

5.2.4 Cell-based memory model

The byte-based concrete semantics is quite attractive as it can precisely model the memory and yet all the computations actually performed by expressions are expressed using only scalars in $\mathbb{V}$, i.e., mathematical integers and reals (as opposed to bit blasting). However, abstracting this semantics directly in a numeric domain is not advisable. Firstly, it requires domains to abstract two kinds of values: bytes in $\mathbb{B}$ (to model environments) and scalars in $\mathbb{V}$ (to model expression values). A more severe problem is that the concrete operators (Fig. 5.10) rely heavily on systematic conversions between the two kinds of values, and so, the conversions must be precisely approximated in the abstract domain. At the very least, we would expect that reading a value with the same type as it was written last in the memory gives back the exact same value, i.e., that $benc_t^\sharp \circ bdec_t^\sharp = \lambda X.\, X$. This would require relational domains able to reason precisely on the linear equalities and the disjunctions appearing in $bdec_t$ and $benc_t$. This imposes a heavy burden on the abstract domain and prevents the use of the most scalable ones, such as intervals.

Concrete semantics

We propose instead to abstract a slightly less concrete semantics, that reasons at the level of cells and scalars only (and not byte values). The gist of the method, which we introduced first in [Min06a], is to decompose the memory into a set of possibly overlapping cells, that evolves dynamically during the analysis.

We first consider the (finite) universe $\mathcal{C}ell$ of cells that may be dereferenced. Each cell is denoted, similarly to pointers, as a variable V and an offset o, to which we add a scalar type t indicating an encoding of values:

$$\begin{aligned}\mathcal{C}ell \stackrel{\text{def}}{=} \{\, \langle V,\, o,\, t\rangle \mid V \in \mathcal{V},\, t \in \textit{scalar-type},\\ 0 \le o \le sizeof(type(V)) - sizeof(t) \,\} \ .\end{aligned} \tag{5.21}$$

By construction, all the bytes in a cell are addressable: we have $\langle V,\, o\rangle, \ldots \langle V,\, o + sizeof(t) - 1\rangle \in \mathcal{A}ddr$. Our domain of environments, denoted as $\mathcal{E}^\flat$, is modeled as a choice of a cell set $C \subseteq \mathcal{C}ell$ and a set of scalar environments on C:

$$\mathcal{E}^\flat \stackrel{\text{def}}{=} \bigcup_{C \subseteq \mathcal{C}ell} \{\, \langle C,\, R\rangle \mid R \in \mathcal{P}(C \to \mathbb{V}) \,\} \ . \tag{5.22}$$

Note that an address in $\mathcal{A}ddr$ may be covered by several cells, or none at all. To give a meaning to such environments, we use a *conjunctive semantics*: if a concrete element provides several information on a given byte, then they must be simultaneously true. Hence, a concrete element $\langle C,\, R\rangle \in \mathcal{E}^\flat$ represents the following set $\gamma_{\mathcal{C}ell}\langle C,\, R\rangle \in \mathcal{P}(\mathcal{A}ddr \to \mathbb{V})$ of byte-level memories:

$$\begin{aligned}\gamma_{\mathcal{C}ell}\langle C,\, R\rangle \stackrel{\text{def}}{=} \{\, \rho \in \mathcal{A}ddr \to \mathbb{V} \mid \exists r \in R : \forall \langle V,\, o,\, t\rangle \in C :\\ \exists (b_0, \ldots, b_{n-1}) \in benc_t(r\langle V,\, o,\, t\rangle) :\\ \forall i < sizeof(t) :\ \rho\langle V,\, o + i\rangle = b_i \,\} \ .\end{aligned} \tag{5.23}$$

The use of a conjunctive semantics matches the intuition that union types and type punning may give access to several typed views of the same underlying sequences of bytes, and that all these views are valid simultaneously.

Two key operations in our domain are cell addition and cell removal. Due to our intersection semantics, it is sound to remove any cell: it corresponds to removing information. Formally, we have: $\gamma_{\mathcal{C}ell}\langle C,\, R\rangle \subseteq \gamma_{\mathcal{C}ell}\langle C \setminus D,\, R_{|_{C \setminus D}}\rangle$. It is also possible to add new cells, as long as we are careful to initialize their value according to the constraints imposed by existing cells overlapping them. We use a *value synthesize function* $\phi \in \mathcal{C}ell \to \mathcal{P}(\mathcal{C}ell) \to expr$ such that $\phi(c)(C)$ returns a syntactic expression denoting (an abstraction of) the value of the cell c as a function of cells in C. An example implementation is proposed in Fig. 5.11. Firstly, if the cell already exists ($c \in C$), it is directly returned. Secondly, it converts between integers of the same size and different signedness using the *wrap* function from (5.4). Thirdly, it extracts unsigned bytes from integers, and aggregates unsigned bytes into integers. When all fails, it

$\underline{benc_{\textit{scalar-type}} \in \mathbb{V} \to \mathcal{P}(\mathbb{B}^*)}$

if $t \in$ *int-type* and t is unsigned, then:

$$benc_t(v) \stackrel{\text{def}}{=} \{\, (b_0, \ldots, b_{n-1}) \,\} \text{ where } \forall i < n :\ b_i \in [0, 255] \wedge \textstyle\sum_{i=0}^{n-1} 2^{8\times i} b_i = v$$

if $t \in$ *int-type* and t is signed, then:

$$benc_t(v) \stackrel{\text{def}}{=} \{\, (b_0, \ldots, b_{n-1}) \,\} \text{ where } \forall i < n :\ b_i \in [0, 255] \wedge \textstyle\sum_{i=0}^{n-1} 2^{8\times i} b_i = \begin{cases} v & \text{if } v \geq 0 \\ v + 2^{8\times n} & \text{if } v < 0 \end{cases}$$

$$benc_{\mathbf{ptr}}(v) \stackrel{\text{def}}{=} \{\, (\langle v,\, 0\rangle, \ldots, \langle v,\, n-1\rangle) \,\}$$

$\underline{bdec_{\textit{scalar-type}} \in \mathbb{B}^* \to \mathcal{P}(\mathbb{V})}$

if $t \in$ *int-type* and t is unsigned, then:

$$bdec_t(b_0, \ldots, b_{n-1}) \stackrel{\text{def}}{=} \begin{cases} \{\, x \,\} & \text{if } \forall i < n :\ b_i \in [0, 255] \wedge x = \textstyle\sum_{i=0}^{n-1} 2^{8\times i} b_i \\ \mathit{range}(t) & \text{otherwise} \end{cases}$$

if $t \in$ *int-type* and t is signed, then:

$$bdec_t(b_0, \ldots, b_{n-1}) \stackrel{\text{def}}{=} \begin{cases} \{\, x \,\} & \text{if } \forall i < n :\ b_i \in [0, 255] \wedge x = \textstyle\sum_{i=0}^{n-1} 2^{8\times i} b_i < 2^{8\times n-1} \\ \{\, x - 2^{8\times n} \,\} & \text{if } \forall i < n :\ b_i \in [0, 255] \wedge x = \textstyle\sum_{i=0}^{n-1} 2^{8\times i} b_i \geq 2^{8\times n-1} \\ \mathit{range}(t) & \text{otherwise} \end{cases}$$

$$bdec_{\mathbf{ptr}}(b_0, \ldots, b_{n-1}) \stackrel{\text{def}}{=} \begin{cases} \{p\} & \text{if } \forall i < n :\ b_i = \langle p,\, i\rangle \\ \{\mathbf{invalid}\} & \text{otherwise} \end{cases}$$

where $n = \mathit{sizeof}(t)$

Figure 5.9: Byte-encoding and decoding of scalars.

$$\mathbb{E}[\![\, *_{t,\omega}\ e_1 \,]\!]\rho \stackrel{\text{def}}{=} \langle \textstyle\bigcup \{\, bdec_t(\rho(v), \ldots, \rho(v +_p (n-1))) \mid v \in V_1^\rho,\ \forall i < n :\ v +_p i \in \mathcal{A}ddr \,\},\ O_1^\rho \cup \{\, \omega \mid \exists v \in V_1^\rho,\ i < n :\ v +_p i \notin \mathcal{A}ddr \,\} \rangle$$

$$\mathbb{S}[\![\, *_{t,\omega} e_1 \leftarrow e_2 \,]\!]\langle R,\, O\rangle \stackrel{\text{def}}{=} \langle \emptyset,\, O\rangle \sqcup \textstyle\bigsqcup_{\rho \in R} \langle \{\, \rho[\forall i < n :\ v_1 +_p i \mapsto b_i] \mid v_1 \in V_1^\rho,\ v_2 \in V_2^\rho,\ (b_0, \ldots, b_{n-1}) \in benc_t(v_2) \,\},\ O_1^\rho \cup O_2^\rho \cup \{\, \omega \mid \exists v \in V_1^\rho,\ i < n :\ v +_p i \notin \mathcal{A}ddr \,\} \rangle$$

where $\langle V_1^\rho,\, O_1^\rho\rangle \stackrel{\text{def}}{=} \mathbb{E}[\![\, e_1 \,]\!]\rho$, $\langle V_2^\rho,\, O_2^\rho\rangle \stackrel{\text{def}}{=} \mathbb{E}[\![\, e_2 \,]\!]\rho$, and $n = \mathit{sizeof}(t)$

Figure 5.10: Memory reads and write in the byte-based semantics.

returns the full range of the type (or **invalid**, for a pointer). Cell addition, $\textit{add-cell} : \mathcal{C}ell \to \mathcal{E}^\flat \to \mathcal{E}^\flat$, then simply adds the cell and initializes its value using the ϕ function:

$$\textit{add-cell}(c)\langle C,\, R\rangle \stackrel{\text{def}}{=} \langle C \cup \{c\},\, \{\, \rho[c \mapsto v] \mid \rho \in R,\, v \in \mathit{fst}(\mathbb{E}[\![\, \phi(c)(C) \,]\!]\rho) \,\} \rangle \ . \tag{5.24}$$

A cell can sometimes be synthesized in several ways (for instance, when synthesizing a byte which is overlapped by several integer cells); in this case, we consider all the possible choices and intersect their result. There is much freedom in designing ϕ, and it is possible to refine it by adding more cases (for instance, Sec. 5.3 will add float synthesis). The only requirement is to obey the following simple soundness condition, stating that *add-cell* over-approximates the identity function:

$$\forall \langle C,\, R\rangle \in \mathcal{E}^\flat,\ c \in \mathcal{C}ell :\ \gamma_{\mathcal{C}ell}(\textit{add-cell}(c)\langle C,\, R\rangle) \supseteq \gamma_{\mathcal{C}ell}\langle C,\, R\rangle \ . \tag{5.25}$$

The converse inclusion naturally holds as $\textit{add-cell}(c)\langle C,\, R\rangle$ has more cells than $\langle C,\, R\rangle$, and so, more constraints; in practice, we thus have an equality in (5.25).

We are now ready to present our concrete cell-based assignments and tests. Similarly to the well-structured semantics, expressions are first transformed into purely scalar expressions by resolving lvalues bottom up. More precisely, any lvalue $*_{t,\omega}\ e$ where e does not contain any dereference is transformed into a cell set by:

- evaluating e into a set of values V and of errors O;
- gathering the cells L corresponding to valid pointers in V: $L \stackrel{\text{def}}{=} \{\, \langle V,\, o,\, t\rangle \mid \langle V,\, o\rangle \in V \wedge \forall i < \mathit{sizeof}(t) :\ \langle V,\, o + i\rangle \in \mathcal{A}ddr \,\}$;
- realizing all the cells in L using *add-cell*.

The returned cell set is L, while the returned error set is O with the possible addition of ω in case some value in V does not correspond to a valid pointer to a cell. The semantics of cell sets $\mathbb{E}[\![\, \{\, X_1, \ldots, X_n \,\} \,]\!]$ appearing in expressions is straightforward: it is the same as in (5.13). The semantics of assignments $\mathbb{S}[\![\, \{\, X_1, \ldots, X_n \,\} \leftarrow e \,]\!]$ is slightly more complicated. Because of our conjunctive semantics, it is not sufficient to update the value of $X_1, \ldots, X_n$ as in (5.13); it is also necessary to update the value of all the cells that overlap $X_1, \ldots, X_n$, which may be complex and costly. We propose an efficient alternate solution: we simply remove all the cells that overlap $X_1, \ldots, X_n$ (these cells can always be created again when needed, i.e., when they are the target of a read or a write).

In addition to assignments and tests, we require a final

$$\phi\langle V,\, o,\, t\rangle(C) \stackrel{\text{def}}{=} \begin{cases} \langle V,\, o,\, t\rangle \\ \quad \text{if } \langle V,\, o,\, t\rangle \in C \\ \mathit{wrap}(\langle V,\, o,\, t'\rangle, \mathit{range}(t)) \\ \quad \text{else if } \langle V,\, o,\, t'\rangle \in C \wedge t, t' \in \mathit{int\text{-}type} \wedge \mathit{sizeof}(t) = \mathit{sizeof}(t') \\ (\langle V,\, o-b,\, t'\rangle / 2^{8b}) \mod 256 \\ \quad \text{else if } \langle V,\, o-b,\, t'\rangle \in C \wedge t = \textbf{unsigned char} \wedge t' \in \mathit{int\text{-}type} \wedge b < \mathit{sizeof}(t') \\ \mathit{wrap}(\sum_{i=0}^{n-1} 2^{8i} \times \langle V,\, o+i,\, t'\rangle, \mathit{range}(t)) \\ \quad \text{else if } n = \mathit{sizeof}(t) \wedge \forall i < n : \langle V,\, o+i,\, t'\rangle \in C \wedge \\ \quad t \in \mathit{int\text{-}type} \wedge t' = \textbf{unsigned char} \\ \mathit{range}(t) \\ \quad \text{else if } t \in \mathit{scalar\text{-}type} \\ \textbf{invalid} \\ \quad \text{else if } t = \textbf{ptr} \end{cases}$$

Figure 5.11: Cell synthesize function.

concrete operator: the join. It must now merge environment sets defined on heterogeneous cell sets. Given two concrete elements, $\langle C_1,\, R_1\rangle$ and $\langle C_2,\, R_2\rangle$, we first unify the cell sets into $C \stackrel{\text{def}}{=} C_1 \cup C_2$ by adding, with *add-cell*, in R_1 and R_2, the missing cells (respectively $C \setminus C_1$ and $C \setminus C_2$) to obtain the elements $\langle C,\, R_1'\rangle$ and $\langle C,\, R_2'\rangle$. The result of the join is then $\langle C,\, R_1' \cup R_2'\rangle$.

Abstract semantics

As for the well-structured semantics, it is straightforward to abstract our concrete cell-based semantics using an arbitrary numeric abstract domain. We assume that we are given, for each possible cell set $C \subseteq \mathcal{C}ell$, an abstract domain $\mathcal{D}^\sharp_C$, with concretization γ_C; it abstracts $\mathcal{P}(C \to \mathbb{R}) \simeq \mathcal{P}(\mathbb{R}^{|C|})$, i.e., sets of points in a $|C|-$dimensional vector space. A cell of integer or float type naturally corresponds to a dimension in an abstract element. We also associate a distinct dimension to each cell with pointer type; it corresponds to the offset o of a symbolic pointer $\langle V,\, o\rangle \in \mathcal{P}tr$. In order to abstract fully pointer values, we enrich abstract environments with a map P associating to each pointer cell the set of variables it may point to (i.e., the V components in $\langle V,\, o\rangle$) which we call the *pointer base*. The base additionally expresses whether the pointer may be **NULL** or **invalid**. Hence, the abstract domain becomes:[3]

$$\mathcal{D}^\sharp_{mem} \stackrel{\text{def}}{=} \{\, \langle C,\, R^\sharp,\, P\rangle \mid C \subseteq \mathcal{C}ell,\, R^\sharp \in \mathcal{D}^\sharp_C,\, P \in \{\, \langle c,\, o,\, \textbf{ptr}\rangle \in C \,\} \to (\mathcal{V} \cup \{\, \textbf{NULL}, \textbf{invalid} \,\}) \,\} \tag{5.26}$$

and the concretization is:

$$\gamma_{mem}\langle C,\, R^\sharp,\, P\rangle \stackrel{\text{def}}{=} \langle C,\, \{\, \rho' \mid \exists \rho \in \gamma_C(R^\sharp) : \forall c = \langle V,\, o,\, t\rangle \in C : \begin{cases} \rho'(c) = \rho(c) & \text{if } t \neq \textbf{ptr} \\ \rho'(c) = \langle p,\, \rho(c)\rangle & \text{if } t = \textbf{ptr} \wedge p \in P(c) \cap \mathcal{V} \\ \rho'(c) = p & \text{if } t = \textbf{ptr} \wedge p \in P(c) \setminus \mathcal{V} \end{cases} \,\}\rangle \ . \tag{5.27}$$

Recall that the cell-based concrete semantics reused the classic numeric concrete semantics of Sec. 2.3; likewise, the cell-based abstract operators can be derived from the classic numeric ones we presented in Sec. 2.3.6. In particular, cell addition can be expressed as adding a new variable and initializing it with an abstract assignment, as: $\mathbb{S}^\sharp[\![\, c \leftarrow \phi(c)(C) \,]\!]$, and lvalue resolution methods can reduce the expressions occurring in any assignment or test to expressions without dereference (in some cases, leading to weak updates, as in (5.14)). Pointer expressions are handled by firstly computing the set of possible bases (which is straightforward as the bases are stored in extension in the P component of abstract elements), and constructing a numeric expression expressing the pointer offset, which can be fed to the underlying numeric domain. The join $\cup^\sharp$ reduces, after unifying the cell sets of both arguments, to a join in the numeric abstract domain and an element-wise join of sets of pointer bases. The widening $\triangledown$ is constructed the same way, but uses the underlying numeric widening instead of the join (while pointer base sets are still joined with unions). This is indeed sufficient to enforce the termination because the cell sets are subsets of the cell universe $\mathcal{C}ell$ which is finite, and the set of pointer bases $\mathcal{V} \cup \{\textbf{NULL}, \textbf{invalid}\}$ is also finite.

