Thèse

présentée à

L'université Paris Diderot (Paris 7) Sorbonne Paris Cité

pour obtenir le titre de

Docteur

spécialité

Informatique

Checking a Checker

Verasco: a Formally Verified C Static Analyzer

soutenue par

Jacques-Henri Jourdan

le 26 mai 2016

devant le jury composé de :

Antoine Miné	Rapporteur, Examinateur
Tobias Nipkow	Rapporteur
Yves Bertot	Examinateur
Patrick Cousot	Examinateur
Roberto Di Cosmo	Examinateur
David Pichardie	Examinateur
Francesco Logozzo	Examinateur
Xavier Leroy	Directeur de thèse, Examinateur

This document has been generated using commit 2128f4d (Tue Aug 23 21:56:53 2016).

Résumé

Afin de développer des logiciels plus sûrs pour des applications critiques, certains analyseurs statiques tentent d'établir, avec une certitude mathématique, l'absence de certains types de bugs dans un programme donné. Une limite possible à cette approche est l'éventualité d'un bug affectant la correction de l'analyseur lui-même, éliminant ainsi les garanties qu'il est censé apporter.

Dans cette thèse, nous proposons d'établir des garanties formelles sur l'analyseur lui-même : nous présentons la conception, l'implantation et la preuve de sûreté en Coq de Verasco, un analyseur statique formellement vérifié utilisant l'interprétation abstraite pour le langage ISO C99 avec l'arithmétique flottante IEEE754 (à l'exception de la récursion et de l'allocation dynamique de mémoire). Verasco a pour but d'établir l'absence d'erreur à l'exécution des programmes donnés. Il est conçu selon une architecture modulaire et extensible contenant plusieurs domaines abstraits et des interfaces bien spécifiées. Nous détaillons le fonctionnement de l'itérateur abstrait de Verasco, son traitement des entiers bornés de la machine, son domaine abstrait d'intervalles, son domaine abstrait symbolique et son domaine abstrait d'octogones. Verasco a donné lieu au développement de nouvelles techniques pour implémenter des structures de données avec partage dans Coq.

Abstract

In order to develop safer software for critical applications, some static analyzers aim at establishing, with mathematical certitude, the absence of some classes of bug in the input program. A possible limit to this approach is the possibility of a soundness bug in the static analyzer itself, which would nullify the guarantees it is supposed to deliver.

In this thesis, we propose to establish formal guarantees on the static analyzer itself: we present the design, implementation and proof of soundness using Coq of Verasco, a formally verified static analyzer based on abstract interpretation handling most of the ISO C99 language, including IEEE754 floating-point arithmetic (except recursion and dynamic memory allocation). Verasco aims at establishing the absence of erroneous behavior of the given programs. It enjoys a modular extendable architecture with several abstract domains and well-specified interfaces. We present the abstract iterator of Verasco, its handling of bounded machine arithmetic, its interval abstract domain, its symbolic abstract domain and its abstract domain of octagons. Verasco led to the development of new techniques for implementing data structure with sharing in Coq.

Remerciements

Il est des jours où l'on regarde derrière soi et où l'on est ému par le chemin accompli. Évidement, je ne peux m'empêcher de penser à toutes ces personnes sans qui tout cela n'aurait pu arriver. Elles sont si nombreuses que j'en oublierai sûrement : aussi, je m'excuse d'avance auprès de celles et ceux qui liront intrépidement ces lignes en vue d'y voir l'expression de ma plus grande gratitude, mais qui seront déçus de ne pas y être représentés à leurs justes valeurs.

Tout d'abord, mes plus chaleureux remerciements vont à Xavier Leroy, mon directeur de thèse avec lequel j'ai eu le privilège de travailler pendant ces quatre années et demie. En me considérant plus comme l'un de ses collègues que comme son étudiant, Xavier a su me donner une totale liberté dans mes recherches tout en gardant un œil bienveillant sur mon travail pour m'éviter d'explorer des impasses ou me suggérer des améliorations en utilisant son talent pour rendre simples les problèmes complexes. Ma curiosité a pu se nourrir de sa grande culture, allant de la conception des orgues au jeu vidéo GTA en passant par les bizarreries de certaines architectures d'ordinateurs. Enfin, je ne peux que m'incliner devant la patience avec laquelle il a relu chaque phrase de ma thèse, pour corriger mon trop mauvais anglais ou me suggérer des améliorations techniques, tout en restant pragmatique en m'évitant de longues réécritures.

Je remercie chaleureusement mes rapporteurs, Antoine Miné et Tobias Nipkow. C'est un grand honneur de savoir que mon travail a reçu une relecture attentive de leur part, et qu'il a pu susciter leur intérêt. J'imagine bien que certaines parties de ce document aient pu leur parraître inutilement longues ou trop techniques, mais leur intérêt n'en a pas été amoindri pour autant, et je leur en suis reconnaissant. Enfin, je remercie Antoine Miné pour les innombrables corrections qu'il m'a proposées pour ce document.

Je remercie les autres membres de mon jury : Yves Bertot, Patrick Cousot, Roberto Di Cosmo, Francesco Logozzo et David Pichardie. Ils me font l'honneur de se libérer et, pour certains, de se déplacer de loin afin de juger mon travail. Certains me connaissent depuis quelques années déjà : je remercie en particulier Francesco Logozzo et Patrick Cousot de m'avoir initié à l'interprétation abstraite lors de mon stage à Microsoft Research.

Je remercie Arthur Charguéraud, Pierre Courtieu et Julien Crétin, qui ont relu des chapitres entiers de ma thèse alors qu'ils étaient encore à un stade préliminaire, et donc indéchiffrables, et m'ont fait part de leur précieuses remarques.

Je remercie tous les membres du projet Verasco avec qui j'ai travaillé et sans qui, évidement, cette thèse n'existerait pas. Cette coopération scientifique fructueuse, de plus de quatre ans, a débouché, entre autres, sur le développement de l'analyseur statique qui est le sujet de ce document. Au fil des régulières visioconférences entres ses différents membres, notre projet avançait petit à petit. Je remercie tout particulièrement Vincent Laporte, doctorant dans l'équipe Celtique, à Rennes, pour le travail qu'il a accompli, et Sandrine Blazy et David Pichardie, ses directeurs de thèse, pour l'accompagnement qu'ils nous ont apporté. Je remercie Jérôme Feret et Xavier Rival pour l'expertise qu'ils nous ont apportée sur l'analyseur Astrée et Sylvie Boldo et Guillaume Melquiond pour m'avoir fait découvrir le monde merveilleux des nombres flottants. Enfin, j'ai une pensée pour l'équipe grenobloise qui travailla sur le domaine polyhédrique : Sylvain Boulmé, Alexis Fouilhé, David Monniaux et Michaël Périn.

Je suis particulièrement reconnaissant envers Jean-Christophe Filliâtre : depuis qu'il a été mon professeur de compilation en première année d'ENS, il m'a toujours encouragé à faire de la recherche. J'ai un souvenir ému du moment où, au pot de la soutenance de thèse d'Arthur, il m'a présenté à Xavier pour un stage. Je le remercie aussi de m'avoir fait confiance pour prendre en charge les travaux dirigés de son cours de compilation.

L'équipe Gallium a été, pour moi, le contexte rêvé pour faire ma thèse. Non seulement le niveau scientifique y est excellent, mais ses membres en font une équipe où il est agréable de travailler. Je les remercie donc pour les interminables discussions au coin café, souvent techniques mais parfois moins, qui m'ont motivé à prendre la fameuse navette Inria tous les matins pour aller jusqu'à Rocquencourt. En particulier, je remercie Damien Doligez pour sa passion pour l'électronique, Fabrice Le Fessant pour ses débats endiablés, Luc Maranget pour son huile d'olive palestinienne, François Pottier pour ses westerns, Didier Rémy pour son expertise de TEX, et Mike Rainey et Umut Acar pour leur constante bonne humeur. J'ai une pensée particulière pour Jonathan Protzenko dont j'ai pu aprécier les goûts musicaux parfois étranges en partageant un bureau à Rocquencourt ; pour Thomas Braibant avec qui j'ai travaillé sur ce qui est devenu le chapitre 9 de cette thèse ; et pour Gabriel Scherer, son "café-vous-fait", son Gagablog et son dévouement pour OCaml. Je remercie tous les autres jeunes chercheurs et stagiaires que j'ai côtoyés plus ou moins longtemps et qui ont tous contribué à faire de Gallium une équipe unique : Vitalii Aksenov, Thibaut Balabonski, Pierre-Evariste Dagand, Maxime Dénès, Keryan Didier, Jacques-Pascal Delpaix, Benjamin Farinier, Armaël Guéneau, Cyprien Mangin, Filip Sieczkowski et Thomas Williams.

Je remercie le groupe des coureurs du centre Inria Rocquencourt qui m'ont permis de garder une condition physique convenable pendant ces quelques années malgré, il faut le dire, mon manque de motivation. Je me rappelle cette randonnée mémorable que nous avons faite autour du Puy de Sancy avec Jonathan, Sarah et Victorien. Merci aussi à Guilherme, Pauline, Philippe, Renaud et Thierry.

Mon doctorat n'aurait pas été le même sans le séminaire des doctorants du centre Inria Rocquencourt, dont j'ai participé à l'organisation pendant les deux dernières années. Merci à Jonathan et Elisa de nous avoir transmis cette responsabilité, et merci à Georgios, Matteo, Pauline et Thomas d'avoir aussi participé. J'espère que ce séminaire contribuera encore longtemps aux échanges entre les différents thèmes de recherche d'Inria.

Je remercie les "amis d'Ulminfo" pour avoir été présents depuis que nous nous sommes connus lors de notre première année à l'ENS. Présents 24h/24 sur IRC lorsqu'un besoin de procrastination urgent se faisait sentir, ils étaient aussi des confidents avec lesquels partager en cas de besoin. Merci donc à Antoine, Guillaume, Guillaume, Louis, Michael, Pablo, Pierre et Stéphane. J'espère que nous pourrons toujours conserver cette complicité.