We do not present formally all these operators here, as they are straightforward; we only illustrate some of them on an example and refer the reader to [Min06a] for more information.

Example 5.2.2. Consider the program in Fig. 5.5 using a union type. We present in Fig. 5.12 the dynamic evolution of the cell set during an abstract analysis:

1. when the program starts, the cell set is empty;
2. the assignment `regs.w.ax = 0x1234` creates a new cell $c_1 \stackrel{\text{def}}{=} \langle \texttt{regs},\, 0,\, \textbf{uint16}\rangle$ initialized to $[0, 65535]$, and issues an assignment $\mathbb{S}^\sharp[\![\, c_1 \leftarrow \texttt{0x1234} \,]\!]$;
3. the test on `regs.b.ah` then creates another cell, $c_2 \stackrel{\text{def}}{=} \langle \texttt{regs},\, 1,\, \textbf{uint8}\rangle$, which is initialized by ϕ by the assignment $\mathbb{S}^\sharp[\![\, c_2 \leftarrow (c_1/256) \mod 256 \,]\!]$; the cell c_2 is then used in the abstract tests $\mathbb{S}^\sharp[\![\, c_2 = 0 \,]\!]$ and $\mathbb{S}^\sharp[\![\, c_2 \neq 0 \,]\!]$;

4–5. both the then and else branches create the cell $c_3 \stackrel{\text{def}}{=} \langle \texttt{regs},\, 0,\, \textbf{uint8}\rangle$ for `regs.b.al`; the then branch creates the cell $c_4 \stackrel{\text{def}}{=} \langle \texttt{regs},\, 2,\, \textbf{uint8}\rangle$ for `regs.b.bl`, and the else branch creates the cell $c_5 \stackrel{\text{def}}{=} \langle \texttt{regs},\, 3,\, \textbf{uint8}\rangle$ for `regs.b.bh`;

6. the join after the branches unifies the cell sets by ensuring that both arguments have the cell set $\{\, c_1, c_2, c_3, c_4, c_5 \,\}$.
7. the assignment into `regs.b.al` is translated into the as-

[3] This is a slight over-simplification. In practice, when a pointer cell admits only values in $\{\, \textbf{NULL}, \textbf{invalid} \,\}$, its offset dimension is omitted.

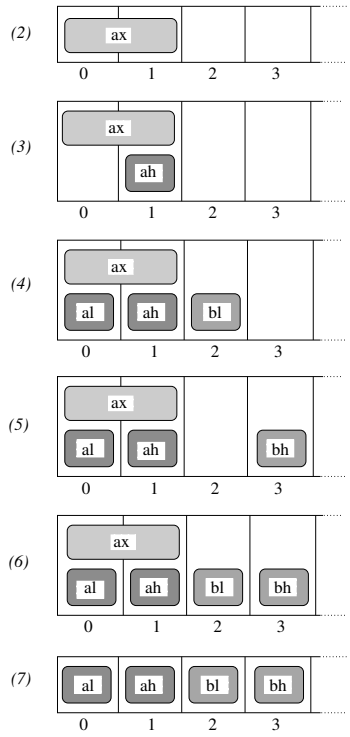


Figure 5.12: Cell sets during the analysis of Fig. 5.5.

signment $\mathbb{S}^\sharp[\![c_3 \leftarrow \texttt{0xab}]\!]$; the cell c_1, which overlaps c_3, is removed as it contents is no longer valid after updating c_3.

In this simple example, all the cells take constant values, so that an analysis with the interval domain gives a precise result. The example from Fig. 5.6, containing pointer casts, would similarly be precisely analyzed.
End of example.

Implementation and experimentation

From the point of view of the analyzer's programmer, the cell-based abstract semantics is a functor that lifts any numeric abstract domain to an abstract domain reasoning on arbitrary C types. While the underlying numeric domain assumes (wrongly) that cells denote independent quantities, the functor corrects this assumption dynamically by maintaining the correspondence between cells (i.e., their overlapping) and issuing cell creation and destruction orders when necessary. A practical benefit is that it makes it easy to convert an abstract interpreter supporting only a well-structured semantics into one supporting a low-level semantics, while reusing all existing numeric abstract domains. This technique was used to adapt the Astrée C analyzer (Sec. 6.2), as reported in [Min06a]. In addition to enabling Astrée to analyze a larger class of software, our experiments showed that switching to a cell-based semantics did not degrade the analysis of software that were analyzed previously with the well-structured semantics: it does not change the precision and only slightly increases the analysis time and memory usage, due to the need to maintain cell maps in addition to abstract invariants. We refer the reader to [Min06a, BCC+10a] for a detailed description of the implementation and experimental results.

5.2.5 Discussion

Precision. Our concrete cell semantics is not complete with respect to the byte-based one, and this can cause some imprecision in the static analysis, whatever numeric abstract domain is chosen. A main source of incompleteness is the cell synthesis function ϕ, which is not exhaustive. Note, however, that ϕ is a parameter of the analysis and it can be refined at will, at the cost of efficiency. Hence, it fits well the design by refinement of static analyzers such as Astrée (Sec. 6.2). For instance, Sec. 5.3 will present a refinement of ϕ to expose the binary encoding of floats. Another source of incompleteness is the choice to remove invalidated overlapping cells after an assignment. More precision could be achieved by keeping and updating overlapping cells. Maintaining more cells results in numeric abstract elements with more dimensions, and comes at a greater cost.

Compared to the byte-level semantics, there is no systematic conversion between scalars and bytes at each memory access. Instead, conversions only occur when trying to read a cell that does not exist, either because no value of the cell's type has been written at this location, or because it was removed by a latter assignment into an overlapping location. Most of the time, $\phi(c)(C)$ simply returns c as the cell already exists; this is always the case if the program refrains from exploiting type punning. Hence, our low-level static analysis is a strict extension of the well-structured analysis and gives the same results for programs obeying strictly the C standard; and it can additionally analyze non-conforming programs.

A final, structural cause of imprecision in the analysis is that pointer bases are abstracted in a non-relational way. Nevertheless, when the underlying numeric domain is relational, the analysis can infer relations between pointer offsets. It can also infer relations between pointer offsets and numeric cells, which shows the benefit of performing a single, combined pointer–numeric analysis instead of trying to resolve all pointer values before performing a purely numeric analysis.

Example 5.2.3. If p points to $\{\, \langle V, o\rangle \mid V \in \{X, Y\},\ o \in [0, 10] \,\}$, after the assignment $q \leftarrow p$ using the polyhedra domain, we can deduce that p and q have the same offset, but not that they point to the same variable.
End of example.

Moreover, using a relational numeric domain also allows retaining and exploiting the relations imposed by ϕ on the different views of the same portion of memory.

Example 5.2.4. Consider a variable A covered with cells of the form $c_i \stackrel{\text{def}}{=} \langle A, i, \textbf{unsigned char}\rangle$ for $0 \leq i < n$. Then $*_{\textbf{unsigned short}}\, \&A$ creates a cell c initialized with $c_0 + 256 \times c_1$. If the numeric domain can represent the relation $c = c_0 + 256 \times c_1$, then the test $*_{\textbf{unsigned short}}\, \&A \leq 1000$ will not only refine the value of c, but also the value of c_1.
End of example.

Offset domains. Due to the use of numeric abstract domains to abstract byte-level offsets, we may need to represent new kinds of numeric properties. In particular, pointers are frequently aligned, which means that the offsets are multiple of *alignof*(t) for some type t (on some processors, dereferencing non-aligned pointers generates a run-time error). These can be represented using the non-relational congruence domain introduced by Granger [Gra89]. In more complex cases, such as traversals of multi-dimensional arrays in nested loops, it might

be necessary to infer relational congruence properties, for instance by using the linear congruence equality domain, also proposed by Granger [Gra91].

Related work. There is a relative lack of support in existing literature for analyses that handle type-punning and creative uses of union types and pointer operations. These uses are generally frowned upon, and more research has been devoted to remove them than to analyze them. This includes the design of static analyzers employing a well-structured model (such as the analysis by Whaley and Lam [WL02]) or the construction of safer dialects of C forbidding them (such as CCured [NMW02]). These methods would reject constructs that are found legitimate by end-users and force them to rewrite their software. Our approach, on the contrary, is to understand these constructs and provide a precise concrete semantics defining their correct and incorrect use, before constructing a static analyzer.

Nevertheless, some existing analyses do support low-level memory operations. This is the case in particular of all field-insensitive analyses. Yong et al. [YHR99] and Venet [Ven04] propose mixed approaches, where only the part of the memory accessed in accordance to the well-structured semantics is abstracted in a field-sensitive way (where the partitioning of the memory can be performed either prior to or during the value analysis). Balakrishnan et al. [BR04] and Wilson et al. [WL95] choose, instead, to use a field-sensitive analysis that returns an imprecise value (e.g., the whole range of the type) for reads that do not match the declared C type. Our analysis is more precise in that it allows the whole memory to be analyzed in a field-sensitive way and tries to synthesise a precise value for accesses that do not obey the well-structured semantics.

There is a large body of work [Hin01] on pointer analysis. Many analyses are intended to be used in optimizing compilers, for instance to check pointer aliasing. They naturally have a large emphasis on scalability over precision. This is the case, for instance, of the popular unification-based analyses pioneered by Steensgaard [Ste96]. Our context, value analysis, is quite different: on the one hand we wish for very precise pointer information in order to limit the amount of weak updates (that degrade the precision of the analysis); on the other hand, our numeric analysis already uses a field-sensitive, flow-sensitive (and, in the case of Astrée, context-sensitive) engine. It is thus natural to include pointers as regular values inferred by a combined pointer and numeric analyzer. In future work, we wish however to evaluate the benefit of performing a fast pointer pre-analysis using one of the existing techniques, with the hope of simplifying the subsequent combined pointer and numeric analysis.

5.3 Bit-aware float abstractions

In Sec. 2.4.4, we showed how to abstract floating-point computations using standard numeric abstract domains originally designed for real arithmetic: we represented floats as reals, and modeled float computations as real computations and rounding as a non-deterministic choice in an interval. This model is already an abstraction of actual float computations: it loses some information, but it is sufficient to analyze most programs; it matches the programmer's expectation that floats compute as reals up to some rounding error. In this section, we discuss a refined concrete model, first introduced in [Min12a], designed to analyze some programming idioms where this abstraction is insufficient. This new model includes the *special floats*: infinities and Not-a-Numbers (*NaN*), which were not represented in the semantics before. Moreover, it takes into account the bit-level encoding of floats, which can be exposed by type-punning through the cell-based memory semantics of Sec. 5.2.3. We then propose a parametric abstract domain based on a well-chosen set of predicates. Similarly to our work on machine integers (Sec. 5.1) this abstraction is quite simple and tied to specific programming patterns; it acts as a complement, not as a replacement, for more generic domains (such as intervals and polyhedra).

```
double validate(double d) {
    unsigned* p = (unsigned*)&d;
    if (((*p & 0x7ff00000) >> 20) == 2047)
        d = 0.;
    return d;
}
```

Figure 5.13: Floating-point validation.

```
union u { int i[2]; double d; };
double cast(int i) {
    union u x,y;
    x.i[0] = 0x43300000;
    y.i[0] = x.i[0];
    x.i[1] = 0x80000000;
    y.i[1] = i ^ x.i[1];
    return y.d - x.d;
}
```

Figure 5.14: Integer to floating-point conversion.

5.3.1 Examples

Our main motivation comes from the example programs in Figs. 5.13, 5.14, and 5.15.

Example 5.3.1. Figure 5.13 presents a validation function that examines the bit-pattern of the double-precision float `d` in order to filter out all special numbers (which are replaced with 0). It always returns a non-special float.
End of example.

Example 5.3.2. Figure 5.14 presents a function that converts a 32-bit signed integer `i` to a 64-bit float, using only integer arithmetic and a float subtraction. It first constructs the float representation for $\texttt{x.d} = 2^{52} + 2^{31}$ and $\texttt{y.d} = 2^{52} + 2^{31} + \texttt{i}$ using integers, and then computes $\texttt{y.d} - \texttt{x.d} = \texttt{i}$ in float. As all 32-bit integers can be represented in a double precision float, this conversion is exact (there is no rounding error). This program example is a C version of the assembly code generated by compilers for PowerPC processors (this is necessary because these processors lack a native instruction to perform the cast). It is common practice for critical software to use a hand-written C function instead of relying on compiler-generated code.
End of example.

Example 5.3.3. Figure 5.15 presents (a simplified version of) a function to compute the square root of a 64-bit float. It first decomposes the argument `d` into a mantissa in $[1, 4]$ and an even exponent. Then, it computes the square root of the mantissa through a polynomial (for the sake of concision, we omit the

```
double sqrt(double d) {
    double r;
    unsigned* p = (unsigned*)&d;
    int e = (*p & 0x7fe00000) >> 20;
    *p = (*p & 0x801fffff) | 0x3fe00000;
    r = ((c1*d+c2)*d+c3)*d+c4;
    *p = (e/2 + 511) << 20;
    p[1] = 0;
    return d * r;
}
```

Figure 5.15: Square root computation.

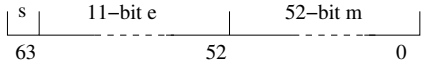
value of the coefficients `c1`,...,`c4`) and divides the exponent by two (with truncation).
End of example.

5.3.2 Concrete semantics

For the sake of presentation, we focus solely on 64-bit double-precision numbers as defined by the widespread IEEE 754 standard [IEE85], and assume a big-endian architecture. A 64-bit float $\langle s, e, m\rangle$ is composed, from the most significant bit to the least significant bit, of:

- a 1-bit sign s;
- a 11-bit *exponent* $e = e_{10} \cdots e_0$;
- a 52-bit *mantissa* $m = m_0 \cdots m_{51}$;

which can be described graphically as:



Float values $\overline{\mathbb{F}}$ now include, in addition to a finite subset of reals $\mathbb{F}$, three special numbers: *NaN* (Not-a-Number), $+\infty$, and $-\infty$. Hence, we state:

$$\overline{\mathbb{F}} \subseteq \mathbb{V} \stackrel{\text{def}}{=} \mathbb{R} \cup \{+\infty, -\infty, NaN\} \ . \tag{5.28}$$

The mapping between the bit-encoding of a float and its value is described by the *dbl* function in Fig. 5.16. Note that *dbl* is not one-to-one as several representations for *NaN* exist. Moreover, the IEEE 754 standard distinguishes between positive zero and negative zero, while $\overline{\mathbb{F}}$ has a single, unsigned zero. These simplifications are justified by the lack of realistic programs where these differences matter (for instance, $+0$ and -0 compare equal with the C operator `==`).

5.3.3 Abstract semantics

Abstracting float values. We first consider the problem of abstracting, using a numeric abstract domain, environments $X \in \mathcal{P}(\mathcal{V} \to \mathbb{V})$ that may include special float values. A straightforward solution is to decompose X into environments R containing only reals (e.g., replacing special values with zero) and a map M from variables to the set of special values they can hold, i.e.:

$$\begin{aligned} R &\stackrel{\text{def}}{=} \{\, \rho \in \mathcal{V} \to \mathbb{R} \mid \exists \rho' \in X : \forall V \in \mathcal{V} : \\ &\qquad \rho(V) = \rho(V') \in \mathbb{R} \vee (\rho(V) = 0 \wedge \rho'(V) \notin \mathbb{R}) \,\} \\ M &\stackrel{\text{def}}{=} \lambda V. \{\, v \in \{+\infty, -\infty, NaN\} \mid \exists \rho \in X : \rho(V) = v \,\} \ . \end{aligned} \tag{5.29}$$

Then R can be abstracted using any numeric domain (such as intervals and polyhedra), while M is represented in extension. Note that this is an abstraction: special values are maintained in a non-relational way, which we justify below.

Special values appear for ill-defined operations (e.g., $1/0 = +\infty$, $0/0 = NaN$) and obey simple algebraic rules (e.g., $-2 \times +\infty = -\infty$, $+\infty + -\infty = NaN$). Hence, it is easy to enrich abstract domain operations to maintain M soundly. Few programs exploit the algebra of special values; generally, floats are often meant as an approximation of reals and the occurrence of a special value is a non-recoverable error. We can model this in the semantics of expressions by returning a run-time error $\omega \in \Omega$ instead of an environment at the location of the offending operator. It is nevertheless useful to represent special floats in our abstract domain as, although they can no longer be created as a result of an operation, they can still appear as program inputs. We wish to analyze programs that input arbitrary (possibly special) values and validate them before use, as in Fig. 5.13. Thus, our analyzer must handle specials, if only to prove that they are successfully removed. In this context, where special values are not propagated, a non-relational information on specials is sufficient.

Bit-level expressions. The bit-level structure of floats cannot be exposed using only float operations. The examples in Figs. 5.13 to 5.15 resort to type punning (using pointer casts and union types). We naturally exploit the low-level memory model of Sec. 5.2.3 to detect such manipulations, but then rely on specific numeric domains to model their effect. To enable some communication between memory and numeric semantics, we enrich the language of numeric expressions with operators that convert values based on their bit-representation:

$$\begin{aligned} \mathit{expr} \quad ::= \quad & \mathit{dbl\text{-}of\text{-}word}(\mathit{expr}, \mathit{expr}) \\ \mid \quad & \mathit{hi\text{-}word\text{-}of\text{-}dbl}(\mathit{expr}) \end{aligned} \tag{5.30}$$

where *dbl-of-word* converts two 32-bit integers into a 64-bit float, and *hi-word-of-dbl* extracts the hi-order 32 bits of a 64-bit float as an unsigned integer. Their semantics is defined formally in Fig. 5.18. The operators are not intended to be used directly in programs; they are introduced by the cell synthesis function to express a float cell as a function of existing integer cells, and the other way round. This is achieved by enriching the ϕ function from Fig. 5.11 as shown in Fig. 5.17.

Example 5.3.4. In the program of Fig. 5.13, the expression `((*p & 0x7ff00000) >> 20) == 2047` first triggers the creation of a cell $c = \langle$`d`, 0, **unsigned int**$\rangle$ which is initialized in the numeric domain by $\mathbb{S}^\sharp[\![\, c \leftarrow \mathit{hi\text{-}word\text{-}of\text{-}dbl}(\langle$`d`, 0, **double**$\rangle) \,]\!]$. Then, the test is evaluated as: $\mathbb{S}^\sharp[\![\, ((c \,\&\, \texttt{0x7ff00000}) \gg 20) == 2047 \,]\!]$.
End of example.

Predicate abstract domain. It is difficult to envision a general domain able to reason about arbitrary binary float manipulations. On the other hand, the programs in Figs. 5.13 to 5.15 are rather idiomatic. Hence, we suggest using a domain based on pattern matching of expressions to detect selected predefined uses. It is not sufficient to match each expression independently as computations are generally spread across sequences of statements. We need, in addition, to maintain some state that retains and propagates information between statements. We maintain this state in a *predicate domain* $\mathcal{D}^\sharp_{\mathcal{P}_{red}}$, which maps each cell in C to a syntactic predicate in a language $\mathcal{P}red$. The exact language of predicates depends on the idioms to be analyzed. For instance, to analyze Fig. 5.14, we

$$\underline{dbl \in \{0,1\}^{64} \to \mathbb{V}}$$

$$dbl(s, e_{10}, \dots e_0, m_0 \dots, m_{51}) \stackrel{\text{def}}{=} \begin{cases} (-1)^s \times (1 + \sum_{i=0}^{51} 2^{-i-1} m_i) \times 2^{(\sum_{i=0}^{10} 2^i e_i - 1023)} & \text{if } \sum_{i=0}^{10} 2^i e_i \notin \{0, 2047\} \\ (-1)^s \times (\sum_{i=0}^{51} 2^{-i-1} m_i) \times 2^{-1022} & \text{if } \forall i : e_i = 0 \\ (-1)^s \times \infty & \text{if } \forall i : e_i = 1 \wedge \forall j : m_j = 0 \\ NaN & \text{if } \forall i : e_i = 1 \wedge \exists j : m_j = 1 \end{cases}$$

Figure 5.16: Bit-encoding of 64-bit floats.

$$\phi\langle V, o, t\rangle(C) \stackrel{\text{def}}{=} \begin{cases} \textit{hi-word-of-dbl}(c) & \text{if } c = \langle V, o, t'\rangle \in C \wedge t \in \textit{int-type} \wedge t' = \textbf{double} \wedge \textit{sizeof}(t) = 4 \\ \textit{dbl-of-word}(c_1, c_2) & \text{if } c_1 = \langle V, o, t'\rangle \in C \wedge c_2 = \langle V, o+4, t'\rangle \in C \wedge \\ & \quad t = \textbf{double} \wedge t' \in \textit{int-type} \wedge \textit{sizeof}(t') = 4 \end{cases}$$

Figure 5.17: Cell synthesize function for floats.

need to express symbolically the reinterpretation of integers as floats and the flipping of the high-order bit of an integer, so, we choose:

$$\begin{array}{lcll} \mathcal{D}^\sharp_{\mathcal{P}red} & \stackrel{\text{def}}{=} & C \to \mathcal{P}red & \\ \mathcal{P}red & ::= & \top & \\ & | & c \text{ ^ } \texttt{0x80000000} & (c \in C) \\ & | & \textit{dbl-of-word}(\texttt{0x43300000}, c) & (c \in C) \end{array} \tag{5.31}$$

where $\top$ denotes the absence of information. The ordering is a flat one based on syntactic predicate equality:

$$X^\sharp \sqsubseteq^\sharp_{\mathcal{P}red} Y^\sharp \iff^{\text{def}} \forall c \in C : X^\sharp(c) = Y^\sharp(c) \vee Y^\sharp(c) = \top \ . \tag{5.32}$$

An abstract element $X^\sharp \in \mathcal{D}^\sharp_{\mathcal{P}red}$ denotes the set of environments that satisfy all the predicates in $X^\sharp$, where predicates are evaluated as expressions using $\mathbb{E}[\![\ ]\!]$:

$$\gamma_{\mathcal{P}red}(X^\sharp) \stackrel{\text{def}}{=} \{\, \rho \in C \to \mathbb{V} \mid \forall c \in C : X^\sharp(c) = \top \vee \rho(c) \in \textit{fst}(\mathbb{E}[\![\, X^\sharp(c) \,]\!]\rho) \,\} \ . \tag{5.33}$$

We present the abstract operators in Fig. 5.19. They actually operate on a pair of a predicate and an interval map, i.e., in a partially reduced product of $\mathcal{D}^\sharp_{\mathcal{P}red}$ and the interval domain $\mathcal{D}^\sharp_i$ (Sec. 2.4.1). This is necessary because, on the one hand, pattern matching of constants requires evaluating expressions (this way, we are able to match complex constant expressions, not reduced to syntactic constants) and, on the other hand, the identities discovered using predicates can lead to identities expressed with intervals (for instance, when we discover that a code is equivalent to an exact conversion from integers to floats, we can safely state that the float bounds are equal to the integer ones). Assignments $c \leftarrow e$ and tests $e \bowtie 0$ are handled in several steps. Firstly, the predicate abstract information is used by the *combine* function to perform some symbolic computation on the argument expression e. Secondly, this new expression is used in the interval assignment or test. Additionally, the assignment $c \leftarrow e$ removes the binding for c in the predicate map, as well as all the bindings where c occurs. If the assigned expression matches that of a predicate in $\mathcal{P}red$, a new binding is created. The abstract join filters predicates to keep only bindings that are equivalent in both arguments. There is no need for a widening in $\mathcal{D}^\sharp_{\mathcal{P}red}$ as the domain is flat.

As for the choice of predicates in (5.31), the abstract operators we have chosen were especially tailored for the analysis of the conversion example from Fig. 5.14.

Example 5.3.5. Consider the analysis of Fig. 5.14 and, more precisely, the evaluation of `y.d - x.d`. Just before this expression, the interval domain states that `x.i[0]` and `y.i[0]` equal `0x43300000`, while `x.i[1]` equals `0x80000000`. Moreover, the predicate domain states (symbolically) that `y.i[1]` equals `i ^ x.i[1]`. The first step of the evaluation of `y.d - x.d` is performed by the memory domain: it creates two cells c_1 (for `x.d`) and c_2 (for `y.d`) initialized respectively as:

$$\mathbb{S}^\sharp_{\mathcal{P}red}[\![\, c_1 \leftarrow \textit{dbl-of-word}(\texttt{x.i[0]}, \texttt{x.i[1]}) \,]\!]$$
$$\mathbb{S}^\sharp_{\mathcal{P}red}[\![\, c_2 \leftarrow \textit{dbl-of-word}(\texttt{y.i[0]}, \texttt{y.i[1]}) \,]\!] \ .$$

These synthetic assignments induce the predicates:

$$c_1 = \textit{dbl-of-word}(\texttt{0x43300000}, \texttt{x.i[1]})$$
$$c_2 = \textit{dbl-of-word}(\texttt{0x43300000}, \texttt{y.i[1]}) \ .$$

Finally, the memory domain passes the expression $c_2 - c_1$ to the predicate domain, which transforms it into `(double)x.i[1]` by the pattern matching in *combine*, and passes it to the interval domain. Hence, the interval domain only sees a regular conversion from integers to floats.
End of example.

Due to the lack of space, we describe only very concisely how our two other examples, in Figs. 5.13 and 5.15, can be handled. To handle them, it is necessary to extend the predicate language and the abstract functions. As our examples extract the high-order word of their float argument, we enrich $\mathcal{P}red$ with the predicate $\textit{hi-word-of-dbl}(c)$, where the parameter c denotes a float cell. Secondly, as these examples use bit-wise operations to apply bit-masks, we perform a reduced product between our predicate domain and the bit-field domain from Sec. 5.1.4, in addition to the interval domain. The validation example of Fig. 5.13, additionally uses a reduction with the maps P storing the set of special values contained in a cell (5.29). In all cases, the domain is straightforward and all the subtlety lies in the reduction: it must transfer information between the various numeric abstractions, based on equivalences of the form $c_1 = \textit{hi-word-of-dbl}(c_2)$ discovered by the predicate domain.

$\mathbb{E}[\![\ \mathit{hi\text{-}word\text{-}of\text{-}dbl}(e_1)\ ]\!]\rho \stackrel{\text{def}}{=} \langle \{\ \sum_{i=0}^{31} 2^i b_{i+32} \mid \exists b_0, \ldots, b_{31} : \mathit{dbl}(b_{63}, \ldots, b_0) \in V_1^\rho\ \},\ O_1^\rho \rangle$
$\mathbb{E}[\![\ \mathit{dbl\text{-}of\text{-}word}(e_1, e_2)\ ]\!]\rho \stackrel{\text{def}}{=} \langle \{\ \mathit{dbl}(b_{31}^1, \ldots, b_0^1, b_{31}^2, \ldots, b_0^2) \mid \forall j \in \{1,2\} : \sum_{i=0}^{31} 2^i b_i^j \in \mathit{wrap}(V_j^\rho, [0, 2^{32}-1])\ \},\ O_1^\rho \cup O_2^\rho \rangle$
where $\langle V_1^\rho,\ O_1^\rho \rangle \stackrel{\text{def}}{=} \mathbb{E}[\![\ e_1\ ]\!]\ \rho$ and $\langle V_2^\rho,\ O_2^\rho \rangle \stackrel{\text{def}}{=} \mathbb{E}[\![\ e_2\ ]\!]\ \rho$

Figure 5.18: Concrete semantics of bit-level conversions between floats and integers.

$\mathbb{S}^\sharp_{\mathcal{P}red}[\![\ V \leftarrow e\ ]\!]\langle X_p^\sharp,\ X_i^\sharp \rangle \stackrel{\text{def}}{=}$
  let $Y_i^\sharp = \mathbb{S}_i^\sharp[\![\ V \leftarrow \mathit{combine}(e, X_p^\sharp, X_i^\sharp)\ ]\!]\ X_i^\sharp$ in
  let $Y_p^\sharp = \lambda W.$ if $W = V$ or $V \in \mathit{var}(X_p^\sharp(W))$ then $\top$ else $X_p^\sharp(W)$ in
  if $\exists e_1, W : e = W$ `^` $e_1 \wedge \mathbb{E}_i^\sharp[\![\ e_1\ ]\!]X_i^\sharp \in \{\{2^{31}\}, \{-2^{31}\}\}$ then
    $\langle Y_p^\sharp[V \mapsto W$ `^ 0x80000000`$],\ Y_i^\sharp \rangle$
  else if $\exists e_1, W : e = \mathit{dbl\text{-}of\text{-}word}(e_1, W) \wedge \mathbb{E}_i^\sharp[\![\ e_1\ ]\!]X_i^\sharp = \{1127219200\}$ then
    $\langle Y_p^\sharp[V \mapsto \mathit{dbl\text{-}of\text{-}word}($`0x43300000`$, W)],\ Y_i^\sharp \rangle$
  otherwise
    $\langle Y_p^\sharp,\ Y_i^\sharp \rangle$

$\mathbb{S}^\sharp_{\mathcal{P}red}[\![\ e \bowtie 0\ ]\!]\langle X_p^\sharp,\ X_i^\sharp \rangle \stackrel{\text{def}}{=} \langle X_p^\sharp,\ \mathbb{S}_i^\sharp[\![\ \mathit{combine}(e, X_p^\sharp, X_i^\sharp) \bowtie 0\ ]\!]X_i^\sharp \rangle$

$X_p^\sharp \cup^\sharp_{\mathcal{P}red} Y_p^\sharp \stackrel{\text{def}}{=} \lambda V.$ if $X_p^\sharp(V) = Y_p^\sharp(V)$ then $X_p^\sharp(V)$ else $\top$

where $\mathit{combine}(e, X_p^\sharp, X_i^\sharp)$ replaces sub-expressions of the form V_1 `-` V_2 in e with `(double)`I when:
$\exists V_1', V_2' : \forall j \in \{1,2\} : X_p^\sharp(V_j) = \mathit{dbl\text{-}of\text{-}word}($`0x43300000`$, V_j') \wedge$
$X_p^\sharp(V_1') = I$ `^ 0x80000000` $\wedge\ X_i^\sharp(V_2') = [-2^{31}, -2^{31}]$

and $\mathit{var}(p)$ denotes the set of cells appearing in the predicate p.

Figure 5.19: Abstract operator examples in the partially reduced product $\mathcal{D}^\sharp_{\mathcal{P}red} \times \mathcal{D}_i^\sharp$ of the predicate and interval domains.

Experiments. We have implemented the predicate domain in the Astrée analyzer (described in Sec. 6.2) in order analyze programs featuring some idiomatic manipulations of floats at the bit level, similar to the examples in Figs. 5.13 to 5.15. Our experience shows that the predicate domain is scalable and its added cost is negligible with respect to the domains already included in Astrée. This comes as no surprise as the predicate domain maintains a single, small information per variable and the abstract operators only perform simple reductions with non-relational domains. By design, it is precise enough to analyze the idioms embedded in its predicate language. We refer the reader to [Min12a] for the detailed experimental results.