J'ai une pensée émue pour ma famille. Je dois dire que cette thèse leur doit beaucoup : parce que je ne serais pas ici sans eux ; parce qu'ils m'ont donné ma curiosité scientifique et mon goût pour les sciences et l'informatique en particulier ; et parce qu'ils m'ont toujours encouragé dans tout ce que j'ai entrepris. Merci donc, Maman, Papa, Adeline, Arnaud, Catherine, Cécile, Claire, Dominique, Nicolas, Nicolas, et leurs enfants.

Pour finir, Marie-Karelle, je te remercie pour tout. Pour avoir été présente tous les jours, pour m'avoir soutenu même lorsque je passais des soirées, des nuits et des week-ends entiers à travailler, pour avoir supporté ma mauvaise humeur lorsque, par exemple, un papier avait été rejeté (à tort, bien entendu), pour accepter mon départ à Sarrebruck sans m'en vouloir... J'ai la chance que tu m'accompagnes et me rendes heureux chaque jour, alors Merci.

Contents

Chapter 0

Introduction

0.1. Software Safety and Formal Methods

We rely heavily on software for many tasks. The amount of work and money needed to develop useful software is enormous, which leads to the development of better tools and methods to ease this task. In particular, one problem that needs to be addressed is the presence of bugs. Bugs, which arise when a program does not behave as expected because of a defect in its design or implementation, seem to be an unavoidable side effect of software development. They range from innocuous easily fixable annoyances for the user to serious defects in critical devices leading to catastrophic consequences such as destruction of very costly equipment or even death. We can cite many infamous examples of software bugs such as the death of patients due to buggy software controlling the Therac-25 radiotherapy equipment [LT93], or the destruction of the first Ariane 5 prototype rocket [ari96].

Therefore, much effort is made to avoid bugs during the development of software and before its deployment. To this end, the most common method is testing: the program is run on a set of inputs in a more or less realistic context and its behavior is checked to correspond to what is expected. Even if this can be very efficient at all the stages of the development, testing suffers from the fundamental issue that it cannot cover all the possible inputs, and hence it can miss bugs. Moreover, in the context of critical software, the coverage requirements needed for tests make them very costly.

As a result, research is conducted to avoid bugs using formal tools relying on mathematics that would prove the absence of bug with strong certitude. These tools differ from *bug finders* (such as, e.g., Clang Static Analyzer or Coverity) that aim at easing the search for bugs by reporting suspicious code to developers: instead, formal verification tools *guarantee* with strong mathematical certitude the absence of some classes of bugs in the program.

Formal verification of software can target very different objectives. We can classify these objectives depending on the kind of formal guarantees established. One of the first goal that can be targeted is the absence of runtime error. In this scenario, the only guarantee given by the formal method is that the program will not have an erroneous behavior. In particular, this does *not* mean that the output of the program will be correct. Depending on the case, the erroneous behaviors handled by the method can include, for example, illegal memory accesses (such as accessing an array out of its bounds, or dereferencing an invalid pointer), arithmetic errors (such as divisions by 0 or overflows), uncaught exceptions or even non-termination.

On the other end of the spectrum, one could want to formally verify the *functional correctness* of programs. In this other scenario, in addition to the absence of runtime error, we formally verify that the program computes the expected result. This goal is more ambitious than the previous one, because of two main reasons. The first difficulty is the need to define precisely what we mean by "expected result". This definition needs to be done using formal tools with mathematical strength. This definition is called the formal *specification* of the program. In the case of complex software, the specification problem is particularly challenging or even sometimes practically impossible. The second difficulty is the complexity of the reasoning needed to prove functional correctness: indeed, the program can use complex algorithms, advanced data structures or an intricate design architecture.

A common obstacle of all verification methods is Rice's theorem. Informally, this theorem states that any non-trivial properties on the semantic properties of programs is not decidable. More precisely, if we have a property P on the behavior of programs that is not always true or false, then there does not exist a program that can decide whether the behavior of a given program meets P. In the context of formal verification of software, this means that we cannot build a universal tool that will guarantee some interesting property for any given valid program. This applies to proving functional correctness, but also to checking the absence of runtime error.

In order to mitigate the impossibility of a universal verification tool stated by Rice's theorem, a wide range of tools have been developed in the area of formal verification, sacrificing either completeness or decidability. Some of them, such as static analyzers based on abstract interpretation, aim at being fully automated, but can often wrongly classify a correct program as incorrect. Moreover, they typically do not have the reasoning strength necessary to prove functional correctness. At the other extreme, some other tools, such as proof assistants, are able to prove correct a large variety of programs, but they need a lot of user interaction. Several of these methods are used throughout this thesis for different purposes: we use the reasoning power of the Coq proof assistant in order to prove the soundness of an automatic static analyzer, and this proof uses some techniques borrowed from deductive verification.

0.2. Context and Goals of this Project

A *static analyzer* based on *abstract interpretation* is a tool that analyzes programs without executing them, and establishes some of their properties, such as the absence of runtime error. A good example of such a static analyzer is Astrée [BCC+02], used to check the absence of runtime error in large critical embedded software. Astrée has been used to verify the fly-by-wire systems used in the A340 and A380 planes, which are large safety-critical programs comprising several hundreds of thousands of lines of code. In order to achieve such a goal, the designers of Astrée needed to develop complex algorithms using difficult mathematics. Therefore, it is far from obvious that the implementation of Astrée itself is free from any bugs, which decreases the strength of the guarantees it provides.

More generally, a natural candidate for formal verification are formal verification tools themselves: indeed, a common criticism made to these tools is that they can *themselves* be buggy, and hence they can miss wrong behaviors in the software they verify. In particular, we propose, in this thesis, to *check the checker*, or, more precisely, to see how far we can go verifying a static analyzer aimed at guaranteeing the absence of runtime error in a program. Several attempts have been made toward the formal verification of static analyzers based on abstract interpretation [Pic05, CKCY13]: in our case, the aim is to build a realistic tool that could be used to check real programs. Our tool, called Verasco, takes as input a

realistic language, C, and contains a complex combination of abstract domains in order to do advanced reasoning on the behavior of the analyzed programs. We worked in cooperation with the designers of Astrée in order to leverage their experience in writing such a powerful analyzer. Of course, Verasco is not as powerful as Astrée, but it shares many of its design ideas. Its main advantage over a non-verified static analyzer such as Astrée is the guaranteed absence of soundness bugs in its design and implementation, owing to the formal proof accompanying its code.

The formal verification of a complex tool such as a static analyzer is an ambitious goal. Fortunately, this project builds on previous achievements in formal verification, in particular CompCert [Ler09a], a formally verified compiler for the C language. It includes, using mathematical tools, the description of its source language, which is a large subset of C99 and the description of its target language, PowerPC, ARM or x86 assembly. The full functional correctness of its implementation is formally proved: it comes with a mathematical argument stating that the behavior of the generated assembly code is permitted by the input C code. CompCert is written using Coq [Coqa], which is both a programming language, allowing us to write software, and a proof assistant, allowing us to prove mathematical properties about this software.

Recent advances in computer-aided theorem proving in Coq, Isabelle/HOL and other proof assistants made possible the development of formally verified software with growing complexity at the cost of a reasonable effort. For example, recent achievements in formal verification of software include the formal verification of the CompCert compiler [Ler09a], of the seL4 [seL] and mCertiKOS [Cer] operating system kernels and of the FSCQ [CZC+15] file system.

Therefore, for Verasco, using Coq, the same implementation and proof language as CompCert, seems natural. We not only use the experience learned from the development of CompCert: indeed, as a formally verified compiler, CompCert describes formally its input language. Hence, reusing in Verasco the formally verified *front-end* of CompCert makes us able to combine the formal guarantees of Verasco and CompCert in order to enforce the absence of runtime error all the way down to the assembly code generated by CompCert. This makes CompCert an essential dependency of our work.

0.3. Contributions and Structure of this Document

In this thesis, we describe the design, implementation and formal verification of Verasco, a static analyzer based on abstract interpretation entirely written within the Coq proof assistant. Verasco comes with a formal proof of soundness guaranteeing that, if the analyzer does not warn for any potential runtime error, then the analyzed program will execute without any runtime error. These runtime errors include arithmetic errors and invalid memory accesses. Verasco handles most of its input language, C99. It contains many different abstract domains, to achieve good precision of analysis: a precise memory abstract domain and several numerical abstract domains, some of which are relational. These abstract domains are organized modularly, using interfaces and specifications, so that they can be easily modified independently of each other.

Verasco is the work of a team, as part of the ANR project Verasco. Most importantly, Verasco received contributions from Vincent Laporte [Lap15] and his advisors David Pichardie and Sandrine Blazy for the preliminary implementation and for the memory abstract domain and from Alexis Fouilhé, Sylvain Boulmé, David Monniaux and Michaël Périn [FMP13, FB14] for the polyhedral abstract domain.

Specifically, this thesis describes the design, implementation and formalization of the

abstract iterator of Verasco, and of most of its numerical back-ends. We detail in the following our contributions:

- We designed modular interfaces between the different components of the analyzer, with their formal specifications (Chapter 3). In particular, we implemented and proved correct a communication system between numerical abstract domains, inspired from that of Astrée.

- We formalized and proved correct a large variety of algorithms and theorems essential to the soundness of Verasco. The proved theorems go from the arithmetic and "bit twiddling" properties of integers and floating-point numbers (Chapter 5, Chapter 6 and Chapter 8) to the symbolic reasoning needed to prove correct, among others, our symbolic abstract domain (Chapter 7). The formalization work also includes the soundness proof of a specially crafted Hoare logic for the C#minor intermediate language and its use for the correctness evidence of the abstract iterator of Verasco (Chapter 4). To the best of our knowledge, some of the methods used to prove properties of this logic are novel.

- We describe in Section 8.2 an improvement of the usual algorithms for the abstract domain of octagons: we give algorithms that keep sparse the representation of difference bound matrices.

- We give in Chapter 9 several techniques to maintain sharing in data structures used in a program developed in Coq. This includes a technique for soundly using the physical equality operator within Coq programs and several techniques for implementing hash-consing.

This thesis is structured as follows. In Part I, we give an introduction to the context of this work: Chapter 1 introduces several tools and methods for formal verification, and Chapter 2 gives an overview of the abstract interpretation methodology for building static analyzers. In Part II, we describe the design of Verasco as a static analyzer: Chapter 3 describes its architecture and many design decisions, and Chapter 4 gives a precise description of the abstract interpreter, which is the module directly in contact with the input program. In Part III, we describe the multiple numerical abstractions present in Verasco: Chapter 5 explains how we deal with overflow and wraparound behavior of bounded machine arithmetic, Chapter 6 reports on the implementation of two non-relational abstract domains (the abstract domain of intervals and the abstract domain of arithmetical congruences), Chapter 7 depicts an abstract domain of symbolic equalities and Chapter 8 describes the weakly relational abstract domain of octagons, together with its accompanying linearization abstract domain. Part IV heads towards conclusion: Chapter 9 describes the methods that we use or could have used for improving the memory sharing in Verasco's data structures, and Chapter 10 concludes.

This document is a more precise and more recent description of Verasco than the description presented in our previous paper at the POPL conference [JLB+15]. We also refer the reader to the thesis of Vincent Laporte [Lap15] for a precise description of the state abstract domain, which is not presented here. The Coq code of Verasco can be downloaded and browsed online at `http://compcert.inria.fr/verasco/`. In the following table, we give an informative correspondence between sections and chapters of this thesis and files of the development:

Files in the development	Description in this thesis
verasco/lib/AdomLib.v	Section 3.1
verasco/AbMemSignatureCsharpminor.v	Section 3.2.1
verasco/AbMachineEnv.v	Section 3.2.2
verasco/AbIdealEnv.v verasco/IdealExpr.v	Section 3.2.3
verasco/AbIdealEnv.v verasco/AbIdealEnvProduct.v	Section 3.3
cfrontend/Csharpminor.v	Section 4.1
verasco/CsharpminorLogic.v	Section 4.2
verasco/CsharpminorIter.v verasco/CsharpminorIterproof.v	Section 4.3
verasco/IdealEnvToMachineEnv.v	Chapter 5
verasco/AbIdealNonrel.v verasco/IdealBoxDomain.v	Section 6.1
verasco/FloatIntervals.v verasco/FloatIntervalsForward.v verasco/FloatIntervalsBackward.v verasco/ZIntervals.v verasco/IdealIntervals.v verasco/IdealIntervalsNonrel.v	Section 6.2
verasco/Zcongruences_defs.v verasco/Zcongruences.v	Section 6.3
verasco/AbVarExpEq.v	Chapter 7
verasco/AbLinearize.v verasco/LinearQuery.v	Section 8.1
verasco/Octagons.v	Section 8.3
verasco/lib/ShareTree.v	Section 9.1

I

Context

Chapter 1

Introduction to Functional Formal Verification of Software

In order to improve the confidence in software, many different technologies have been developed to write programs that meet their specifications. In particular, formal verification methods aim at using formal arguments based on logic and mathematics with the help of computers for guaranteeing the correspondence between a program and its specification. In this chapter, we give a quick overview of some of these techniques. In particular, we emphasize the concepts and methods we use throughout the thesis: operational semantics, Hoare logic, and the Coq proof assistant. We focus, in this chapter, on techniques targeted at proving precise functional specifications: in Chapter 2, we describe abstract interpretation, which usually proves less precise specifications with less user interaction.

1.1. Formalizing Proofs of Programs

Formal verification methods are based on strong mathematical foundations. As a consequence, it is necessary to view programs as well-defined mathematical objects. We give an example here with the `Toy` programming language. Then, we give an overview of different formal verification techniques and tools.

1.1.1. The `Toy` Language and its Operational Semantics

This introduction is based on a simplistic programming language, called the `Toy` language. We take a formal point of view for the description of `Toy`: `Toy` is seen as a mathematical object, so that we will be able to write theorems and proofs on this language.

The syntax of `Toy` is described in Figure 1.1. It depends on a finite set $\mathcal{V}$ of program variables. The expressions of this programming language are built using program variables, standard arithmetical operators and integer constants. A statement (or instruction) of this programming language is either an assignment, a sequence of two statements (to be executed one after the other) or a while loop that repeats its body until the given expression evaluates to 0. Finally, in `Toy`, a program using a variable $x \in \mathcal{V}$ as input is defined by a statement, constituting its body, followed by the `return` construct producing the result value of the program.

Expressions:

$e ::= x, y, z \in \mathcal{V}$		variables
	$\mid \ldots -2, -1, 0, 1, 2 \ \ldots$	integer constants
	$\mid -e \mid e + e \mid e \times e \mid e \div e$	arithmetic operations

Statements:

$s ::= x := e$		assignment
	$\mid s; s$	sequence
	$\mid$ `while` e `do` s `done`	while loop

Programs:

$p ::= x \Rightarrow \ s;$ `return` e

Figure 1.1: Syntax of `Toy`

As an example, the following `Toy` program, when given a non-negative integer as input, returns the factorial of this number:

$x \Rightarrow$ $r := 1;$
`while` x `do`
$r := x \times r;$
$x := x + -1$
`done`;
`return` r

The next steps towards defining `Toy` as a mathematical object is to give meanings to `Toy` programs, that are, for now, bare syntax trees. Such a definition is called a *semantics* of the `Toy` language. There are different kinds of semantics for programming languages: we describe here an *operational semantics* for `Toy`, in *small-step* style. This kind of semantics describes the behavior of `Toy` programs closely to the actual behavior of a machine executing a `Toy` program.

We describe the behavior of the program as a sequence of *machine states*. For `Toy`, a machine state is given by a pair $\langle \rho, l \rangle$ of an environment $\rho : \mathcal{V} \to \mathbb{Z}$ giving integer values to variables, and of a list l of statements that remain to be executed. The first component, the environment ρ, represents the memory of the program: it stores the values of the variables, and is modified when a variable is assigned. The second component, l, represents at which position the machine is in the code. This component stores literally the pieces of the program that remain to be executed before its end.

The semantics of `Toy` statements is given by an initial state, a transition relation between states describing the steps of computations, written $\langle \cdot, \cdot \rangle \to \langle \cdot, \cdot \rangle$, and a final state from which we can extract the result of the execution of the program. Intuitively, this corresponds to giving a formal description of the "physical" behavior of a computer executing the program.

Consider the `Toy` program $x \Rightarrow s;$ `return` e, for some arbitrary statement s and expression e. When given the value v as input, we define the initial state of this program as being the state $\langle \rho^0_{x \leftarrow v}, s :: \texttt{nil} \rangle$, where $\rho^0_{x \leftarrow v}$ is the environment such that $\rho^0_{x \leftarrow v}(x) = v$ and

$$\frac{x \in \mathcal{V}}{\rho \vdash x \Downarrow \rho(x)} \qquad \frac{n \in \mathbb{Z}}{\rho \vdash n \Downarrow n} \qquad \frac{\rho \vdash e \Downarrow n}{\rho \vdash -e \Downarrow -n} \qquad \frac{\rho \vdash e_1 \Downarrow n_1 \qquad \rho \vdash e_2 \Downarrow n_2}{\rho \vdash e_1 + e_2 \Downarrow n_1 + n_2}$$

$$\frac{\rho \vdash e_1 \Downarrow n_1 \qquad \rho \vdash e_2 \Downarrow n_2}{\rho \vdash e_1 \times e_2 \Downarrow n_1 n_2} \qquad \frac{\rho \vdash e_1 \Downarrow n_1 \qquad \rho \vdash e_2 \Downarrow n_2 \qquad n_2 \neq 0}{\rho \vdash e_1 \div e_2 \Downarrow \lfloor n_1/n_2 \rfloor}$$

Figure 1.2: Big-step operational semantics for `Toy` expressions

$\rho^0_{x \leftarrow v}(y) = 0$ if $y \neq x$; and $s :: \texttt{nil}$ is the list of statements containing the statement s only.

The state transition relation of the language is defined by giving rules under which a transition may happen. For example, the rule for the sequence statement is the following:

$$\langle \rho, (s_1; s_2) :: k \rangle \to \langle \rho, s_1 :: s_2 :: k \rangle \tag{1.1}$$

That is, if we have to execute the sequence of two statements s_1 and s_2, followed by a list k of other statements, we just queue the execution of s_1 followed by s_2 and k.

The rule for assignment depends on the semantics of `Toy` expressions. We write $\rho \vdash e \Downarrow v$ to denote the fact that, in environment ρ, the expression e evaluates to the value v. The definition of this judgment is given in Figure 1.2. It follows the structure of the syntax tree of the expression: this method for defining the semantics of expressions is said to be in *big-step style*. We do not give more detail on this definition, the understanding of which is not essential for the following. The rule for the assignment follows:

$$\frac{\rho \vdash e \Downarrow v}{\langle \rho, (x := e) :: k \rangle \to \langle \rho_{x \leftarrow v}, k \rangle} \tag{1.2}$$

where $\rho_{x \leftarrow v}$ denote the environment ρ modified with the new value v for variable x. The rule is stated as an inference rule: if the condition stated above the vertical line is true, then we can deduce the property under the vertical line. This rule says that executing an assignment amounts to evaluating the expression and modifying the environment with a new binding for the assigned variable. The assignment is discarded from the list of statements to be executed.