The domain is parametrised, but adapting it to new idioms requires developing new abstract functions, which is a costly, non-automated process. So, a natural question is whether the design by refinement scales up. In our experience, refining the predicate domain was never a bottleneck: a fixed set of a dozen predicates is sufficient to analyze our code base of several millions of lines. We attribute this to the facts that the predicates are sufficiently generic and the reduction with the numeric domains provide sufficient semantic information to balance the syntactic nature of the predicates.

5.3.4 Future work

A natural continuation of this work is to keep enriching the domain when needed by novel idioms encountered in newly analyzed programs, with the hope of building a library of predicates covering most analysis needs. Future work include alleviating the burden on the analysis designer when the domain needs to be refined. We could envision more general predicates and more powerful propagation methods (possibly at the expense of scalability). An attractive solution consists in transferring some burden to the end-user by allowing him to add predicates and rules using a dedicated language (a source of inspiration for this is the TVLA system [LAMS04], which defines such a language for parametric shape analysis).

5.4 Conclusion

In this chapter, we have proposed an heterogeneous set of domains, with the common purpose of switching from the static analysis of an idealized language (whether it is a pure numeric language, as introduced for the sake of formal exposition in Sec. 2.3.1, or the abstract, fully-portable C language described by the C specification) to a static analysis of a real language (the C language as it is used most often in practice). This chapter shows, in particular, the importance of providing an adequate concrete semantics, capturing precisely the complex effects of actual languages (such as two's complement arithmetic or type punning) in a clear, mathematical way. Once the concrete semantics is defined, the solutions to abstract it often follow without effort.

These domains found an application in our static analyzers for C programs, Astrée and AstréeA. In fact, applications often came first, bringing the need to refine the analysis by the construction of new, adapted domains. Language implementations are pragmatic, as they must take into account the limitations of hardware. Our domains are pragmatic as well. The classic domains discussed in Chap. 2 and the domains we introduced in Chap. 4 were constructed on an ideal semantics, and the domains we introduced here sometimes lack their formal perfection. Faced with a lack of Galois connections and optimal

operators, we rely on practical use-cases and experiments to guide the design of our abstractions.

Chapter 6

Applications

One of the expected results of our work is the development of new static analysis methods that have practical usefulness and positively impact the quality of software. We thus spend significant efforts to implement the proposed techniques and validate them by experiments. While the soundness of the methods is guaranteed by our careful application of the abstract interpretation methodology, its actual usefulness is not. Firstly, it is important to assess the efficiency of the methods and their capacity to scale up to realistic programs (that now count in millions of lines of code). Secondly, it is necessary to assess their precision as they are, by necessity, incomplete. The use of non-exact and non-monotonic abstract operators (in particular widenings) makes the theoretical prediction of the precision on actual analyses quite difficult. Thirdly, experiments become an integral part of the analysis development when considering specialized analyzers aiming at proving a specific class of properties on a specified class of programs with few or no false alarm. In this context, the classes of programs and properties are set first, and the abstractions developed later, so that only experimentation provides a measure of the success of those abstractions. Moreover, experimental failures provide a positive feedback as they uncover the need for more complex abstractions, trigger a refinement of the specialized analyzer, and drive further the research on static analysis by abstract interpretation, until the precision goal is reached.

For these reasons, we have implemented and tested all the static analysis techniques introduced in this report. This is a time-consuming task, that can only be envisioned as a team effort. We participated in two kinds of implementations. Firstly, we developed research tools, intended for academic use. They provide reusable components that can be easily perused by academic peers, integrated into a variety of academic tools, thus encouraging further researches. The core example is the Apron library of numeric abstractions, described in Sec. 6.1. Secondly, we developed analysis tools intended for industrial use. They can be directly applied to real programs with minimal knowledge of abstract interpretation. A first example is the Astrée analyzer, dedicated to proving the absence of run-time error in embedded synchronous C programs (Sec. 6.2). Astrée is now industrialized by AbsInt [Abs]. A second example is the AstréeA analyzer, that extends Astrée to analyze concurrent programs (Sec. 6.3). AstréeA is still a prototype in heavy development, but nevertheless targets realistic programs and is intended to be industrialized.

6.1 Apron: numeric abstract domain library

Apron is a library of numeric abstract domains. It was developed mainly by Bertrand Jeannet and myself. Its development started in 2006, following the theoretical work during a French project, also named Apron, coordinated by François Irigoin. The Apron library is described in the publication [JM09] and available on-line [JM06].

Motivation. Apron has three main goals.

Firstly, it provides a robust and versatile implementation of classic abstract domains under a common API. We wish to facilitate the work of analysis implementers by providing ready-to-use building blocks. The common API makes it possible to switch between and experiment with the various domains with minimal effort. Moreover, bindings are available for a variety of popular languages: C, C++, OCaml, and Java.

Secondly, it encourages the research in numeric abstract domains. Apron provides a platform allowing the integration of new domains with minimal efforts. Domain implementers benefit from a ready-to-use toolbox facilitating domain development: scalar and interval arithmetic in many types (machine integers and floats, but also arbitrary precision integers and rationals through the GMP library [GNUa], and extended precision floats through the MPFR library [GNUb]), operations on affine expressions, interval affine expressions, as well as arbitrary expression trees. Domain implementers are required to provide only a core set of operators, as many operations benefit from generic fallback implementations (these can be overloaded with custom ones if the domain implementer sees a benefit in it, such as improved efficiency).

Thirdly, it provides a teaching and demonstration tool to disseminate knowledge on abstract interpretation. Apron is freely available on-line under the LGPL license [JM06]. It is accompanied with a sample static analyzer, Interproc [LAJ11], developed by Gaël Lalire, Mathias Argoud, and Bertrand Jeannet. It performs a forward and backward analysis to infer invariants on a toy inter-procedural numeric language. Interproc is intuitive and can be used on-line from a Web browser without installation.

Principles. A distinguishing feature of Apron is that its API is not tied to a specific abstract domain, but rather to a concrete semantics. This in unlike many other libraries, which only provide exact or best abstractions for operations matching the expressive power of one domain, ignoring other operations, and give access to the internal representation of abstract elements (examples include the Parma Polyehdra Library [BHZ08], as well as the NewPolka and octagon library precursors to Apron). By contrast, a domain in Apron is only required to implement a sound abstraction of the concrete semantics, using best effort to ensure a good precision despite the limited expressiveness of the domain and its implementation details (for instance, in case

float arithmetic is used internally). This enables Apron to expose a rich set of semantic operations, which are supported by all domains. These include: assignments, tests, and substitutions of affine and non-affine expressions, including the support for integer and floating-point expressions and non-deterministic expressions, joins, meets, widenings, narrowings, projections, as well as a host of less standard operations such as: n-ary joins, parallel assignments, dimension folds and expands [GDD+04], etc. Hence, many kinds of program semantics can be directly mapped to Apron operators.

Included abstractions. In addition to its well-defined API, Apron is useful for the abstractions it already contains. Apron includes an implementation of the polyhedra domain based on the double description method and exact rational arithmetic [CH78] and its restriction to affine equalities [Kar76]; it also includes an implementation of the octagon domain [Min06b] and the interval domain [CC76] with bounds of arbitrary type. More recently, the Zonotope domain [GGP09] was added to Apron. On relational domains, non-linear and floating-point expressions are handled using linearization [Min04a]. These domains also support integer tightening to model more precisely integer-valued variables. The library currently consists of 130,000 lines of C, C++, OCaml, and Java.

Applications. The Apron library has been used in several research projects, including the construction and experimentation of real-life analyzers [OHL+12]), the inference of non-numeric invariants [BDES12], the inference of properties beyond invariants such as termination [BCC+07], and novel uses of abstract interpretation such as solving constraint programming problems [PMTB13]. We also used Apron in our research work described here: all the abstract domains developed in Chap. 4 were implemented as proofs-of-concepts by Liqian Chen in the Apron library and tested using Interproc.

Apron is now a stable platform for research. In the future, we wish to enrich Apron with new domains, in particular: robust versions of the domains developed in Chap. 4, and the most popular domains proposed in the recent literature (such as template polyhedra [SSM05]).

6.2 Astrée: proving the absence of run-time error in synchronous embedded C software

Astrée is a static analyzer by abstract interpretation checking for the absence of run-time error in embedded C programs. It has been developed since 2001 at ENS by the Abstraction team: Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, David Monniaux, Xavier Rival, and myself. Astrée is industrialized and made commercially available since 2009 by AbsInt [Abs]. Astrée stands for *Analyseur statique de logiciels temps-réel embarqués*, i.e., real-time embedded software static analyzer.

This section only briefly describes Astrée. We refer the reader to [BCC+10a] for more information, as it is the most comprehensive and up-to-date publication on Astrée. Several additional articles describe the scientific aspects of Astrée [BCC+02, BCC+03, CCF+06, CCF+07, CCF+09, BCC+10b], while others focus on its industrial use [DS07, SD07, BCC+09, KFW+09, KWN+10, BCC+11]. More information on Astrée is also available on its web-site [BCC+].

I started working on Astrée from its beginning, in 2001. My early contributions are reported in my PhD [Min04b]. I continued working on Astrée after my PhD, in particular by developing the abstractions described in Chap. 5 and by extending it to parallel programs (Sec. 6.3).

6.2.1 Scope and limitations

Language. Astrée accepts programs written in a large subset of C 99 [ISO07]. However, it does not support dynamic memory allocation nor recursivity. Given that Astrée targets embedded software, this is not a strong limitation: these features are generally forbidden in such software. Astrée does not support parallel nor concurrent software (this imitation is addressed by AstréeA, in Sec. 6.3). Astrée performs a monolithic analysis of whole programs, which must not contain undefined symbols. The source code of all libraries or, alternatively, stubs modeling their effect, must be provided. The range of input values (such as memory-mapped registers of input devices) must also be specified. The analysis is sound only with respect to these model stubs and input specifications.

Semantics. The semantics of Astrée is based on the C99 standard [ISO07] (including stubs modeling the standard C library), supplemented by the IEEE 754 standard [IEE85] defining floating-point computations.

Given the low-level nature of most C programs, the C language standard is surprisingly high-level and abstract: many features are not specified concretely in order to allow different interpretations by compiler designers. The actual effect of under-defined features can range from producing a well-defined and documented outcome on a given implementation to producing a completely non-deterministic effect, possibly causing the program to exhibit erratic behaviors at some later point (for instance, when silently corrupting the memory). The large majority of C programs are not portable and make specific hypotheses about some of these behaviors. This is especially the case in embedded programs, which are designed to run on a single platform and often require a low-level access to the system. Thus, it is important when analyzing them to take this refined semantics into account.

As a consequence, the semantics used by Astrée is actually slightly stricter and much lower-level than that of the C standard. It assumes two's complement integer arithmetic, a flat memory model (pointers as integers), and it models precisely the layout of variable fields in memory. It is also more permissive, allowing type-punning as described in Sec. 5.2.3. Finally, it is parametrized by platform-specific choices, such as the bit-size and alignment of types, the byte ordering (endianess), etc. Once the platform parameters are fixed, an analysis is only sound with respect to the program executions on this platform.

Run-time errors. Astrée performs a value analysis in order to check for run-time errors. Such errors include:
- overflows in signed and unsigned arithmetic,
- division and modulo by zero,
- bit shifts exceeding the bit-size of arguments,
- invalid values for enumerated types,
- overflows and invalid operations in float (generating infinities or NaN),

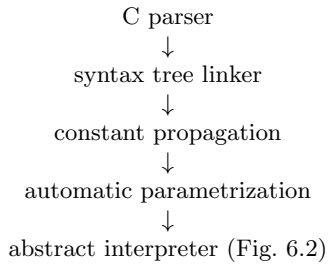


Figure 6.1: Architecture of Astrée.

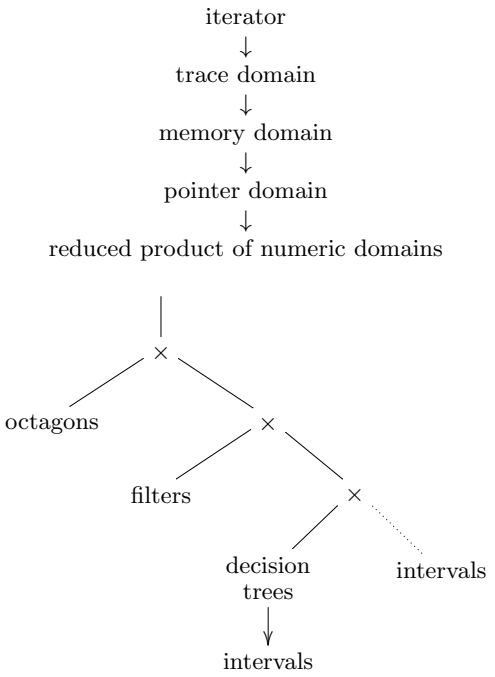


Figure 6.2: Abstract domain hierarchy in Astrée.

– out-of-bound array accesses,
– invalid, NULL, dangling, or unaligned pointer dereferences,
– invalid pointer arithmetic or comparison,
– violated user assertions (arbitrary boolean C expressions).

Most errors correspond to a lack of conformance with respect to the definition of the language (such as: no division by zero) or good programming practices (such as: no wrap-around). However, assertions allow the user to specify its own safety requirements and introduce a slight amount of functional property checking. The analysis is sound in that it reports all run-time errors in the above list. Due to abstractions, it is incomplete and may report false alarms but, if the analysis reports no alarm, then the program is effectively free of errors.

In addition to defining which behaviors cause run-time errors, the semantics must also specify the behavior of the program after a run-time error. Three choices are possible. Firstly, the program can have a well-defined behavior; for instance, an integer overflow silently results in a modular wrap-around. Despite its benign nature, this class of errors is useful to warn the programmer that the actual semantics may differ from the intended or natural one, such as the semantics in perfect integers $\mathbb{Z}$. Secondly, the program can have a range of specified behaviors; for instance, an invalid arithmetic operation may result in any value in the type of the expression to be returned. Thirdly, the program can halt. This last case is used to model actual program termination (such as an uncaught signal caused by a division by zero), but also cases where the outcome is truly unpredictable (such as a memory corruption caused by writing through an invalid pointer) and there is no meaningful way to analyze the program after the error occurs. The semantics after each kind of run-time errors can be parametrized to suit the intended platform and programming rules.

An important point is that Astrée tries as much as possible to continue analyzing program executions, even those that exhibit errors. This requires more complex concrete semantics and abstractions (e.g., taking wrap-around into account, as in Sec. 5.1), but rewards us with a more powerful analyzer, able to precisely analyze low-level programs that exhibit these behaviors on purpose. By contrast, an analyzer stopping at each benign error would leave large parts of such programs unanalyzed.

6.2.2 Architecture

Front-end. As shown in Fig. 6.1, the analysis starts by preprocessing and parsing the C sources. Each C file is converted into an abstract syntax tree; they are then merged by a linker to resolve symbols and incomplete definitions. A simple and fast intra-procedural constant analysis is then performed, in order to remove constant variables and simplify the program before more costly analyses are performed. An automatic parametrization phase then occurs. It consists in examining the program to determine which variables and which parts of the code would benefit from an improved precision, such as inferring relational or disjunctive properties. The automatic parametrization is based on simple syntactic pattern matching algorithms. For instance, we identify loop counters, array search loops, and boolean variables encoding control information. The result serves to parametrize the subsequent abstract analysis.

Iterator. The abstract analysis itself is performed as an abstract interpretation by induction on the program syntax, in the spirit of the big-step semantics of Sec. 2.3.5, but extended to the more complex control structures offered by the C language. When encountering a function call, the interpreter calls itself recursively to analyze it. Hence the analysis is both fully flow-sensitive and context-sensitive (this also explains the lack of support for recursive functions in the language). Jumps, such as gotos, breaks from loops, and early returns from functions, disrupt the normal flow of a structured program; they are integrated into the big-step semantics by using continuations (i.e., we maintain a table of abstract environments for jump instructions the target of which has not been encountered yet). Backward gotos, which are equivalent to loops, are handled by extra function-level iterations with widening.

Abstract domains. Astrée has a modular design: instead of using a single abstraction, it uses a combination of many abstract domains. This is illustrated in Fig. 6.2.

Firstly, Astrée employs trace partitioning [MR05], a technique to improve the precision of abstract analyses by imbuing them with a small degree of path-sensitivity: at a given program point, we distinguish environments coming from different

classes of execution paths, which are abstracted separately, enabling a limited form of disjunctive invariants. This is implemented in Astrée as a generic functor that lifts any (trace-unaware) state abstraction into a trace abstraction. Trace partitioning can be costly, and so, it is only applied to the parts of the programs that have been selected by the automatic parametrization.

The pointer and the memory domains add the support for pointer data-types and structured C data-types (arrays, structures, unions) to plain numeric abstractions. They follow the technique described in details in Sec. 5.2.4.

Finally, numeric environments (composed of only machine integers and floats) are abstracted using a reduced product of numeric domains. We find there the interval domain, which is quite important as it is scalable and infers the bounds required to express the absence of run-time error. Figure 6.2 shows a few other example domains used in Astrée: octagons [Min06b] to infer a limited subset of affine relations, a domain specialized for the analysis of digital filters [Fer04], and a domain to partition numeric invariants with respect to boolean variables based on decision diagrams [Bry86]. In total, Astrée, features over 30 numeric domains, many of which are described in [BCC$^+$10a]. More costly domains, such as octagons and boolean partitioning, are only applied to the variables selected by the automatic parametrization. Astrée performs a partial reduced product of these domains and it is very parsimonious in the amount of information exchanged between them (given the large number of domains, propagating each invariant to every domain would not scale up). We use a modular framework for domain communication, including the ability for domains to request invariants of a specific shape and to broadcast a portion of their invariants; it is described in details in [CCF$^+$06].

6.2.3 Specialization

Principle. The main difference between Astrée and other sound static analyzers for C based on abstract interpretation (such as [Ya, The]) is that it is specialized: Astrée is designed to perform with a high precision (few or no false alarm) and efficiency on a specific class of programs, while other programs are analyzed soundly but possibly imprecisely or less efficiently. Astrée has thus been specialized towards embedded control-command avionic software, and later extended to embedded space control-command software. The specialization corresponds to choosing a specific set of abstract domains and control/precision trade-offs. The specialization is achieved by considering a set of representative programs in the class of interest and iteratively refining the abstractions until no false alarm remains on this set. We describe this process below and refer the reader to [CCF$^+$07] for a more detailed comparison between Astrée and other static analyzers.

Target codes. Control-command embedded software, targeted by Astrée, generally have a simple form, described by the following pseudo-code:

- initialize state variables;
- loop for the duration of the mission (e.g., 10h):
  - read input variables from sensors;
  - update state variables and compute output variables;
  - output variables to actuators;
  - wait for next clock tick (e.g., every 10ms).

Thus, programs are composed of a large synchronous loop, driven by a clock, that computes a flow of outputs based on a flow of inputs. (This is, of course, a simplified view; in practice, the body of the loop is broken into functions; the input, compute, and output steps may be intertwined; some computations may be triggered only at some multiple of clock ticks.) The computation performed at each iteration step is numeric intensive (using mostly floating-point arithmetic). Moreover, boolean state variables are used to store and propagate control information from one iteration to the following ones.

Programs are generally quite large, from 100 Klines to more than 1 Mlines (most of which are effectively executed at each loop iteration), and have a large state (around 10 K global variables, the value of which must be tracked from one loop iteration to the other).

Another feature of these programs is that they are automatically generated from graphical block-languages, such as Scade [Est]. One benefit is that the programs are very regular: they are composed of many instances of a small set of hand-written macro-instructions. The disadvantage is that code generators often group unrelated computation arbitrarily and flatten any high-level structure the original program may have.

Abstraction refinement. We started in 2001 with a simple interval analyzer, which is fast but quite coarse. We then iteratively refined the analyzer based on its result on our target programs. By examining by hand the alarms it raised, we were able to determine that some properties required to prove the absence of error were not inferred by the analyzer. We proceeded to construct the relevant abstract domains and add them to Astrée. In some cases, this consisted in implementing in Astrée an existing domain. For instance, the need for simple relational loop invariants (Ex. 2.4.2) triggered the addition of the octagon domain [Min06b]. We also encountered the case where no domain existed to infer the required invariants, and new domains had to be designed. An example is the digital filter domain developed by Feret [Fer04]. The need for this domain is closely tied to our target application domain, avionic control-command software, where such digital filtering is commonplace, while it may not occur in other kinds of embedded software. Hence, Astrée is specialized to an application domain by the choice of its abstractions. Note that this refinement process is greatly facilitated by the modular design of Astrée, allowing us to easily add a new domain or modify a domain independently from the others.

As we start from an efficient analyzer and only add precision when actually needed, the result remains an efficient analyzer. This is in contrast to methods that reduce the problem at hand to a class of well-studied but very costly problems (such as the inference of arbitrary affine relations with polyhedra, that do not scale up well, while octagons may be sufficient to solve the problem at hand and are more efficient).

We stress on the fact that the refinement process cannot be achieved by automatic means as it is done, for instance, in counter example guided abstract refinement [CGJ$^+$00] (CEGAR). Indeed, refinement in CEGAR consists in automatically selecting a finite subset of potential invariants in an fixed infinite domain (such as $V_i - V_j \leq c$ for a finite set of c) which is not needed in our case as we employ infinite domains with widening [CC92b] (such as octagons, that represent directly the infinite family $V_i - V_j \leq c$). In the context of Astrée, by refinement we mean switching to another infinite family of

properties (e.g., to analyze digital filters), which requires the design of a new domain; it is an intellectual process out of the reach of current automatic refinement methods.

In some circumstances, adding a new abstract domain is not actually necessary and Astrée already posses a domain able to represent the required invariant. It is thus sufficient to enable the domain where needed, which is achieved by refining the automatic parametrization that selects the degree of relationality and path-sensitivity (for instance, we can consider more variables in octagons). Additionally, we may need to refine the communications between the sdomains so that the invariants inferred by one domain can be effectively exploited by the other ones (for instance, propagating bounds from octagons to intervals).

Semantic specialization. In addition to improving the precision of the analysis by refining its abstractions, it was also sometimes necessary to refine the concrete semantics. Indeed, we started in 2001 with a high-level semantics that was very close to the C specification and left undefined many aspects of the language. They were modeled either as errors that stop the program or errors that return a large set of possible values. Later, we encountered programs that made very precise hypotheses on the semantics of some operations that are undefined by the C standard but perfectly well defined and documented on their particular platforms. This triggered the work described in Chap. 5. More precisely, our initial semantics of signed and unsigned integer overflow was non-deterministic, and later refined to wrap-around (Sec. 5.1). Likewise, we first assumed that no infinity nor *NaN* float could be constructed as these cause run-time errors, before modeling them precisely in our semantics in order to analyze programs that manipulate them (Sec. 5.3). The largest change consisted in switching from a well-structured semantics of memory to a low-level one allowing unrestricted pointer arithmetic, casts, union types, and type-punning (Sec. 5.2).

Throughout the changes in semantics, we could reuse all of the abstract domains included in Astrée with minimal change. This can be explained by two reasons. Firstly, in many cases, the new semantics is a refinement of the former one, so that domains that are sound with respect to the former are also sound with respect to the newer (for instance, modeling wrap-around as a non-deterministic choice is still sound). The change simply allows new domains to be included, that would not be sound with respect to the former semantics (such as the modular intervals from Sec. 5.1.3). Secondly, in many cases, the new concrete semantics is expressed as a function or reuses parts of the former one. For instance, our machine integer semantics is expressed using mathematical integers. Likewise, our low-level memory semantics reduces to a semantics on independent cells, which can be abstracted using classic pointer and numeric domains that are not aware of type-punning. Nevertheless, these changes of semantics triggered the need for new abstract domains to maintain an acceptable level of precision; for instance, it was necessary to add a congruence domain [Gra89] to precisely abstract pointer offsets and avoid mis-aligned dereference alarms (which could not occur in the former, structured memory semantics). Experimental results show that changing the semantic model and adding the required abstract domains did not impact the performance much, while greatly widening the range of C programs that can be analyzed by Astrée (precise figures are reported in [Min06a, Min12a]).

Parametrization. Astrée has many user-visible configuration options and analysis directives that allow changing the cost/precision tradeoff. In particular, the parametrization algorithms can be configured (and even overridden) by the user. Unlike the addition of new domains and reductions, which requires an intervention from the analysis developers, tuning these options can be performed by a knowledgeable end-user.

6.2.4 Interface

Astrée outputs its results as a set of alarms at positions where run-time errors could not be ruled-out by the value analysis, with some context information (such as the full call stack, as the analysis is context-sensitive). It is however quite important to present to the user, in addition to the list of locations, the inferred invariants. Firstly, it helps the user determine if an alarm is justified or spurious and, in this case, the origin of the imprecision causing the alarm (which must sometimes be traced to one or several computations occurring much earlier). Secondly, these invariants provide valuable information to validate the analyzed software beyond the simple absence of run-time error (for instance, by checking the range of outputs against those provided by functional specification documents). Thirdly, it improves the confidence of the end-user in the result of the analysis by allowing him to reconstruct the reasoning made by the analyzer.

The main challenge, when analyzing large programs, is to store invariants and present them to the user. We have developed methods to filter invariants and compress them at analysis time, and store them to the disk. We then developed a graphical interface to interactively navigate the invariants after the analysis has completed. A screenshot of this interface is presented in Fig. 6.3. The invariant storage mechanism and the graphical interface was used as a starting point by AbsInt to develop an industrial-strength interface for the industrial version of Astrée.

6.2.5 Industrial applications

We describe succinctly our experience adapting Astrée to industrial programs. More information on the analyzed programs is also available on Astrée's Web page [BCC$^+$].

A first application of Astrée was the analysis of two families of avionic embedded control-command applications. We present in Fig. 6.4 benchmark analyses on programs of increasing size in each family. The analyses are performed on our 64-bit 2.66 GHz Intel server. We started from small representative program fragments and ended analyzing several revisions of the complete program. The initial development of Astrée and its specialization to the first family was performed from 2001 to 2003, while the specialization for the second family was performed from 2003 to 2004. Compared to the first family, software in the second family are much larger and perform more complex computations; they use a different code generator and different macro-instructions. Nevertheless, it was possible to reuse all the abstractions developed for the first family, which considerably sped up the analysis refinement. We feel that the analysis times reported in Fig. 6.4, a few tens of hours, are sufficiently low to enable the use of Astrée in production: it is only a fraction of the time devoted to testing. More importantly, industrial users report zero alarm when analyzing production versions of the software [DS07], thereby achieving a proof of absence of run-time error in a realistic, industrial context.

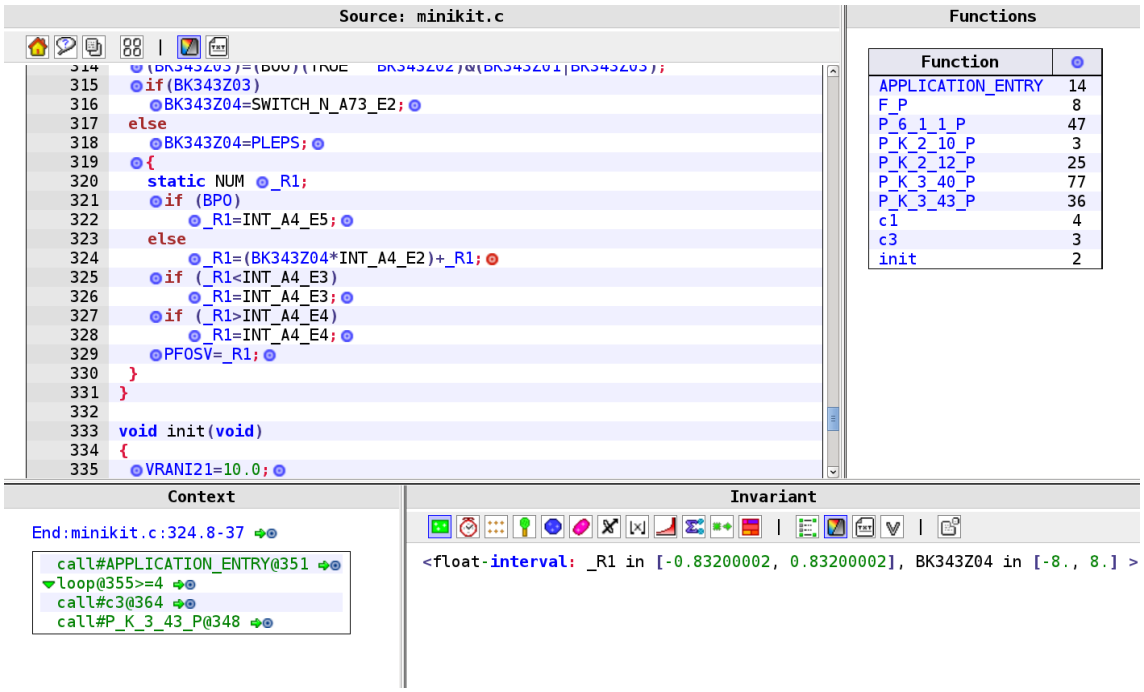


Figure 6.3: Academic graphical user interface for Astrée

# lines	time	memory	alarms
370	5s	205 MB	0
70,000	2h 10mn	740 MB	2
166,000	6h 14mn	1.2 GB	10
82,000	41mn	588 MB	2
290,000	7h 2mn	1.2 GB	3
492,000	13h 21mn	2.2 GB	2
647,000	22h 40mn	2.2 GB	13
808 800	50h 13mn	2.7 GB	1

Figure 6.4: Analysis with Astrée of two families of avionic applications.

A second application of Astrée was the analysis of space software. From 2006 to 2008, we specialized Astrée to analyze a C version of the Monitoring and Safing Unit software of the Automated Transfer Vehicle built by the European Space Agency (the original version is written in Ada, which is not supported by Astrée). Most of the domains developed for avionic applications were useful, which is natural as both applications are synchronous embedded control-command software with float computations. Additionally, a new domain was developed by Feret in order to analyze quaternions [BCC+10a], which are a specificity of space software which we never encountered when analyzing avionic code. After specialization, the absence of run-time error could be proved by Astrée in under 1h of analysis time. This experiment provides some information on the cost, in term of research effort, required to adapt Astrée to a new application domain.

Finally, the industrialization of Astrée by AbsInt [Abs] in 2009 considerably broadened the set of analyzed programs, triggering a new round of refinements. For instance, we developed the integer domains described in Sec. 5.1 to improve the precision when analyzing code generated by TargetLink [dSp], a popular back-end for Simulink heavily used in the automotive industry.

Even after specializing Astrée to a family of programs, the analysis can be easily adapted to handle new software in the same family (such as new versions or corrections). Ideally, this adaptation should only require fine-tuning by the end-user of the user-visible configuration options, and not any modification by the analyzer developers to the core of the analyzer or its domains. This is indeed the case, according to some of our end-users [DS07, SD07]. This validates the claim that the specialization by abstraction refinement does not targets a single program, but a whole family of similar programs, and that specialized analyzers can be a useful tool to help the production of verified programs in an industrial context.

6.3 AstréeA: detecting run-time errors in concurrent embedded C software

AstréeA is an extension of Astrée that focuses on the static analysis of concurrent embedded C software. It is based on the theoretical results described in Chap. 3. Similarly to Astrée, it is also an analyzer designed by specialization which aims towards high precision and efficiency on a given class of applications. Unlike Astrée, however, AstréeA is still a prototype in heavy development: it is being refined on our target software as the precision goal (zero false alarm) has not been reached yet.