There are two rules for `while`, depending on whether the expression evaluates to 0 or not. The first one follows:

$$\frac{\rho \vdash e \Downarrow 0}{\langle \rho, (\texttt{while}\ e\ \texttt{do}\ s\ \texttt{done}) :: k \rangle \to \langle \rho, k \rangle} \tag{1.3}$$

That is, if e evaluates to 0, the loop can be discarded and the environment stays unchanged. The other rule follows:

$$\frac{\rho \vdash e \Downarrow v \qquad v \neq 0}{\langle \rho, (\texttt{while}\ e\ \texttt{do}\ s\ \texttt{done}) :: k \rangle \to \langle \rho, s :: (\texttt{while}\ e\ \texttt{do}\ s\ \texttt{done}) :: k \rangle} \tag{1.4}$$

That is, if the expression evaluates to a non-zero integer, we place the body of the loop as the next statement to be executed, followed by another copy of the loop, which will take care of the following iterations once s will be finished. It is worth noting that the body of the loop can modify the environment, so that the expression will not necessarily evaluate to the same value in the following iterations.

It remains to describe the final states of `Toy`. They are of the form $\langle \rho, \texttt{nil} \rangle$, where `nil` is the empty list.

Putting all the pieces together, we can say that a program $x \Rightarrow s; \texttt{return}\ e$ returns value v_o when given input v_i if there exists a sequence of states $S_0, \ldots, S_n$ such that the two following properties hold:

$$\begin{cases} \langle \rho^0_{x \leftarrow v_i}, s :: \texttt{nil} \rangle = S_0 \ \rightarrow \ S_1 \ \rightarrow \ \ldots \ \rightarrow \ S_{n-1} \ \rightarrow \ S_n = \langle \rho, \texttt{nil} \rangle \\ \rho \vdash e \Downarrow v_o \end{cases} \tag{1.5}$$

`Toy` programs can have runtime errors. For instance, when evaluating a division in an expression e, if the divisor evaluates to 0, there is no value v such that $\rho \vdash e \Downarrow v$. As a consequence, if the expression e appears in the next statement to be executed, there is no following state defined by the transition relation $\langle \cdot, \cdot \rangle \rightarrow \langle \cdot, \cdot \rangle$. In this situation, we say that we are in an *erroneous* or *stuck* state. In a practical implementation of `Toy`, this could correspond to an *undefined behavior* (i.e., anything can happen), or to a runtime exception (the implementation stops the execution of the programs and shows an error message). In any case, a programmer wants to avoid these erroneous states.

A fundamental property of this semantics for `Toy` is that a given program can return at most one value: either it does not terminate, or it stops in an erroneous state, or it does terminate, and in this latter case there is only one path to a final state for a given input value. This property is called *determinism*, and is not valid for every programming language. For example, it is well-known [Kre15, ISO99] that the C language does not fully specify the evaluation order of expressions, making it non-deterministic: an implementation of C can choose among several orders for evaluating expressions, leading to potentially different results.

Of course, `Toy` is an over-simplified language compared to real-life languages such as C. Many features are missing, such as functions, complex data-structures or even `if-then-else` statements. We develop in Section 4.1 the syntax and operational semantics of a much more realistic language, the C#minor intermediate language. We refer the reader to [BDL06, Kre15] for formal semantics of C, which is even more realistic as a language.

1.1.2. Reasoning on `Toy` Programs

The operational semantics of `Toy` describes precisely the behavior of programs. It has the right level of detail to prove correct some programming tools, such as compilers [Ler06]. However, using this formalism directly for proving properties of programs is not convenient. To ease this task, formal verification scientists often design another kind of semantics for their programming language. These so-called *axiomatic* semantics are well-adapted for reasoning, but describe the meaning of programs in a way that is further to the actual "physical" behavior of the program, when compared to operational semantics. In order to fill this gap, which can lead to a loss of confidence in the proofs of programs, axiomatic semantics often come with a soundness proof with respect to an operational semantics. Informally, this proof guarantees that any property proved on programs using the axiomatic semantics is actually valid when interpreted in the context of the operational semantics. Soundness proofs for Hoare logic are technical: we omit them in this introductory chapter, but refer the reader to Section 4.2.5 for the soundness proof of such a logic.

We give here an example of an axiomatic semantics for `Toy`, in the style of Hoare [Hoa69]. The idea is to give to each statement of the program a pre-condition and a post-condition. The combination of a pre-condition P, a program statement s and a post-condition Q forms a *Hoare triple*, noted $\{P\}\ s\ \{Q\}$. Both the pre-condition P and the post-condition Q are

logical properties over the values of the variables of the program. The intuitive meaning of the Hoare triple $\{P\}\ s\ \{Q\}$ is that if P holds when the execution of s starts, then s will not produce any runtime error (i.e., in `Toy`, a division by 0), and Q will hold when s terminates, if it does.

Similarly to the transition relation of the small-step operational semantics, we give a set of rules to build Hoare triples. As an example, here is the rule that lets us build a Hoare triple for a sequence of two statements:

$$\frac{\{P\}\ s_1\ \{Q\} \qquad \{Q\}\ s_2\ \{R\}}{\{P\}\ s_1; s_2\ \{R\}} \tag{1.6}$$

Recall that this rule is given in the style of an inference rule: in order to deduce what is under the line, one has to prove what is above the line. This rule says that, to prove correct the Hoare triple $\{P\}\ s_1; s_2\ \{R\}$ for the sequence of the two statements s_1 and s_2, we have to provide an intermediate predicate Q that will be valid in between the execution of s_1 and s_2. Then, we will have to prove that the two corresponding Hoare triples for s_1 and s_2 are also valid.

Another important rule is the consequence rule. It says that if we were able to prove a Hoare triple, then it is always possible to weaken the post-condition and strengthen the pre-condition: strengthening the pre-condition correspond to considering fewer program executions, and weakening the post-condition correspond to lowering our requirements on the final state of the execution of the statement. Formally, the consequence rule is stated as follows:

$$\frac{P' \Rightarrow P \qquad \{P\}\ s\ \{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\ s\ \{Q'\}} \tag{1.7}$$

The pre-condition for an assignment should not only contain the necessary restrictions to validate the post-condition, but also a condition ensuring the expression being assigned evaluates without errors (such as division by 0). To this end, we introduce the notation $e\Downarrow$ to denote the fact that, in the current environment, e evaluates without errors, i.e., does not divide by zero. Then, the rule for assignment follows:

$$\{e\Downarrow \wedge\ P[x \leftarrow e]\}\ x := e\ \{P\} \tag{1.8}$$

In logic, $\wedge$ is the notation for "and": the logical property $P \wedge Q$ holds whenever P holds and Q holds. Moreover, the notation $P[x \leftarrow e]$ denotes the property P where all the occurrences of x have been replaced by e. The rule (1.8) states that, in order to prove that a post-condition P holds after the execution of an assignment, we have first to assume that the expression involved will evaluate correctly. Second, we have to assume something to deduce P, but P may contain references to the variable x that may not have the same value before the assignment. Therefore, we replace all the occurrences of x in P with the expression it is assigned to. It can seem counter-intuitive that we have to do the substitution in the post-condition in order to find the pre-condition. There exists another formulation of this rule using a substitution in the pre-condition, but the deduced post-condition is more complex and involves an existential quantifier.

The last rule of our Hoare logic for `Toy` is the rule for `while` loops. The usual way of reasoning on loops is to exhibit an *invariant*, a logical property that will hold at the beginning of any iteration of the loop. We have to prove that the invariant is preserved by the body of the loop: that is, if it is valid before the execution of the loop body and the loop condition evaluates to a non-zero value, then it should hold at the end of the execution of the body. Moreover, the invariant should guarantee that the condition of the loop does not

have a division by 0. Given all these hypotheses, we can deduce a Hoare triple for the loop, giving the invariant as a pre-condition (the invariant should hold for the first iteration), and the invariant as a post-condition, together with the fact that the loop condition evaluates to 0. Formally, this can be written as follows:

$$\frac{I \Rightarrow e\Downarrow \qquad \{I \wedge e \neq 0\}\ s\ \{I\}}{\{I\}\ \texttt{while}\ e\ \texttt{do}\ s\ \texttt{done}\ \{I \wedge e = 0\}} \tag{1.9}$$

It is worth noting that this rule does not guarantee that the loop terminates: our axiomatic semantic for `Toy` does not ensure termination of programs. We say that such program correctness proofs are *partial*, in contrary to correctness proofs guaranteeing termination, which are called *total* correctness proofs. Our rule for the while loop could be adapted to total correctness, but we do not need total correctness in the context of this thesis.

Finally, suppose we want to prove correct a program $x \Rightarrow\ s; \texttt{return}\ e$. More precisely, suppose we want to prove that, if given the input v, the program either does not terminate or returns a value verifying the logical predicate P_v. Then, the only thing we have to prove is the following Hoare triple for the statement s:

$$\{x = v\}\ s\ \{P_v(e)\} \tag{1.10}$$

The kind of logical predicate that can be used as pre- and post-conditions depend on the context. The Hoare logic we presented is well adapted for languages with variables and possibly arrays, but is not convenient for languages with pointers or references. Often, for pointer-based programs, a variant of separation logic [Rey02] is used. Separation logic gives a formalism for proving properties about ownership and organization of memory, which is useful when proving correct realistic programs.

Hoare logic at work!

We have just given the rules of our Hoare logic for `Toy`, but they may seem abstract. We give here an example of application of this logic. Namely, we are going to prove that the program for factorial given earlier actually computes the factorial of its input. Let n be such an input, with $0 \leq n$. It suffices to prove valid the following Hoare triple:

$$\begin{array}{l}\{x = n\}\\ \quad r := 1;\\ \quad \texttt{while}\ x\ \texttt{do}\\ \quad\quad r := x \times r;\\ \quad\quad x := x + -1\\ \quad \texttt{done}\\ \{r = n!\}\end{array}$$

The body of the program is the sequence of the assignment $r = 1$ and of the loop. Therefore, we have to apply the rule (1.6), and to exhibit an intermediate predicate that is valid in between. A good candidate is $x = n \wedge r = 1$. Then, we have to prove the following

two Hoare triples:

$$\{x = n\}\ r := 1\ \{x = n \wedge r = 1\} \qquad \begin{array}{l}\{x = n \wedge r = 1\}\\ \texttt{while}\ x\ \texttt{do}\\ \quad r := x \times r;\\ \quad x := x + -1\\ \texttt{done}\\ \{r = n!\}\end{array}$$

For the first one, we are tempted to use the rule for assignments (1.8), but the precondition do not have the required form: it only let us prove the following Hoare triple:

$$\{1\Downarrow \wedge\ x = n\ \wedge\ 1 = 1\}\ r := 1\ \{x = n\ \wedge\ r = 1\}$$

We can easily see, however, that $x = n$ implies $1\Downarrow \wedge\ x = n\ \wedge\ 1 = 1$. As a result, we can apply successively the assignment rule (1.8) and the consequence rule (1.7) to deduce the desired Hoare triple.

Similarly, in order to prove the Hoare triple of the loop, we use successively the rule for loops (1.9) and the consequence rule (1.7). We choose as invariant "$0 \leq x \wedge r = \frac{n!}{x!}$". It remains to prove the following lemmas:

$$(x = n \wedge r = 1) \Longrightarrow 0 \leq x \wedge r = \frac{n!}{x!} \tag{1.11}$$

$$\left(0 \leq x \wedge r = \frac{n!}{x!}\ \wedge\ x = 0\right) \Longrightarrow r = n! \tag{1.12}$$

$$\left(0 \leq x \wedge r = \frac{n!}{x!}\right) \Longrightarrow x\Downarrow \tag{1.13}$$

$$\left\{0 \leq x \wedge r = \frac{n!}{x!} \wedge x \neq 0\right\}\ r := x \times r;\ x := x + -1\ \left\{0 \leq x \wedge r = \frac{n!}{x!}\right\} \tag{1.14}$$

The first two properties are the premises of the consequence rule (1.7) and can be proved using basic maths. The last two are the premises of the rule for loops (1.9). The proof of (1.13) is trivial. The proof of (1.14) is also easy: it involves applying the consequence rule (1.7), the sequence rule (1.6) and the assignment rule (1.8) twice.

1.1.3. Weakest Preconditions and Deductive Verification

As can be seen in the example of the factorial program, even proofs of simple programs involve many intermediate steps. Hopefully, many of these steps can be automated, so that much less user interaction is needed.

The first step towards automation is to notice that, for any given post-condition and non-looping statement, it is easy to build the *weakest pre-condition*. The weakest pre-condition is a valid pre-condition for a given statement and post-condition that is implied by all other valid pre-condition. For example, in our Hoare logic for `Toy`, the computation of the weakest pre-condition for assignments is a direct application of (1.8). For computing the weakest pre-condition of a sequence $s_1; s_2$, we first compute for the weakest pre-condition for statement s_2. Then, following (1.6), we use this pre-condition as a post-condition for s_1 and compute the weakest pre-condition of s_1. This latter pre-condition is the weakest pre-condition of $s_1; s_2$.

Continuing the example of the factorial program, the following Hoare triple for the loop body can be automatically deduced by this algorithm:

$$\left\{0 \leq x + -1 \wedge x \times r = \frac{n!}{(x+-1)!}\right\}\ r := x \times r;\ x := x + -1\ \left\{0 \leq x \wedge r = \frac{n!}{x!}\right\} \quad (1.15)$$

In order to prove (1.14) by using the consequence rule (1.7), it remains to prove the simple implication:

$$\left(0 \leq x \wedge r = \frac{n!}{x!} \wedge x \neq 0\right) \Rightarrow \left(0 \leq x + -1 \wedge x \times r = \frac{n!}{(x+-1)!}\right) \quad (1.16)$$

Therefore, in order to prove correct a Hoare triple for a statement that does not contain any loop, we can automatically compute the weakest pre-condition for the given post-condition and statement. It remains to prove that the desired pre-condition implies the computed weakest pre-condition. This can be done manually, but, in practice, most of these so called *verification conditions* can be proved by automated theorem provers, which are able to do basic but tedious logical reasoning efficiently.

We have just described the basic operating principle of deductive verification tools, which are one of the popular approaches to software verification. In this approach, the user gives pre- and post-conditions to pieces of program, such as functions or loop bodies, and the tool tries to prove them using some sort of weakest pre-condition computation and a theorem prover. This category of tools includes, for example, Why3 [Why], FramaC-WP [Fwp], Boogie [Boo] and many others. In order to discharge the generated verification conditions, they use many different automated theorem provers, such as Alt-Ergo [Alt], Z3 [Z3] or CVC4 [CVC]. In case these automated provers cannot prove automatically a verification condition, some tools allow the user to fall-back to a fully manual proof assistant such as Coq [Coqa] or Isabelle [Isa].

A common problem with deductive verification is the fact that weakest precondition computation does not usually handle loops. Indeed, an invariant has to be computed in a way or another, and this process cannot be automated in all cases. That is why many of these deductive verification tools require the user to give the loop invariants explicitly. Others use heuristic algorithms to infer loop invariants. As we will see in Chapter 2, a generic method for inferring loop invariants is provided by abstract interpretation.

1.1.4. Model Checking

Model checking is another software verification technique. We do not use this method at all in the context of this thesis. For the sake of completeness, we give here a rough description of this active research domain.

The idea of model checking is to explore systematically the set of states the program can reach during its execution, and to check that all the paths that may be taken by the system meet the specification. The possible specifications are potentially not limited to a set of erroneous states to exclude: they are usually expressed in temporal logic, enabling the user to express properties over the whole program trace.

A common problem with model checking is that the set of states can be large and potentially infinite, thus impractical to explore. Several solutions exist to circumvent this problem:

- Instead of verifying the actual program that will be executed, the users of model checking often use a model of the program. Such a model is described in an idealized

formalism (such as Büchi automata, Petri nets or even over-simplified programming languages). Typically, the behaviors of such models are much easier to explore systematically than realistic programs which may contain many implementation details.

- The model checking algorithms need not explicitly explore each state one by one. Instead, designers of model checking tools have developed efficient algorithms using symbolic methods. Schematically, they use logical formulae to represent sets of states. To this end, they typically depend on binary decision diagrams, Boolean satisfiability solvers or satisfiability modulo theory solvers.
- Model checking tools may be used to explore only a subset of the set of reachable states. That is, they will enumerate all the states reachable after k computing steps, where k is a parameter of the tool. By doing so, the tool becomes unsound: it may miss a bug in the software, but it is still useful in improving the confidence in the software.

Compared to the other functional verification tools described in this chapter, model checking has advantages: once the model and the specification are described in the right formalism, the tool is mostly automatic. Moreover, when such a tool finds a bug, it also provides an example of an erroneous program execution, which can be useful in debugging. These two advantages make model checking popular in industry. However, it suffers from serious scalability problems that make difficult the use of model checking for large realistic programs.

1.1.5. Dependent Type Systems

Type systems are a popular method to improve reliability and ease development of software. In this methodology, a set of rules lets a type-checker (which is usually a component of the compiler) assign a type to each value to be computed. A type system soundness theorem guarantees that the types computed at compile time are fulfilled at runtime.

Example types are `int` or `bool` for integer values or Boolean values. A more involved type would be `int -> bool` for functions taking an integer and returning a Boolean. Lastly, some type systems allow polymorphism: a value may have, at the same time, all the types specified by a pattern. For example, a function of type `∀ a, a -> a` has all the types `A -> A` for any type `A`.

There exists a huge variety of type systems, letting the user express in the types more or less precise information about the values computed at runtime. Some of them are particularly easy to use: in the ML type system, all the types can be inferred by the type-checker. However, these type systems are not expressive enough to give functional specifications we need in formal verification of software.

In order to do formal verification of software, we need a particular feature of type systems: we need to be able to speak directly about the values of the programs in the types. For example, in these so-called *dependent type systems*, the type of the value returned by a function can give strong properties on this return value, depending on the values of the parameters. The type of a function computing the factorial of its parameter could entail the fact that it returns indeed the factorial of its parameter, or that this result is always positive. For example, the following F* `factorial` function requires that its argument is non-negative and ensures that it always terminates and returns a strictly positive integer:

```
val factorial: x:int{x>=0} -> Tot (y:int{y>=1})
let rec factorial n =
    if n = 0 then 1
    else n * factorial (n - 1)
```

Because dependent type systems are very expressive, they require the user to give complex correctness arguments for proving that some value has some type. For this reason, dependently typed languages most often come with tools for proving logical properties. Traditionally, the Curry-Howard correspondence, which lets us see logical properties as types and programs as proofs of these properties, is used as such a tool. As a consequence, it is difficult to see the frontier between a programming language using dependent types and a proof assistant using programs as proof. Examples of languages that support specifications using dependent types include F* [F*], Idris [Idr], Agda [Agd] and Coq [Coqa].

1.2. The Coq Proof Assistant

In this section, we give a quick introduction to the Coq proof assistant, which implements the language in which Verasco is programmed and formally verified. This introduction is designed to be a starting point to the understanding of the description of Verasco in this thesis. However, a complete introduction to Coq is far beyond the scope of this document. We refer the reader to excellent books on the subject [BC04, Chl13, PCG+15] and to the Coq reference manual [Coqb] for the most technical aspects.

1.2.1. Coq as a Programming Language

Coq implements a well-studied programming language called the *calculus of inductive constructions*. The calculus of inductive constructions can be seen as a simple functional programming language with a very involved type system.

Being basic, this programming language lacks some important features of usual programming languages: most importantly, every term in Coq (that is, every expression, function or program) is *pure*. That is, in Coq, one cannot modify the value of variables. Moreover, the type system of Coq ensures that every program will eventually terminate: if one wants to write in Coq a complex algorithm, she will need to give a proof of termination for this algorithm. These major restrictions over programs written in Coq can make the work of the programmer much more difficult. But this is a major advantage for reasoning about programs: indeed, because all terms are pure and terminate, they can be thought of as mathematical objects. A Coq function can *really* be thought as a mathematical function (without worrying about whether its execution can modify e.g., global variables). As in mathematics or logic, a variable can *really* be thought as an unspecified object (without worrying about its possible change with time).