The first results on AstréeA were published in [BCC+10a]. We refer the reader to [Min12d] and AstréeA's Web site [CCF+] for the most up-to-date description of AstréeA.[1]

[1] Some publications still refer to AstréeA using its former name, "Thésée." The name was changed to better reflect the lineage to Astrée,

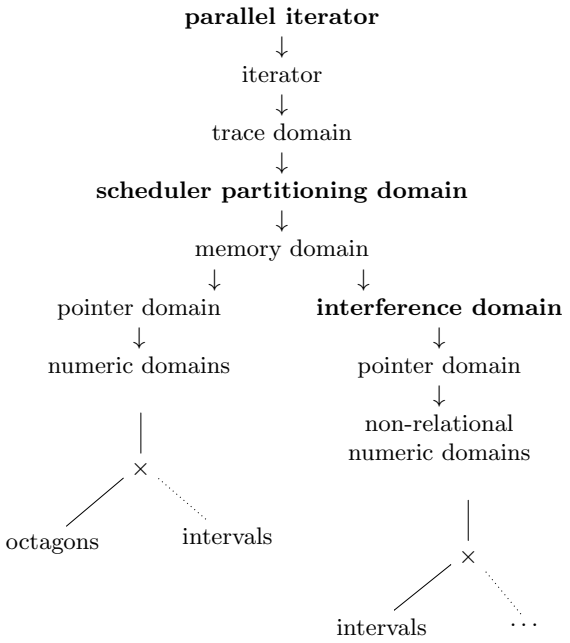


Figure 6.5: Abstract domain hierarchy in AstréeA. Added domains with respect to Astrée (Fig. 6.2) are shown in boldface.

6.3.1 Architecture

AstréeA directly benefits from Astrée's front-end, iterator, and multiple abstract domains. While the global architecture from Fig. 6.1 is still valid, the domain hierarchy presented in Fig. 6.2 is now enriched to give that of Fig. 6.5 (where modifications are shown in boldface).

Firstly, we added a new parallel iterator. It iterates individual thread analyses until the inferred interferences are stable. Astrée's classic iterator that iterates by induction on the syntax is still present: it is used, without modification, to perform each thread analysis. The analysis is thus effectively thread-modular.

Secondly, we added a scheduler partitioning domain. It tracks an abstraction of the scheduler state (such as the set of mutexes currently held by the analyzed thread) and ensures that program states and interferences coming from different scheduler states are abstracted separately, as advocated in Sec. 3.4.1 to analyze precisely critical sections. The scheduler domain also intercepts all the instructions related to synchronization (such as mutex locking) so that all the other domains can completely ignore this aspect of the semantics.

Thirdly, we added a new hierarchy of domains to model thread interferences, which is parallel to the hierarchy modeling local environments. It is actually built by combining existing non-relational domains from Astrée (including non-relational numeric domains, such as intervals, and the pointer domain to abstract sets of pointed-to variables). The memory domain was also modified to drive the interference domain: it feeds it the values written to variables (extracted from the abstraction of the local environments) and queries the interferences on variables that are read. The memory domain takes care of integrating interferences from other threads into each expression before passing them to the underlying pointer and numeric domains, which are completely unaware of interferences. This scheme allowed us to reuse all the numeric and pointer domains from Astrée without any change.

In total, all the changes and additions amounted to only 10% of the size of Astrée and did not require any significant structural modification.

and a final "A" was added to mean "asynchronous" and emphasize the difference with the synchronous programs targeted by Astrée.

6.3.2 Target code

AstréeA targets embedded avionic concurrent C applications. These are increasingly prevalent since the adoption of Integrated Modular Avionics [WW07], which transitions from the use of a network of mono-application processors communicating on a bus to a single processor running concurrent applications communicating in a shared memory. These applications obey the restrictions imposed on Astrée which are related to the embedded critical nature of the target software: there is no dynamic memory allocation nor recursive call. Additionally, there is no dynamic creation of thread nor of synchronization object.

Analyzed code. The particular software we currently focus on is a large industrial application provided by our industrial partner. It consists of 1.7 Mlines of C code and 15 threads,[2] corresponding to different services that run concurrently and communicate implicitly through the shared memory and explicitly through synchronization objects offered by the system. It runs under an operating system based on ARINC 653 [Aer], an avionic specification describing a real-time operating system. The analyzed program is quite complex and heterogeneous. While some services contain code generated automatically from a block-diagram specification, similar to the software targeted by Astrée, other services are hand-written and exhibit a large variety of programming styles and C idioms. In particular, some services implement string formatting, manipulations of arrays (such as sorting), messages to implement network protocols, and even linked lists (where individual cells are allocated from static array pools using custom allocators due to the lack of actual dynamic memory allocation service).

Operating system modeling. The ARINC 653 specification [Aer] supports a set of concurrency-related objects, which includes: threads, synchronisation objects (semaphores and events), and communication objects (blackboards and message queues). They are manipulated through a well-documented API. In order to analyze our program, we wrote a set of stubs implementing the ARINC 653 API. To simplify the design of the analyzer, AstréeA implements only a restricted set of low-level synchronization objects, which are moreover identified by simple integers. The stubs must then map high-level ARINC objects to these low-level AstréeA objects; it maintains maps storing the properties of objects and implements the necessary look-up mechanisms. Moreover, the stubs must map the semantics of the rich set of ARINC operations to sequences of lower-level ones understood by AstréeA. For instance, AstréeA models only simple mutexes that block forever, as described in Sec. 3.4.1. Nevertheless, an ARINC lock with a timeout can be

[2] Individual execution units in a shared memory are called "processes" in the ARINC 653 terminology. Most other operating systems call these "threads" and reserve the word "process" to denote execution units with their own memory space. To avoid any confusion, we use the term "threads" here.

modeled as a non-deterministic choice that either successfully locks the mutex or returns with a failure. The system model is approximately 2,500-line long.

Our stubs do not define a concrete implementation of an operating system, but rather an abstract modeling that soundly includes all the behaviors documented in the ARINC 653 specification and may also exhibit extra behaviors. For instance, as AstréeA does not model the physical time, any time-related property is modeled abstractly as a non-deterministic wait (i.e., a yield). Moreover, ARINC 653 features concurrency primitives that are not yet supported by AstréeA. The main example is that of events, which can be set, reset, or waited on by threads to perform a form of synchronization. As events only restrict the set of thread interleavings, it is sound to ignore them, which is what AstréeA currently does. This naturally results in a loss of precision that we wish to address in future work.

We found it is more convenient to specify the operating system in the target language of the analyzer, C, as much as possible, rather than in the analyzer itself. Such a model can be easily inspected by the end-user and modified to suit a particular instance of an ARINC 653 implementation. We also hope to build models of other systems while reusing AstréeA's low-level primitives for concurrency.

Execution model. The execution model enforced by the ARINC 653 specification ensures that all the concurrency objects (including threads and mutexes) are created during a mono-thread initialization phase, prior to the multi-thread phase, where they are used. Our analysis also works in two phases: the first one analyzes the initialization code and collects the set of concurrency objects it creates. Then, the actual multi-thread analysis proceeds from the entry-points of the collected threads. Hence, there is no real dynamic creation of threads.

ARINC 653 specifies a real-time operating system, where threads have fixed and distinct priorities and these are obeyed strictly by the scheduler. Moreover, our target application schedules all its threads by time-sharing on a single execution unit. Hence, only the unblocked thread with highest priority actually runs and there is no true parallelism. AstréeA uses the scheduling semantics described in Sec. 3.4.2, which is more precise than a semantics assuming arbitrary preemption and true parallelism.

As we do not have any information on the memory consistency model in case of data-races, we err on the safe side and use the model proposed in Sec. 3.5, which is quite general and justifies the flow-insensitive abstraction of non synchronized interferences.

Run-time errors. AstréeA reports the same set of run-time errors as Astrée. AstréeA also reports data-races, i.e., variables that can be accessed by two threads, one access at least being a write, while the threads do not lock a common mutex. After a data-race, the analysis continues assuming a weakly consistent memory semantics: the value written to the variable may be visible by any read that forms a data-race with the write. Additionally, AstréeA reports any violation of the ARINC 653 API (such as creating a thread while in multi-thread mode) through the use of assertions in the ARINC 653 stub. Our target application never issues blocking calls and systematically uses timeouts. Hence, there is no dead-lock by construction.

6.3.3 Results

Following the design by refinement that made the success of Astrée, we focused on a single program in a well defined family of applications, and started refining our analysis in order to approach the zero false alarm goal. We performed all our analyses on a 64-bit 2.66 GHz Intel server.

In order to test our idea, we first considered, in 2009, a lightweight version of our target software, reduced to a functional fraction composed of 100 Klines and 5 threads. After some initial refinements, we could analyze this slice in 1h and find 64 alarms. In 2010, we turned our attention to the full code, consisting of 1.7 Mlines and 15 threads. Initial analyses exhibited around 12,000 alarms. This number could be reduced to around 7,000 in early 2011, and then to 2,000 in early 2012. Our latest analyses now exhibit 1,208 alarms, for an analysis time of 43h. An important remark on efficiency is that only six iterations of the parallel iterator are required into order to stabilize the abstract interferences. Intuitively, it means that the analysis of the concurrent software is not much more costly (around six times) than an analysis of a synchronous program of the same size and complexity. The analyzer is not very efficient in memory as 32 GB of memory are needed for the analysis to proceed. This is due in part to the scheduler partitioning, which duplicates abstract elements (in average, we manipulate four partitions in the abstract environments and 52 in the abstract interferences; note however that abstract interference partitions are inexpensive as they are flow-insensitive and non-relational).

The improvement in precision could be achieved by an iterative refinement of our abstraction. A significant improvement in precision was brought by the addition of the scheduler partitioning domain and the ability for the analysis to exploit the real-time features of the system. This is naturally explained by the fact that the program exploits heavily these features (in particular to avoid locking when priorities are sufficient to ensure mutual exclusion). We note, however, that the large majority of alarms could be removed by improving the abstract domains inherited from Astrée and by adding new domains, unrelated to the issue of concurrency. In particular, we brought some improvements to the memory and pointer domains, as well as the integer numeric domains used to model pointer offsets and the trace partitioning domain (helping to model disjunctions); this helped improving the precision when analyzing strings, large arrays, message buffers, and linked lists.

We refer the reader to [Min12d] for more information on our experiments.

6.3.4 Future work

The design of AstréeA is much a work in progress, as is our research on the analysis of concurrent programs by abstract interpretation described in Chap. 3 on which it is based.

A first avenue of future work is a theoretical one. Chapter 3 ends with a set of examples that we cannot precisely analyze as they require relational or flow-sensitive abstract interferences. Some examples are inspired by current false alarms in our target program. Our framework needs to be extended before new abstractions can be defined.

Secondly, we wish to reduce the number of alarms on our target program. Developing new interference abstractions will surely be necessary, but not sufficient. Many remaining alarms are caused by imprecisions that are not related to concurrency,

but rather to the use of strings, lists, and buffers in the target program. We believe that we have reached the limit precision that can be achieved on these data-structures with the generic memory abstraction used in AstréeA (Sec. 5.2). In order to improve the analysis further, it seems necessary to develop specialized memory abstractions, with a built-in knowledge of the data-structures that are abstracted. This echoes our work on Astrée, where generic numeric abstractions were supplemented with specific numeric abstractions tied to each application domain. For instance, the precise analysis of lists might be achieved by a dedicated shape abstract domain, that would supplement our current memory domain through a reduced product. A mid-term goal consists in achieving by specialization a similar result as Astrée did: the proof of absence of run-time error of an actual industrial (concurrent) program.

Finally, we would like to extend AstréeA to support the analysis of programs running under alternate operating systems, such as POSIX Threads [IT95] or Autosar (a popular automotive standard). This requires defining the concrete semantics of their scheduler and their synchronization primitives, and modeling all the system calls as we did for ARINC 653. Supporting a variety of widespread operating systems in a sound way is a natural requirement before AstréeA can be industrialized.

Chapter 7

Conclusion and perspectives

Ensuring the correctness of software has been a constant concern since the birth of computers and programming languages, starting with the early program proofs by Alan Turing [Tur49] and the development of the first fully formal systems to reason about programs [Flo67, Hoa69]. Formal verification, once the province of theoreticians, has slowly entered the industry in the last one and half decade, with the realisation that software errors have an important economic impact [Lio96, NIS02]. Industrial semantic-based static analysis tools and, in particular, verification tools based on abstract interpretation, have been made available starting in the early 2000s with the introduction of the Polyspace static analyzer [The], after which other tools followed, including Sparrow [Ya], Code Contracts [LF], Astrée [BCC$^+$], etc.

Our aim is to advance the research in static analysis by abstract interpretation to ensure the safety of software. Our contributions concern the development of new analysis methods, with two focuses: the analysis of concurrent programs (reported in Chap. 3) and the design of abstract domains (reported in Chaps. 4 and 5). These contributions are both theoretical and applied. All the methods we propose are not only proved correct (sound) mathematically, but also implemented and validated experimentally. One part of our research is motivated by more fundamental concerns. This is the case for our work on the links between abstract interpretation and proof methods for concurrent programs (Sec. 3.2). This is also the case when designing new abstract domains extending polyhedra and affine equalities, with a focus on semantic expressiveness and algorithms, and not (yet) on application in existing tools (Chap. 4). Finally, in an attempt to ease the fundamental research on numeric abstract domain, we designed with Bertrand Jeannet the Apron library (Sec. 6.1).

However, another part of our research is motivated by more practical concerns. This is the case when adapting existing domains to handle the semantics of realistic programming datatypes: machine integers, floating-point arithmetic, and C-like data-structures (Chap. 5). As we escape the ideal case of mathematical numbers and the Galois connection based abstract interpretation framework, we use pragmatic methods to guide the choice of abstract properties and the design of abstract operators: we design ad-hoc domains for specific use-cases and rely on experimental evaluations on actual programs to validate their usefulness. Finally, we participated in the design of specialized static analyzers: Astrée (Sec. 6.2) and AstréeA (Sec. 6.3). In order to achieve a good precision and performance, these analyzers target specific classes of programs and properties, namely: proving the absence of run-time error in embedded critical synchronous control-command C software and embedded critical concurrent C software (with a predilection for aerospace software). These are also pragmatic tools as they target actual industrial programs and must compose with existing programming practices and semantic irregularities. Moreover, they are designed by refinement of their abstractions on instances of software in the target family. Their design is thus guided by experimentation.

We end this report with some perspectives for future research, in the short and in the long term.

7.1 Concurrency analysis

Our main research topic has been, for the last few years, and still remains the analysis of concurrent software. Chapter 3 described our early results. It the future, we would like to extend our analysis and improve its precision while keeping its main attractive features: thread-modularity, scalability, parametrization by numeric abstract domains, and the ability to reuse existing analysis methods and implementations available for sequential programs.

Interference abstraction. Chapter 3 concludes with a list of concurrent program examples that our analysis cannot handle precisely. The cause of these imprecisions can be traced back to our flow-insensitive and non-relational modeling of thread interactions. In fact, our latest results (Sec. 3.2) present formally this modeling as an incomplete abstraction of a complete semantics based on rely–guarantee proof methods. We thus wish to explore this connection further and derive new classes of interference abstractions that embed some amount of flow-sensitivity and relationality. We now describe two classes of properties that seem promising to us.

Firstly, we would like to infer lock invariants, i.e., properties that are true outside critical sections. These properties may be invalidated locally by a thread while in a critical section but, as long as all the variables involved are protected by a common lock in all the threads, these modifications are invisible for the other threads, which can thus assume the properties to be uniformly true (i.e., in a flow-insensitive way). Such properties play, on concurrent programs, the same role as class invariants in object-oriented languages, or contracts in procedural languages [LF]; they are key to proving complex properties in a (thread-)modular way. Note that our well synchronized interferences are a very imprecise, non-relational version of lock invariants. We wish to extend them to more expressive invariants by exploiting relational numeric domains. As always when relational domains are concerned, a main challenge consists in ensuring the scalability. One solution would consist in using packing techniques inspired from Astrée [BCC$^+$10a] to limit

the degree of relationality and replace a single large relational abstract element with many small ones.

Secondly, we wish to infer properties that depend on the inter-thread flow of execution. A first example consists in proving that an event in a thread always precedes another event in another thread, which is useful to prove well-initialization properties. A more complex example consists in inferring quantitative properties, relating for instance the number of events generated by different threads using numeric abstractions. Such properties appear naturally in producer/consumer concurrent programming patterns, which are found in many applications. The main challenge consists in describing an abstraction of the inter-thread control flow while keeping a thread-modular analysis and avoiding constructing explicitly all the thread interleavings. A possible solution is to introduce auxiliary and history variables, as used classically in proof methods for concurrent programs. This would allow us to express, in the abstraction of the local thread state, properties related to the current location of the other threads or to the history of thread interleavings. The main problem to solve is then to design adequate abstractions to discuss about these properties (for instance, it is necessary to determine precisely the required amount of disjunctive information as we want to avoid using a too expressive, and so, too costly, domain). We would also like to bridge the gap between our interference-based analysis and the fully flow-sensitive but not thread-modular method by Goubault et al. [GH05]. This may allow us to incorporate, in our analysis, their geometric semantics and the abstractions they developed.

Scheduling models. A scheduler controls the execution of concurrent programs by deciding which thread gets to run. Our experience with AstréeA showed us that actual programs rely on the guarantees provided by most schedulers and are not correct when assuming a completely non-deterministic scheduling. This is in particular the case for embedded applications running under real-time operating systems, where schedulers enforce strict policies on the ordering of thread executions.

Our current analysis can only exploit very few properties of schedulers. Indeed, such properties cannot be expressed in our flow-insensitive abstraction of thread interactions. Once flow-sensitive inter-thread abstractions are in place, it will become possible to exploit more information on the ordering of thread executions.

A second concern is that our current analysis can only exploit the case of threads of fixed distinct priorities scheduled on a single execution unit (preventing true parallelism). We revert to non-deterministic scheduling if these hypotheses do not hold. We thus need to extend our analysis to handle more cases. For instance, we wish to support dynamic priority policies, such as priority ceiling and priority inheritance, which are widespread in embedded systems.

A last extension consists in supporting more synchronization objects, such as events, conditional variables, and fences. Indeed, we currently only support mutual exclusion locks and soundly ignore other objects. As scheduler properties, the properties enforced by synchronization mechanisms are related to the inter-thread control flow, and so, will benefit from and motivate the design of flow-sensitive inter-thread abstractions.

Fairness and liveness. In all our work, we restrict ourselves to inferring invariance properties, ignoring liveness properties [LS85]. This limitation comes from the initial abstraction of the maximal trace semantics into the partial finite trace semantics. Intuitively, while invariant properties state that nothing bad ever happens (such as a run-time error), liveness properties state that something good eventually happens (such as terminating with a result). On concurrent programs, liveness can express highly desirable properties, such as the absence of starvation or livelock. One of our long-term goals is the automatic inference of such properties. They have been mainly studied (by model checking) in the case of finite-state models, but seldom in the case of infinite abstract domains which are the norm in abstract interpretation. A possible inspiration comes from existing proof methods for liveness properties. The connections between these proof methods and abstract interpretation have been extensively studied by Radhia Cousot [Cou85]. We still need to design computable abstractions adapted to properties of interest. Another source of inspiration comes from the recent work by Cousot and Cousot [CC12] on proving program termination (a special case of liveness property) using abstract interpretation. It will be necessary to extend the proposed abstractions to the case of concurrent programs.

Once the inference of liveness properties is established, it will become possible to consider proofs under fairness conditions. These conditions restrict the set of program executions by ensuring that threads are not denied indefinitely the right to run. Hence, they correspond to idealized models of schedulers.

Time properties. Another long-term goal consists in inferring properties related to the physical time. Indeed, our analysis currently ignores time, which is not modeled in the program semantics. However, time-related properties are often desirable, in particular in the realm of real-time embedded critical software. While successful methods exist to infer execution time [HF04], these are limited to sequential programs. Hence, we wish to develop time-related abstractions for concurrent programs. We will naturally exploit the numeric abstract domains we developed and design new ones adapted to time-related invariants.

Time-related properties are more precise than liveness ones as, in addition to stating that some expected event will occur, they can also bound the time it will take for it to occur. In particular, liveness alone is not sufficient to verify programs that feature intermittent starvation and livelock phenomenons. However, a timing analysis may bound the duration of a starvation or a livelock, and prove that they are acceptable.

Applications. Our analysis of concurrent programs was initially motivated by the success of the Astrée analyzer on synchronous embedded C software, and the desire to extend this success to the ever growing set of concurrent embedded C software. Hence the design of the AstréeA analyzer. As reported in Sec. 6.3, AstréeA still exhibits more that a thousand alarms on our main target program. Hence, a natural future work consists in improving our prototype and, in the mid-term, get closer to the zero false alarm goal. This will require in particular more precise abstractions of thread interferences and schedulers. The analysis with AstréeA of our target code will thus serve as an incentive, guide, and validation for these abstractions. However, it will surely also trigger the design of abstractions that are not related to concurrency and will enrich the ever growing library of abstract domains that are motivated by practical uses-cases and validated experimentally.

In the mid-term, we also wish to industrialize AstréeA, as it

was done for Astrée. For this, it is not only necessary to prove that the analyzer can reach a high precision by specialization on a selected code family, but also to extend its scope. In particular, we need to extend it to support new concurrency models, scheduler policies, and synchronization primitives from a large set of operating systems (for instance, POSIX Threads and Autosar).

7.2 Numeric abstractions

The design of numeric abstract domains was the main subject of my PhD [Min04b], and I continued working on this topic on a fundamental level with the design of affine domains (Chap. 4) and, on the practical level, with the design of domains for machine integers and floats (Chap. 5).

In future work, however, we wish to explore two different and novel applications of numeric abstract domains: the design of abstract under-approximations, and the connection between abstract interpretation and constraint programming. Our research on these two subjects started very recently and we did not care to report here the very preliminary results we obtained, preferring instead to present them as perspectives for future work.

Under-approximations. The large majority of abstractions considered in abstract interpretation correspond to over-approximations. This is in particular the case of all the abstractions presented in this report. Existing methods to achieve under-approximations either employ actually exact numeric abstractions (as in disjunctive completions, which are costly) or are restricted to deterministic programs (which makes it difficult to handle program inputs or floating-point rounding errors).

In [Min12b], we introduced under-approximating backward operators for the polyhedra domain that solve both problems: they permit an abstraction to a given class of properties (here, affine inequality constraints) and an effective approximation to achieve a cost versus precision trade-off. In the presence of loops, it is necessary to under-approximate greatest fixpoints, and we designed an effective lower widening operator (theorized by Cousot [Cou78] but never effectively constructed before) to compute such an approximation in finite time. Applications include the inference of sufficient pre-conditions ensuring program correctness (e.g., function or class contracts), or the derivation of definite counter-examples. However, our construction is very preliminary and mostly untried. Much work is still required to refine our design, construct new polyhedral abstract operators, and consider under-approximations in other numeric abstract domains.

A main theoretical issue is that, while most abstract domains enjoy a simple notion of best over-approximations (at least on the semantic level), such a notion seldom exists for under-approximations. Several solutions for this problem have been proposed, such as the construction by Schmidt [Sch06]; while these are aesthetically pleasing on a theoretical level, they do not seem very practical to us as they incur a change in domain expressiveness and algorithms, which may degrade scalability (such as powerset or lower closure constructions). Our pragmatic solution consists in designing non-optimal under-approximating operators guided by the properties we need to prove on selected examples. Much more work is required in that direction, including a systematic experimentation on a realistic code base. In addition to the design of new operators in existing domains, we also wish to uncover new classes of properties that may better match those that appear naturally when inferring pre-conditions or counter-examples, and then design the corresponding abstract domains.

Constraint programming. In a collaboration with Marie Pelleau, Charlotte Truchet, and Frédéric Benhamou, we studied the connections between constraint programming and abstract interpretation. In particular, we exhibited in [PMTB13] a few semantic and algorithmic correspondences between these two fields, and used them to design a constraint solving algorithm based on abstract interpretation principles. It is parametrized by the choice of a numeric abstract domain and, unlike classic solving algorithms, enables the use of relational domains, as well as reduced products of domains to handle mixed integer-real constraint programming problems.

The connection between the two fields seems deep and we wish to pursue our collaboration in order explore it further. On the theoretical side, we wish to understand more precisely how fixpoint solving with iteration compares in both fields. An open question is whether there exists a constraint programming analogue to widenings. On the practical side, a possible future work consists in improving abstract interpretation techniques by using methods from constraint programming. In particular, constraint programming features techniques for precise post-fixpoint refinement by decreasing iterations, which are much more advanced than the narrowings used in abstract interpretation. Moreover, the split operators used in constraint programming could be useful to refine partitioning and disjunctive completion domains used in abstract interpretation.

Bibliography

[ABBM10] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *Proc. of the 37th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages (POPL'10)*, pages 7–18. ACM, Jan. 2010.

[Abs] AbsInt, Angewandte Informatik. Astrée run-time error analyzer. `http://www.absint.com/astree`.

[Aer] Aeronautical Radio Inc. ARINC 653. `http://www.arinc.com`.

[AG96] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Comp.*, 29(12):66–76, 1996.

[AGG08] X. Allamigeon, S. Gaubert, and E. Goubault. Inferring min and max invariants using max-plus polyhedra. In *Proc. of the 15th Int. Static Analysis Symp. (SAS'08)*, volume 5079 of *LNCS*, pages 189–204. Springer, 2008.

[AI99] ANSI Technical Committee and ISO/IEC JTC 1 Working Group. Rationale for international standard, Programming languages, C. Technical Report 897, rev. 2, ANSI, ISO/IEC, Oct. 1999.

[AKL+11] J. Alglave, D. Kroening, J. Lugton, V. Nimal, and M. Tautschnig. Soundness of data flow analyses for weak memory models. In *Proc. of the 9th Asian Symp. on Programming Languages and Systems (APLAS'2011)*, volume 7078 of *LNCS*, pages 272–288, Dec. 2011.

[AMSS11] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *Proc. of 17th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)*, volume 6605 of *LNCS*, pages 41–44. Springer, Mar. 2011.

[ATSCOI97] AT & T and The Santa Cruz Operation Inc. System V application binary interface, 1997.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[BCC+] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée static analyzer. `http://www.astree.ens.fr`.

[BCC+02] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, pages 85–108. Springer, Oct. 2002.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. of the ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'03)*, pages 196–207. ACM, June 2003.

[BCC+07] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *Proc. of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'07)*, pages 211–224. ACM, 2007.

[BCC+09] O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, E. Goubault, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, and M. Turin. Space software validation using abstract interpretation. In *Proc. of the Int. Space System Engineering Conf., Data Systems in Aerospace (DASIA'09)*, volume SP-669, pages 1–7. ESA, May 2009.

[BCC+10a] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace*, number 2010-3385 in AIAA, pages 1–38. AIAA (American Institute of Aeronautics and Astronautics), Apr. 2010.

[BCC+10b] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. *Proc. of the 3rd IEEE Int. Workshop on UML and Formal Methods (UML&FM'10)*, 36(1):1–8, Nov. 2010.

[BCC+11] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. L'analyseur statique Astrée. In *Utilisation industrielles des techniques formelles : interprétation abstraite*, pages 67–114. Hermes Science, Jun. 2011.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of the Workshop on Tools and Algorithms for the Construction and Analysis of System (TACAS'99)*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[BDES12] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In *Proc. of the 10th Int. Symp. on Automated Technology for Verification and Analysis (ATVA'12)*, volume 7561 of *LNCS*, pages 167–182. Springer, 2012.

[BGGP99] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revisiting hull and box consistency. In *Proc.*

of the 16th Int. Conf. on Logic Programming, pages 230–244, 1999.

[BHRZ05] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1–2):28–56, Oct. 2005.

[BHZ04] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. In *Proc. of the 5h Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, volume 2477 of *LNCS*, pages 135–148. Springer, 2004.

[BHZ08] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[BK11] J. Brauer and A. King. Transfer function synthesis without quantifier elimination. In *Proc. of the 20th European Symp. on Prog. (ESOP'11)*, volume 6602 of *LNCS*, pages 97–115. Springer, Mar. 2011.

[BKM05] F. Benoy, A. King, and F. Mesnard. Computing convex hulls with a linear solver. *Theory and Practice of Logic Programming*, 5(1–2):259–271, 2005.

[Bou93] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the Int. Conf. on Formal Methods in Programming and their Applications (FMPA'93)*, volume 735 of *LNCS*, pages 128–141. Springer, June 1993.

[BR04] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. of the Int. Conf. on Compiler Construction (CC'04)*, number 2985 in LNCS, pages 5–23. Springer, 2004.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35:677–691, 1986.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. of the 2d Int. Symp. on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, Jan. 1977.

[CC79a] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

[CC79b] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conf. Rec. of the 6th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'79)*, pages 269–282. ACM Press, New York, NY, 1979.

[CC84] P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, New York, NY, USA, 1984.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, Aug. 1992.

[CC92b] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In *Proc. of the Int. Workshop on Programming Language Implementation and Logic Programming (PLILP'92)*, volume 631 of *LNCS*, pages 269–295. Springer, 1992.

[CC04] R. Clarisó and J. Cortadella. The octahedron abstract domain. In *Proc. of the 11th Int. Symp. on Static Analysis (SAS'04)*, volume 3148 of *LNCS*, pages 312–327. Springer, 2004.

[CC10] P. Cousot and R. Cousot. *A gentle introduction to formal verification of computer systems by abstract interpretation*, pages 1–29. NATO Science Series III: Computer and Systems Sciences. IOS Press, 2010.

[CC12] P. Cousot and R. Cousot. An abstract interpretation framework for termination. In *Conf. Rec. of the 39th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'12)*, pages 245–258. ACM Press, 2012.

[CCF+] P. Cousot, R. Cousot, J. Feret, A. Miné, and X. Rival. The AstréeA static analyzer. `http://www.astreea.ens.fr`.

[CCF+06] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. In *Proc. of the 11th Annual Asian Computing Science Conf. (ASIAN'06)*, volume 4435 of *LNCS*, pages 272–300. Springer, Dec. 2006.

[CCF+07] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Varieties of static analyzers: A comparison with Astrée, invited paper. In *Proc. of the First IEEE & IFIP Int. Symp. on Theoretical Aspects of Software Engineering (TASE'07)*, pages 3–17. IEEE CS Press, June 2007.

[CCF+09] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, December 2009.

[CCM10] P. Cousot, R. Cousot, and L. Mauborgne. A scalable segmented decision tree abstract domain. In *Pnueli Festschrift*, volume 6200 of *LNCS*, pages 72–95. Springer, 2010.

[CES86] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8:244–263, 1986.

[CGJ+00] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc 12th Int. Conf. on Computer Aided Verificatoin (CAV'00)*, volume 1855 of *LNCS*, pages 154–169. Springer, Jul. 2000.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conf. Rec. of the 5th Annual ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM, 1978.

[CH09] J.-L. Carré and C. Hymans. From single-thread to multithreaded: An efficient static analysis algorithm. Technical Report arXiv:0910.5833v1, EADS, Oct. 2009.

[Che10] L. Chen. *Sound floating-point and non-convex static analysis using interval linear abstract domains.* PhD thesis, National University of Defense Technology, Changsha, China, Apr. 2010.

[CMC08] L. Chen, A. Miné, and P. Cousot. A sound floating-point polyhedra abstract domain. In *Proc. of the Sixth Asian Symp. on Programming Languages and Systems (APLAS'08)*, volume 5356 of *LNCS*, pages 3–18. Springer, Dec. 2008.

[CMWC09] L. Chen, A. Miné, J. Wang, and P. Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *Proc. of the 16th Int. Symp. on Static Analysis (SAS'09)*, volume 5673 of *LNCS*, pages 309–325. Springer, Aug. 2009.

[CMWC10] L. Chen, A. Miné, J. Wang, and P. Cousot. An abstract domain to discover interval linear equalities. In *Proc. of the 11th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'10)*, volume 5944 of *LNCS*, pages 112–128. Springer, Jan. 2010.

[CMWC11] L. Chen, A. Miné, J. Wang, and P. Cousot. Linear absolute value relation analysis. In *Proc. of the 20th European Symp. on Programming (ESOP'11)*, volume 6602 of *LNCS*, pages 156–175. Springer, Mar. 2011.

[Cou78] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* Thèse d'État ès sciences mathématiques, Université Joseph Fourier, Grenoble, France, Mar. 1978.

[Cou85] R. Cousot. *Fondements des méthodes de preuve d'invariance et de fatalité de programmes parallèles.* Thèse d'État ès sciences mathématiques, Institut National Polytechnique de Lorraine, Nancy, France, Nov. 1985.

[Cou02] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.

[CPS92] R. W. Cottle, J.-S. Pang, and R. E. Stone. *The Linear Complementarity Problem.* Academic Press, 1992.

[CR00] J. W. Chineck and K. Ramadan. Linear programming with interval coefficients. *Journal of the Operational Research Society*, 51(2):209–220, 2000.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8:569, 1965.

[Dij68] E. W. Dijkstra. Cooperating sequential processes. In *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[dRdBH+01] W.-P. de Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods.* Cambridge University Press, 2001.

[DS07] D. Delmas and J. Souyris. Astrée: from research to industry. In *Proc. of the 14th Int. Symp. on Static Analysis (SAS'07)*, volume 4634 of *LNCS*, pages 437–451. Springer, Aug. 2007.

[dSp] dSpace. TargetLink code generator. `http://www.dspaceinc.com`.

[Est] Esterel Technologies. Scade suite™, the standard for the development of safety-critical embedded software in the avionics industry. `http://www.esterel-technologies.com`.

[Fer01] J. Feret. Occurrence counting analysis for the pi-calculus. *Electronic Notes in Theoretical Computer Science*, 39(2), 2001.

[Fer04] J. Feret. Static analysis of digital filters. In *Proc. of the 13th European Symp. on Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 33–48. Springer, Mar. 2004.

[Fer08] P. Ferrara. Static analysis via abstract interpretation of the happens-before memory model. In *Proc. of the 2nd Int. Conf. on Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 116–133. Springer, 2008.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proc. of the American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–32, Providence, USA, 1967.

[Fra86] N. Francez. *Fairness.* Springer, 1986.

[GDD+04] D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Proc. of the 10h Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 512–529. Springer, Mar. 2004.

[GGP09] K. Ghorbal, E. Goubault, and S. Putot. The zonotope abstract domain Taylor1+. In *Proc. of the 21st Int. Conf. on Computer Aided Verification (CAV'09)*, volume 5643 of *LNCS*, pages 627–633. Springer, June 2009.

[GGP10] K. Ghorbal, E. Goubault, and S. Putot. A logical product approach to zonotope intersection. In *Proc. on the Conf. on Computer Aided Verification (CAV'10)*, volume 6174 of *LNCS*, pages 212–226. Springer, 2010.

[GH05] E. Goubault and E. Haucourt. A practical application of geometric semantics to static analysis of concurrent programs. In *Proc. of the 16th Int. Conf. on Concurrency Theory (CONCUR'05)*, volume 3653 of *LNCS*, pages 503–517. Springer, 2005.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 3rd edition, June 2005.

[GNUa] GNU. GMP: The GNU multiple precision arithmetic library. `http://gmplib.org/`.

[GNUb] GNU. MPFR: The GNU MPFR library. `http://www.mpfr.org/`.

[God94] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. PhD thesis, University of Liege, Computer Science Department, 1994.

[Gra89] P. Granger. Static analysis of arithmetic congruences. *Int. Journal of Computer Mathematics*, 30:165–199, 1989.

[Gra91] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *Proc. of the Int. Joint Conf. on Theory and Practice of Soft. Development (TAPSOFT'91)*, volume 493 of *LNCS*, pages 169–192. Springer, 1991.

[GRS98] R. Giacobazzi, F. Ranzato, and F. Scozzari. Complete abstract interpretations made constructive. In *Proc. of the 23rd Int. Symp. on Mathematical Foundations of Computer Science (MFCS'98)*, volume 1450 of *LNCS*, pages 366–377. Springer, 1998.

[HF04] R. Heckmann and C. Ferdinand. Worst-case execution time prediction by static program analysis. In *Proc. of the 18th Int. Parallel and Distributed Processing Symp. (IPDPS'04)*, pages 26–30. IEEE Computer Society, 2004.

[Hin01] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proc. of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE'01)*, pages 54–61. ACM Press, 2001.

[HLL92] T. Huynh, C. Lassez, and J.-L. Lassez. Practical issues on the projection of polyhedral sets. *Annals of Mathematics and Artificial Intelligence*, 6(4):295–315, 1992.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[Hoa03] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, Jan, 2003.

[IEE85] IEEE Computer Society. Standard for binary floating-point arithmetic. Technical report, ANSI/IEEE Std. 745-1985, 1985.

[Imb93] J.-L. Imbert. Fourier's elimination: Which to choose? In *PCPP'93*, pages 117–129, 1993.

[ISO07] ISO/IEC JTC1/SC22/WG14 working group. C standard. Technical Report 1124, ISO & IEC, 2007.

[IT95] IEEE Computer Society and The Open Group. Portable operating system interface (POSIX) – Application program interface (API) amendment 2: Threads extension (C language). Technical report, ANSI/IEEE Std. 1003.1c-1995, 1995.

[Jan04] C. Jansson. Rigorous lower and upper bounds in linear programming. *SIAM Journal on Optimization*, 14(3):914–935, 2004.

[Jea11] B. Jeannet. Concurinterproc static analyzer, 2011. `http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi`.

[JM06] B. Jeannet and A. Miné. Apron numerical abstract domain library, 2006. `http://apron.cri.ensmp.fr/library/`.

[JM09] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. of the 21th Int. Conf. on Computer Aided Verification (CAV'09)*, volume 5643 of *LNCS*, pages 661–667. Springer, June 2009.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, Jun. 1981.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proc. of the 16th annual ACM Symp. on Theory of Computing (STOC'84)*, pages 302–311. ACM, 1984.

[KFW+09] D. Kästner, C. Ferdinand, S. Wilhelm, S. Nenova, O. Honcharova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, X. Rival, and É.-J. Sims. Astrée: Nachweis der Abwesenheit von Laufzeitfehlern. In *Proc. of Workshop Entwicklung zuverlässiger Software-Systeme (ESS'09)*, page 6, Jun. 2009.

[Kil73] G. Kildall. A unified approach to global program optimization. In *Proc. of the 1st Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL'73)*, pages 194–206. ACM, 1973.

[Kin76] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[KWN+10] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Astrée: Proving the absence of runtime errors. In *Proc. of Embedded Real Time Software and Systems (ERTS2 2010)*, page 9, May 2010.

[LAJ11] G. Lalire, M. Argoud, and B. Jeannet. Interproc static analyzer, 2011. `http://pop-art.inrialpes.fr/interproc/interprocweb.cgi`.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering*, 3(2):125–143, Mar. 1977.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. In *IEEE Trans. on Computers*, volume 28, pages 690–691. IEEE Comp. Soc., Sep. 1979.

[Lam80] L. Lamport. The "Hoare logic" of concurrent programs. *Acta Informatica*, 14(1):21–37, June 1980.

[LAMS04] T. Lev-Ami, R. Manevich, and M. Sagiv. TVLA: A system for generating abstract interpreters. In *Proc. of the 18th IFIP Congress Topical*, pages 367–376. Kluwer Academic Publishers, 2004.

[LeV92] H. LeVerge. A note on Chernikova's algorithm. Technical Report 635, IRISA, 1992.

[LF] F. Logozzo and M. Fähndrich. Code contracts. `http://research.microsoft.com/en-us/projects/contracts/`.

[LF10] F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Science of Computer Programming*, 75(9):796–807, 2010.

[Lio96] J. L. Lions. ARIANE 5, flight 501 failure, report by the inquiry board, 1996.

[LL09] V. Laviron and F. Logozzo. SubPolyhedra: A (more) scalable approach to infer linear inequalities. In *Proc. of the 10th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'09)*, volume 5403 of *LNCS*, pages 229–244. Springer, 2009.

[LS85] L. Lamport and F. B. Schneider. Formal foundation for specification and verification. In *Distributed Systems*, volume 190 of *LNCS*, chapter 5, pages 203–285. Springer, 1985.

[Mak00] A. Makhorin. The GNU Linear Programming Kit, 2000. http://www.gnu.org/software/glpk/.

[Mal10] A. Malkis. *Cartesian Abstraction and Verification of Multithreaded Programs*. PhD thesis, University of Freiburg, 2010.

[Mas93] F. Masdupuy. Semantic analysis of interval congruences. In *Proc. of the Int. Conf on Formal Methods in Prog. and Their Applications (FMPTA'93)*, volume 735 of *LNCS*, pages 142–155. Springer, 1993.

[Mas02] D. Massé. Semantics for abstract interpretation-based static analyses of temporal properties. In *Proc. of the 9th Symp. on Static Analysis (SAS'02)*, volume 2477 of *LNCS*, pages 428–443. Springer, Sep. 2002.

[McM93] K. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[Min04a] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *Proc. of the European Symp. on Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 3–17. Springer, Mar. 2004.

[Min04b] A. Miné. *Weakly relational numerical abstract domains*. PhD thesis, École Polytechnique, Dec. 2004.

[Min06a] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of the ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, pages 54–63. ACM, June 2006.

[Min06b] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[Min11] A. Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *Proc. of the 20th European Symp. on Programming (ESOP'11)*, volume 6602 of *LNCS*, pages 398–418. Springer, Mar. 2011.

[Min12a] A. Miné. Abstract domains for bit-level machine integer and floating-point operations. In *Proc. of the 4th Int. Workshop on Invariant Generation (WING'12)*, number HW-MACS-TR-0097, page 16. Computer Science, School of Mathematical and Computer Science, Heriot-Watt University, UK, Jun. 2012.

[Min12b] A. Miné. Inferring sufficient conditions with backward polyhedral under-approximations. In *Proc. of the 4th International Workshop on Numerical and Symbolic Abstract Domains (NSAD'12)*, ENTCS, page 12. Elsevier, 2012.

[Min12c] A. Miné. Static analysis by abstract interpretation of sequential and multi-thread programs. In *Proc. of the 10th School of Modelling and Verifying Parallel Processes (MOVEP 2012)*, pages 35–48, Dec. 2012.

[Min12d] A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science (LMCS)*, 8(26):63, Mar. 2012.

[MJ84] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6:139–143, apr. 1984.

[Mon07] David Monniaux. Verification of device drivers and intelligent controllers: A case study. In *Proc. of the 7th ACM & IEEE International conference on Embedded software (EMSOFT'07)*, pages 30–36. ACM, Oct. 2007.

[MP95] O. L. Mangasarian and J. S. Pang. The extended linear complementarity problem. *SIAM J. Matrix Anal. Appl.*, 16(2):359–368, 1995.

[MPA05] J. Manson, B. Pugh, and S. V. Adve. The Java memory model. In *Proc. of the 32nd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'05)*, pages 378–391. ACM, Jan. 2005.

[MR05] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzer. In *Proc. of the 14th European Symp. on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 5–20. Springer, Apr. 2005.

[NIS02] NIST. Software errors cost U.S. economy $59.5 billion annually. Technical report, NIST Planning Report, 2002.

[NMW02] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proc. of the Int. Conf. on Principles of Programming Languages (POPL'02)*, pages 128–139. ACM Press, 2002.

[NQ10] D. Nguyen Que. *Robust and generic abstract domain for static program analysis: The polyhedral case.* PhD thesis, École des Mines de Paris, 2010.

[NS04] A. Neumaier and O. Shcherbina. Safe bounds in linear and mixed-integer linear programming. *Math. Program.*, 99(2):283–296, 2004.

[NS07] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation (PLDI 2007)*, volume 42, pages 89–100. ACM, Jun. 2007.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, Dec. 1976.

[OHL+12] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. In *Proc. of 33rd ACM Conf. on Programming Language Design and Implementation (PLDI'12)*, pages 229–238. ACM, 2012.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proc. of the 11th Int. Conf. on Automated Deduction (CADE'92)*, volume 607 of *LNAI*, pages 748–752. Springer, jun 1992.

[PH99] A. Pioli and M. Hind. Combining interprocedural pointer analysis and conditional constant propagation. Technical Report 99-103, IBM, 1999.

[PMTB13] M. Pelleau, A. Miné, C. Truchet, and F. Benhamou. A constraint solver based on abstract domains. In *Proc. of the 14th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'13)*, LNCS, page 17. Springer, Jan. 2013.

[Pug92] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Commun. of the ACM*, 8:4–13, Aug. 1992.

[Pug99] B. Pugh. Fixing the Java memory model. In *Proc. of the ACM Conf. on Java Grande*, pages 89–98. ACM, 1999.

[QR05] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Proc. of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

[RD06] J. Regehr and U. Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *Proc. of the ACM Conf. on Lang., Compilers, and Tools for Embedded Syst. (LCTES'06)*, pages 34–43. ACM, June 2006.

[Rey04] J. C. Reynolds. Toward a grainless semantics for shared-variable concurrency. In *Proc. of the Annual Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*, volume 3328 of *LNCS*, pages 35–48. Springer, Dec. 2004.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[Rin01] M. C. Rinard. Analysis of multithreaded programs. In *Proc. of the 8th Int. Symp. on Static Analysis (SAS'01)*, volume 2126 of *LNCS*, pages 1–19. Springer, Jul 2001.

[RM07] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 29(5), 2007.

[Roh06] J. Rohn. Solvability of systems of interval linear equations and inequalities. In *Linear Optimization Problems with Inexact Data*, pages 35–77. Springer, 2006.

[ŠA08] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *Proc. of the 22nd European Conf. on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 27–51. Springer, July 2008.

[Sch86] A. Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, Inc., 1986.

[Sch06] D. A. Schmidt. Underapproximating predicate transformers. In *Proc. of 13th Int. Static Analysis Symposium (SAS'06)*, volume 4134 of *LNCS*, pages 127–143. Springer, 2006.

[Sch09] D. A. Schmidt. Abstract interpretation from a denotational semantics perspective. In *Proc. 25th Conf. Mathematical Foundations of Programming Semantics (MFPS'09)*, volume 249 of *ENTCS*, pages 19–37. Elsevier, Aug. 2009.

[SD07] J. Souyris and D. Delmas. Experimental assessment of Astrée on safety-critical avionics software. In *Proc. Int. Conf. Computer Safety, Reliability, and Security (SAFECOMP'07)*, volume 4680 of *LNCS*, pages 479–490. Springer, Sep. 2007.

[SJMvP07] V. A. Saraswat, R. Jagadeesan, M. M. Michael, and C. von Praun. A theory of memory models. In *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programs (PPoPP'07)*, pages 161–172. ACM, Mar. 2007.

[SK05] A. Simon and A. King. Exploiting sparsity in polyhedral analysis. In *Proc. of the 12th Int. Symp. on Static Analysis (SAS'05)*, volume 3672 of *LNCS*, pages 336–351. Springer, Sep. 2005.

[SK07] A. Simon and A. King. Taming the wrapping of integer arithmetic. In *Proc. of the 14th Int. Symp. on Static Analysis (SAS'07)*, volume 4634 of *LNCS*, pages 121–136. Springer, Aug. 2007.

[SKH02] A. Simon, A. King, and J. M. Howe. Two variables per linear inequality as an abstract domain. In *Proc. of the 12th Int. Conf. on Logic based program synthesis and transformation (LOPSTR'02)*, volume 2664 of *LNCS*, pages 71–89. Springer, 2002.

[SR01] A. Sălcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proc. the 8th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming (PPoPP'01)*, pages 12–23. ACM, 2001.

[SSM05] S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Proc. of the 6th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 21–47. Springer, 2005.

[SSO+10] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53, 2010.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. of the 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'96)*, pages 32–41. ACM, 1996.

[Sut05] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30, Mar. 2005.

[Tar55] A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.

[The] The Mathworks. Polyspace static analyzer. `http://www.mathworks.fr/products/polyspace/`.

[Tur49] A. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, 1949.

[Ven04] A. Venet. A scalable nonuniform pointer analysis for embedded programs. In *Proc. of the Int. Symp. on Static Analysis (SAS'04)*, number 3148 in LNCS, pages 149–164. Springer, 2004.

[WL95] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. of the Int. Conf. on Programming Language Design and Implementation (PLDI'95)*, pages 1–12. ACM Press, 1995.

[WL02] J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proc. of the Int. Symp. on Static Analysis (SAS'02)*, volume 2477 of *LNCS*, pages 180–195. Springer, 2002.

[WM11] G. Whyte and D. L. Mulder. Mitigating the impact of software test constraints on software testing effectiveness. *Electronic Journal of Information Systems Evaluation*, 14:254–270, sep 2011.

[WW07] C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *Proc. of the 26th IEEE/AIAA Digital Avionics Systems Conf. (DASC'07)*, volume 2.A.1, pages 1–10. IEEE, Oct. 2007.

[Ya] K. Yi and al. Sparrow. `http://ropas.snu.ac.kr/sparrow/`.

[YHR99] S. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Proc. of the Int. Conf. on Programming Language Design and Implementation (PLDI'99)*, pages 91–103. ACM Press, 1999.

Index of notations

Semantic operators

Index