An interesting feature of the type system of Coq is that *types are terms*. This means that one may write a program that takes types as parameters or return types, and this program is written using the same constructs as usual programs. In particular, this means that types themselves have a type: this type is `Type`.

As in any typed functional programming language, Coq has a type for functions. For instance, `nat -> bool` is the type for functions from natural integers to Booleans. What is particular to dependently typed programming languages such as Coq, is that the type of function can be *dependent*. That is, the return type of a function can depend on the *value* of the parameter. As an artificial example, a function can return a natural integer when its Boolean parameter is `true` and a Boolean when it is `false`. For reasons that will become clear later, the type of such a function would be written `∀ x:bool, if x then nat else bool`. That is, if a function takes a parameter `x` of type `T` and returns a value of type `U(x)`, then the type of the function is noted `∀ x:T, U(x)`. In particular, in the non-dependent case, `T -> U` is simply a notation for `∀ x:T, U`. Often, the type of the parameter `x` can be inferred

from the context, and we can omit it in the notation: `∀ x, U(x)`.

Dependent types for functions may seem artificial. For now, let us note that this construction unifies the concepts of function and of polymorphism. Indeed, consider the OCaml polymorphic function type `'a -> 'a list`: this is the type of all functions taking an object of any type `'a` and returning a list of values of type `'a`. In Coq, this type can be written `∀ T:Type, T -> list T`: that is, a polymorphic function, in Coq, is nothing more than a curried function taking the quantified type as parameter.

1.2.2. The Curry-Howard Correspondence: Coq as a Proof Assistant

The Curry-Howard correspondence lets us use the Coq programming language as a powerful tool to write logical proofs over mathematical objects and programs. The idea is to interpret logical propositions as types and proofs of these propositions as programs having these types.

More precisely, let us examine what would be the proof of an implication $P \Rightarrow Q$: we can see such a proof as an object that, when given a proof of P, returns a proof of Q; this is the actual *meaning* of an implication. Seeing this interpretation from a computer scientist point of view, this amounts to telling that a proof of $P \Rightarrow Q$ is a function taking a proof of P as parameter and returning a proof of Q: such a proof is a *program* of *type* `P -> Q`.

Similarly, consider a proof of the property $\forall x : T, P(x)$, for a type T and a predicate P over values of type T. The *meaning* of the universal quantifier is that such a proof is an object able to build a proof of $P(x)$ when given any x of type T. That is, from a computer scientist point of view, such a proof is a function taking an x of type T as parameter and returning a proof of $P(x)$. This means that the return type of the function representing the proof depends on the *value* of the parameter: the type of this function is the dependent function type `∀ x:T, P(x)`.

This analogy does not stop here: all the usual ways of building logical propositions have an interpretation in terms of types and their proofs in terms of programs of the corresponding type. The logical *and* of two proposition `P` and `Q`, noted `P /\ Q` in Coq can be interpreted as the type of pairs of the proofs of the two propositions; the logical *or* of `P` and `Q`, noted `P \/ Q` in Coq, can be interpreted as a sum type, etc...

Therefore, in Coq, a proposition is a type, and a proof of a proposition is a program with this type. Yet, for several technical reasons, types and propositions are not exactly alike: to differentiate them, we use the type `Type` for types and the type `Prop` for propositions, but they behave in similar ways. In particular, we can actually prove properties by writing programs of the right types, although, in practice, these programs can be large and tedious to write: hence, we instead use *tactics* that automatically generate pieces of programs corresponding to certain well-known reasoning schemes.

The benefit of using the same formalism for programming and theorem proving is that we can mix proofs and programs. Namely, we can mention directly the programs in logical propositions as if they were objects of the logic. We can also manipulate logical properties in programs. For example, we can define a function taking as parameter the proof of a property over another parameter: this is the responsibility of the caller to provide a proof that the parameter verifies the property. This way, we are sure that the function will always be called with valid values for its parameters.

1.2.3. Inductive and Coinductive Types in Coq

Inductive types are a generalization of algebraic data types such as those present in OCaml: they are defined by a set of constructors that let one build elements of the type; then, we

can use pattern matching to operate on values of the type.

As in OCaml and in many functional programming languages, inductive types can be used to define data structures: in the standard library of Coq, in CompCert or in Flocq, inductive types are used to define Booleans, pairs, lists, various sorts of integers, floating-point numbers, syntax trees... To define such an inductive type, one has to give the types of every constructors as curried functions[1]. As an example, in order to define the type of statements of `Toy` as described in Figure 1.1, assuming that the types `var` and `expr` of `Toy` variables and expressions were defined earlier, we would write the following inductive type definition:

```
Inductive stmt: Type :=
| Assign : var -> expr -> stmt
| Sequence : stmt -> stmt -> stmt
| While : expr -> stmt -> stmt
```

That is, we define the inductive type `stmt`, with type `Type`, having 3 constructors, `Assign`, `Sequence` and `While`. The type of the constructor representing `Toy` assignments, `Assign`, is `var -> expr -> stmt`, meaning it should contain two values: a `Toy` variable and a `Toy` expression. Like algebraic data types in OCaml, inductive types can have parameters. For example, we could make the definition of `stmt` generic over the type of `Toy` variables:

```
Inductive stmt (var:Type): Type :=
| Assign : var -> expr var -> stmt var
| Sequence : stmt var -> stmt var -> stmt var
| While : expr -> stmt var -> stmt var
```

Because, as we have seen in Section 1.2.2, types and propositions behave similarly, arguments of constructors can also be proofs. A typical example is the `sig` type from the standard library of Coq:

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
| exist : ∀ (x : A), P x -> sig A P
```

This type is parameterized by `A`, a type, and `P`, a predicate over elements of `A`. It has a single constructor, `exist`. This constructor has two parameters: the first one, `x`, is an element of `A`, and the second one is a proof that this element meets `P`. That is, from a programmatic point of view, `sig A P` corresponds to the type of all elements of `A` satisfying the property `P`. We call such type a *subset type*, and usually we use the notation `{x : A | P x}`, borrowed from set theory. For example, the type `{x : nat | x > 0}` is the type of strictly positive natural integers.

When developing formally verified software, subset types are often useful: they help us maintain invariants over our data structures, by construction, instead of proving them afterwards. That is, instead of manipulating values of type `A` and then proving that the invariant `P` is maintained, we can, from the beginning, manipulate values of type `{x : A | P x}`.

Inductive types are useful to define data structures, but not only. Because of the Curry-Howard correspondence, they can also be used to define logical properties in `Prop`. In fact, apart from implication, universal quantification and negation, all the logical connectives in Coq are defined using inductive definitions in `Prop`. This include `and`, noted `/\`, `or`, noted `\/`, equality, and existential quantification. For example, we give in the following the definitions of `and`, `or` and of the existential quantifier as inductive types, as they appear in the Coq standard library:

[1] Not any type for a constructor is allowed: some technical syntactical restrictions are enforced by the Coq type-checker, but this is out of the scope of this overview.

```
Inductive and (A B : Prop) : Prop :=
| conj : A -> B -> and A B.

Inductive or (A B : Prop) : Prop :=
| or_introl : A -> or A B
| or_intror : B -> or A B.

Inductive ex (A : Type) (P : A -> Prop) : Prop :=
| ex_intro : ∀ x : A, P x -> ex A P.
```

In Verasco, we use inductive definitions in `Prop` for defining some predicates. They are convenient to use when the predicate is defined by a set of inference rules, which is particularly frequent when defining semantics of programming languages. For example, in order to define the small-step operational semantics of `Toy` used in Section 1.1.1, we would define an inductive predicate `step` taking as parameter the initial and final states or a step, as follows:

```
Inductive step : (env * list stmt) -> (env * list stmt) -> Prop :=
| step_Assign:   ∀ ρ e v k, eval_expr ρ e v ->
                   step (ρ,               (Assign x e)::k)
                        (update_env ρ x v, k)
| step_Sequence: ∀ ρ s1 s2 k,
                   step (ρ, (Sequence s1 s2)::k)
                        (ρ, s1::s2::k)
| step_While_0:  ∀ ρ e s k, eval_expr ρ e 0 ->
                   step (ρ, (While e s)::k)
                        (ρ, k)
| step_While_v:  ∀ ρ e v s k, eval_expr ρ e v -> v <> 0 ->
                   step (ρ, (While e s)::k)
                        (ρ, s::(While e s)::k)
```

We assume that `eval_expr`, representing the semantics of expressions and `update_env`, performing updates in environments have already been defined. The constructor `step_Assign` corresponds to rule (1.2), `step_Sequence` to rule (1.1), `step_While_0` to rule (1.3) and `step_While_v` to rule (1.4).

Coinductive types

When we use inductive types, we implicitly assume that the data structure or proof that we are defining is finite. It is impossible, for example, to define a list with infinitely many elements (also known as a *stream*). For this reason, Coq lets us write recursive function definitions, as long as one parameter of the function is an inductive value that strictly decreases at each call: such definitions have no risk to break the property that every Coq program terminates.

There are, however, situations where one would like to build infinite data structures. The typical example is *streams*, which are infinite lists. These infinite data structures are called *coinductive types*. Values of coinductive types can be defined by calling constructors as for inductive types, but the parameters of the constructors can mention the coinductive value being currently built. For example, one can build the stream S containing only ones by saying that $S = 1 :: S$. The constructor :: is called with, as second parameter, the stream S being currently defined.

Coinductive types can also be used to define logical predicates: for example, coinductively defined predicates are used, in Verasco, for defining the axiomatic semantics of the C#minor language in Section 4.2. Such definitions correspond to the usual definition of coinductively defined predicates.

1.2.4. Extracting OCaml Programs from Coq Terms

Running Coq programs can be done directly inside Coq, using Coq's own evaluation mechanisms. However, it may be useful to make formally verified code written in Coq cooperate with arbitrary non-verified code written in another, more convenient language. To this end, we use the so-called *extraction* feature of Coq. Extraction transforms a Coq program into OCaml code, so that it can be used in the context of a larger OCaml program.

One of the reasons for the separation of `Prop` and `Type` is that the extraction mechanism is able to erase from a Coq term all the uses of `Prop` in order to keep only its computational contents. Indeed, as proofs are programs, they can actually compute things, even though we are generally not interested in the result. This is a common problem when executing programs directly inside Coq: it may spend a lot of time in proofs computing things we are not interested in. Extraction lets us use proofs in programs without worrying about the time spent computing in proofs.

Using fuel for avoiding termination proofs

As we explained, every program in Coq terminates. Even if this is a nice property for reasoning, it may be inconvenient when programming. Indeed, the proof of termination of certain algorithms can be tedious. For example, the termination of fixpoint computation with widening can be complex to prove in the case of complex abstract domain combinations. Moreover, termination is not actually important for the soundness of Verasco: if the analysis of some program does not terminate, we do not get any result, including a wrong one.

A solution to this problem is to use *fuel*: we parameterize the whole program with an integer `fuel` parameter. When we program an algorithm of which we do not want to prove the termination, we force the algorithm to terminate after `fuel` steps and return a dummy result if `fuel` was not large enough to get a meaningful result. This makes it obviously terminating from the point of view of Coq. We prove that the analyzer is correct for any value of `fuel`.

For example, assume that we want to define a function searching for an integer for which a given function from natural integers to Booleans returns true. Further assume that we have good reasons to think that there exists such an integer, but we do not want to prove this formally, and we can afford the risk of a non-terminating search in the case such integer would not exist. The search function iterates over the natural integers until finding such an integer, by using a fuel parameter in order to convince the termination checker of Coq that this search is terminating. In the event we exhaust the fuel, the function returns `None`, and otherwise `Some x` where `x` is the integer found by the function. Then, we prove that the returned integer is indeed a value with the desired property for any value of the fuel:

```
Definition find_true (fuel:nat) (f:nat -> bool) : option nat :=
  let fix aux fuel x0 :=
    if f x0 then Some x0
    else match fuel with
      | O => None
      | S fuel => aux fuel (x0+1)
      end
  in aux fuel 0.

Lemma find_true_correct:
  ∀ fuel f res, find_true fuel f = Some res -> f res = true.
Proof. [...] Qed.
```

In order to feed the fuel parameter, we have two solutions: the first possibility is to use a very large integer that would never be exhausted in reasonable time. This cannot be done

reasonably using fuels in `nat`, because of the Peano encoding of this type, but this can easily be achieved by using, e.g., the `positive` type of binary encoded strictly positive integers.

The other solution we use in Verasco is leave the whole program and its correctness theorem parameterized by the fuel. When it comes to execute the program, we give an "infinite natural integer" defined by the OCaml definition `let rec fuel = S fuel`. When running the analyzer, either it does not terminate, and it does not return a wrong result, or it does terminate and return a result. In the latter case, we know that the program has traversed only a finite portion of `fuel`, and hence, there exists another *finite* natural integer that would have made the analyzer return the same result. Therefore, the value returned by the analyzer is correct because the correction theorem is proven for any value of `fuel`.

1.2.5. Axioms

Proving theorems using programs with the Curry-Howard correspondence in the calculus of inductive constructions is a powerful method. However, in order to use the real numbers of the Coq standard library or just for convenience, it is sometimes necessary to assume axioms in our proofs. We summarize here the collection of axioms used in Verasco. This section is for specialists of formal verification in Coq: non-specialists may ignore this section and proceed to the next one. The takeaway of this section is that the formalization of Verasco depends only on harmless axioms.

The hypotheses used in the final soundness theorem of Verasco can be divided into three categories:

Logical axioms. In Verasco, we use three purely logical axioms: proof irrelevance, excluded middle and dependent functional extensionality. We recall here their exact statements:

```
∀ (A:Prop) (a1 a2:A), a1 = a2
∀ P:Prop, P \/ ~P
∀ (A:Type) (B : A -> Type) (f g : ∀ x:A, B x), (∀ x, f x = g x) -> f = g
```

These are widely recognized as harmless axioms and included in many developments using Coq.

Axiomatization of reals. The axiomatization of real numbers from Coq's standard library is used in the Flocq library for describing the real value of floating-point numbers. CompCert and Verasco rely on Flocq for their handling of floating-point numbers. We recall that this axiomatization includes the type of reals itself, the operations over reals and their properties.

Parameters and external functions. The Coq development uses many parameters and external functions replaced with actual values during the extraction to OCaml code. These externals do not carry any logical contents because they have clearly inhabited types and no hypotheses are made on their value, hence they do not endanger the guarantees provided by the logic.

1.3. Other Formal Verification Projects

The use of formal verification led to many different projects aiming at formally verifying large software. Here, we describe briefly a few projects that we find particularly impressive.

We do not claim that this list is comprehensive: our aim is to give an idea of how far we can go when using formal verification tools.

1.3.1. Formal Verification of Programming Tools

CompCert: a formally verified C compiler. CompCert [Com, Ler06, Ler09a, Ler09b, BDL06] is a compiler for most of the C99 language written in the programming language of Coq. It takes as input a C99 program and produces an assembly code corresponding to this input program. It can target x86, PowerPC or ARM assembly. It comes with a formal proof of semantic preservation: if the compiler produces some code without reporting an error, then any observable behavior of the produced program is an observable behavior permitted by the semantics of the source program.

Importantly, CompCert includes many optimization passes, making the performance of the produced code competitive with industrial C compilers with few optimizations activated. Given the amount of research needed to build these unverified compilers, this level of optimization is already impressive.

An essential prerequisite to verifying a compiler is formalizing the semantics of the input and output languages. Therefore, CompCert includes a mathematical definition of the semantics of C99 and assembly, but also of the many intermediate languages it uses. Verasco depends on CompCert, because it uses directly one of its intermediate languages, called C#minor: this way, Verasco avoids the tedious design and proof of a C front-end. By using the same formal semantics for C#minor as CompCert, Verasco and CompCert, when combined, bring formal safety guaranties on the produced assembly code.

Similarly to Verasco, CompCert is directly programmed using the Coq programming language, and uses Coq's extraction mechanism to generate OCaml code that is compiled into an executable.

Additionally to Verasco, CompCert has a large ecosystem of variants adding features nonexistent in the main version. This include, for examples, CompCertTSO [ŠVZN+13] for shared-memory concurrency, Compositional CompCert [SBCA15] for separate compilation and CompCertSSA [BDP14] for further optimizations.

Jinja: formal verification of a Java subset. Jinja [KN06] is a formally verified implementation in Isabelle/HOL of a large subset of the Java language, together with its virtual machine and a Java compiler. Additionally, Jinja includes a type checker for its input language, and a static checker of the definite assignment rule requiring that every variable is assigned before being read. Their work also includes a bytecode verifier, ensuring that the given bytecode program verifies all the properties required by the Java Virtual Machine. The authors of Jinja emphasized the fact that the different tools they formally verified are tightly integrated: for example, the compiler correctness theorem not only states that it preserves semantics, but it also preserves well-formedness. That is, any source program validated by the type-checker will pass the bytecode verifier.

The Verifiable Software Toolchain. The Verifiable Software Toolchain (VST) is a framework [VST] for formally verifying C code using a separation logic for Clight, an intermediate language of CompCert. VST uses the same semantics as CompCert for Clight, so that programs formally verified using VST can be translated into formally verified assembly code using the CompCert compiler. Interestingly, VST is able to verify concurrent code using shared memory.

1.3.2. Formal Verification of Operating System Components

Operating systems, and, in particular, operating system kernels are an essential component of any computer system. The correctness of virtually every software relies on the correctness of the operating system this software is running on. Moreover, they are usually very complex software, mixing low-level code with advanced algorithms. Therefore, the need of formal verification of these programs is important. We present here four projects aiming at formally verifying operating system components.

The seL4 microkernel. One of the problems of the formal verification of operating systems is the usual large size of such software. Therefore, it is natural to verify a *microkernel*, which is an operating system kernel reduced to its strict minimum: a microkernel typically only contains a scheduler, a memory management system and a basic inter-process communication system, but no device drivers, which are usually implemented as independent programs.

L4 is a family of high-performance microkernels, of which seL4 [seL, KEH+09] is a particular variant. The particularity of seL4 is that it is formally verified using the Isabelle/HOL proof assistant. More precisely, the authors of seL4 wrote an abstract specification of the kernel within Isabelle/HOL, refined by an executable specification, which is itself a refinement of their C implementation. These two refinements are proved correct using Isabelle/HOL.

The formal proof of seL4 guarantees that the microkernel primitives always terminate without performing illegal operations (such as illegal memory accesses), and that it complies with the abstract specification. Moreover, it gives strong security guarantees by using a specially crafted access control mechanism.

CertiKOS: certified kit operating system. The CertiKOS [Cer] project aims at certifying operating system kernels. An achievement of this project is the formal verification of mCertiKOS, an operating system kernel. It has several variants, mCertiKOS-hyp is a hypervisor able to boot Linux, and mCertiKOS-emb targets embedded applications.

The authors of mCertiKOS report an impressively low amount of work needed for their formalization (about 1 person year). The originality of their approach is the use of deep specifications and layered programming [GKR+15]. The mCertiKOS kernel is divided into several stacked layers, each of them implementing a component of the kernel. Every layer has a deep specification, which gives an executable, deterministic description of its behavior, using abstract mathematical objects. The layer itself is an implementation of its deep specification using that of the underlying layer and a piece of low-level code. The deep specification of the highest layer gives an abstract specification of the whole kernel, from which one can prove functional properties (e.g., security guarantees...).

The different layers of mCertiKOS are either implemented in assembly or in Clight and compiled with a modified version of CompCert. The result is that, contrarily to seL4, the compiler is not in the trusted code base: only the formal semantics of assembly (and the Coq proof assistant) are relied on.

Verve: a type and memory safe operating system. Verve [YH11] is an operating system the implementation of which is guaranteed to be free of type and memory errors: that is, it targets a less ambitious goal than seL4 and CertiKOS (there is no proof of functional properties), but with a smaller formalization effort. The methodology used by the developers of Verve is to reduce to the minimum the amount of code to verify, and use a typed assembly language, obtained from compilation of C# code, to make sure the rest

of the code is safe. The verified part of the code, called the *Nucleus*, is implemented using the Boogie tool and easily translates into assembly. The Nucleus provides basic runtime features: a basic garbage collector, interrupt handling, multiple stacks and device access.

The FSCQ certified file system. FSCQ [CZC+15] is a file system written and formally verified using Coq. Its authors created a so-called *crash Hoare logic*, enabling them to reason about the recoverability of the state of the file system: they have proved that FSCQ is able to reconstruct a valid state for the file system without loss of data, even if the system crashes at any unexpected point in time. FSCQ is written in Coq, extracted into Haskell and then ran in user space under Linux using Fuse. It has reasonable performances and supports all the core features required by the POSIX standard.

1.3.3. Verification of Security Properties

miTLS: a Verified Reference Implementation of TLS. TLS is a widely used protocol for guaranteeing confidentiality, authenticity and integrity of connections on the Internet. This protocol is the basis of the HTTPS protocol providing security for the World Wide Web. Because of its large complexity, TLS has suffered from many security breaches, coming not only from its implementations, but also from the design of the protocol itself. Any of these breaches are taken very seriously by software makers, because TLS is probably the most used cryptographic protocol in the world.

Because of its complexity and of the previously discovered breaches, doubts still exist about the security of TLS. The TLS implementation miTLS [miT] aims at removing these doubts by formally verifying a reference implementation, and proving theorems about the security of the protocol. The latest release is written in F7 and F#, but the development version uses F* instead.

This reference implementation provides unprecedented security guarantees compared to existing implementations using most often complex C code subject to bugs such as buffer overflows. Moreover, in the process of proving security properties of TLS, developers of miTLS found new security vulnerabilities in the TLS protocol.

Formal verification of a Java Card product. The Gemalto company used a Coq formalization [CN08] as a formal argument for qualifying a Java Card product at the level EAL7 of Common Criteria, which is the highest possible level of certification. Their Coq formalization proves the equivalence between the high-level specification and the low-level design, but the translation between the low-level design and the C implementation is performed by hand and checked informally.

1.3.4. Formally Verified Mathematical Theorems

Proof assistants such as Coq or Isabelle/HOL are primarily targeted at proving mathematical theorems. Therefore, mathematical theories are direct clients of these tools, even before thinking of proving programs. As a result, the use of proof assistants as evidence of the correctness of a theoretical result became routine in some research communities, like at the POPL and ICFP conferences.

Mathematics are not, properly speaking, computer software. We decided to include these projects in this list for two reasons: first, developing the proof of a theorem is a very similar activity as proving correct a program, and second, some of these results can be used in proofs of software.

Flocq: a formalization of floating-point arithmetic. The Flocq library [BM11] is a formalization of floating-point arithmetic within the Coq proof assistant. It is very general: any basis, precision, formats and rounding modes are allowed, from base-2 to base-10 and above, from floating-point formats to fixed-point formats and from rounding to nearest to rounding to infinities. Importantly, all the arithmetic operations are specified in terms of ideal real arithmetic, so that no space is left for a bug in these operations.

In CompCert and, consequently, in Verasco, specific kinds of floating-point numbers are used: the "binary64" and "binary32" floating-point numbers specified by the IEEE754 standard [IEE08]. Flocq contains specialized implementations of all the arithmetic operations that are formally verified with respect to the general reference implementation and optimized for the case of IEEE754 floating-point numbers. CompCert depends on this implementation of floating-point numbers, but as a black-box: it uses few of their properties.

On the other hand, in Verasco, we rely heavily on many theorems over floating-point numbers proved in Flocq. Floating-point numbers are used in the source language but also in the implementation of the abstract domains. Therefore, the Flocq theorems are used to characterize the behavior of floating-point operations in the source language and in the implementation of Verasco itself.

Mathematical Components. Mathematical components is a library, written using the Coq proof assistant, aiming at proving mathematical theorems. It has been successfully used to formalize modern, abstract mathematics. This includes the four color theorem [Gon08] and the Odd Order theorem [GAA+13], which are both challenging theorems whose manual proof would need several hundred of pages.

Chapter 2

Introduction to Abstract Interpretation

Static analyzers are tools designed to predict the behavior of programs before executing them. Static analyzers are often used for finding bugs or guaranteeing the absence of some classes of bugs in programs such as erroneous behavior. Their advantage compared to other formal methods is their ability to process large programs with little user interaction.

A largely used methodology for building static analyzers is *abstract interpretation* [CC76, CC77, CC79]. It gives a general framework for speaking about the abstractions that need to be used in order to approximate the behavior of programs in a computable manner. We give in Section 2.1 and Section 2.2 an introduction to abstract interpretation. In Section 2.3, we study the current state of the art on static analyzers based on abstract interpretation, and more specifically its formalization.

In this chapter, our presentation of abstract interpretation is idealized: the theory we describe is close to its historical formulation, but not adapted to the formalization in Coq. The understanding of this idealized setting is important for designing and formalizing static analyzers based on abstract interpretation, although we describe later in Section 3.1 the adjustments needed for Verasco.

2.1. Abstract Interpretation Concepts

The challenge of static analysis is to approximate various sets representing states and behaviors of programs. For example, we need to approximate the set of states a program can reach or the sets of values a variable can take at a given program point. These sets are potentially infinite and cannot be directly represented by the static analyzer. Therefore, the static analyzer stores approximations of these sets. These approximations are called *abstract values*, belonging to *abstract domains*, while the approximated sets are the *concrete values*, belonging to *concrete domains*.

In order to approximate sets in abstract domains, we need to reflect basic set operators, such as inclusion, union and intersection, but without directly describing sets, which are too complex. The nice theoretical structure for describing abstract domains is the structure of lattice, and the theoretical tool for defining the link between the approximated concrete domain and the approximating abstract domain is the notion of Galois connection.

2.1.1. Lattices

A *lattice* is a set equipped with a partial order relation $\leq$ and two operators[1] $\vee$ and $\wedge$, called respectively *join* and *meet*. The join operator, noted $\vee$, computes the least upper bound of two elements: that is, $A \vee B$ is larger than A and B, but smaller than any element of the lattice verifying this property:

$$A \leq A \vee B \qquad B \leq A \vee B \qquad \forall C,\ (A \leq C \text{ and } B \leq C) \Rightarrow A \vee B \leq C$$

Dually, $\wedge$ computes the greatest lower bound of two elements: that is, $A \wedge B$ is smaller than A and B, but larger than any element of the lattice verifying this property:

$$A \wedge B \leq A \qquad A \wedge B \leq B \qquad \forall C,\ (C \leq A \text{ and } C \leq B) \Rightarrow C \leq A \wedge B$$

An important example of lattice is the powerset lattice of a given set. Let Ω be a set and $\mathcal{P}(\Omega)$ the set of its subsets (i.e., the *powerset* of Ω). The powerset $\mathcal{P}(\Omega)$ naturally forms a lattice: its elements can be ordered by set inclusion, union is the join and intersection is the meet.

Another example of lattice is the parity lattice. It has 4 abstract values: $\top$, $\bot$, `Odd` and `Even`. The lattice structure is given in the following tables:

$\leq$	$\top$	`Odd`	`Even`	$\bot$
$\top$	×			
`Odd`	×	×		
`Even`	×		×	
$\bot$	×	×	×	×

$\vee$	$\top$	`Odd`	`Even`	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$
`Odd`	$\top$	`Odd`	$\top$	`Odd`
`Even`	$\top$	$\top$	`Even`	`Even`
$\bot$	$\top$	`Odd`	`Even`	$\bot$

$\wedge$	$\top$	`Odd`	`Even`	$\bot$
$\top$	$\top$	`Odd`	`Even`	$\bot$
`Odd`	`Odd`	`Odd`	$\bot$	$\bot$
`Even`	`Even`	$\bot$	`Even`	$\bot$
$\bot$	$\top$	$\bot$	$\bot$	$\bot$

One last example of lattice is the lattice of possibly unbounded integer intervals. It is given by the set I:

$$I = \{[a, b] \mid a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}\}$$

It can be ordered by the inclusion of intervals defined by the relation $\leq$ defined by:

$$\frac{a_2 \leq a_1 \qquad b_1 \leq b_2}{[a_1, b_1] \leq [a_2, b_2]}$$

Then, join and meet for intervals can easily be defined (and their properties proved):

$$[a_1, b_1] \vee [a_2, b_2] = [\min\{a_1; a_2\}, \max\{b_1; b_2\}]$$
$$[a_1, b_1] \wedge [a_2, b_2] = [\max\{a_1; a_2\}, \min\{b_1; b_2\}]$$

The largest element of the lattice of intervals is $[-\infty, +\infty]$, and the smallest element is $[+\infty, -\infty]$.

[1] It appears that these two operators have the same notation as the logical connectives *and* and *or* used in Chapter 1. It should be easy throughout this document to distinguish the lattice operators from the logical connectives using the context.

2.1.2. Galois Connections

There is a strong intuition behind integer intervals or parity: they represent sets of integers. This intuition can be formalized by defining a *concretization function*, written γ, that gives the set of integers represented by a given interval or parity information:

$$\gamma_{\texttt{Itv}}([a,b]) = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$$
$$\gamma_{\texttt{Parity}}(\bot) = \emptyset \quad \gamma_{\texttt{Parity}}(\texttt{Even}) = \{x \in \mathbb{Z} \mid x \mod 2 = 0\}$$
$$\gamma_{\texttt{Parity}}(\top) = \mathbb{Z} \quad \gamma_{\texttt{Parity}}(\texttt{Odd}) = \{x \in \mathbb{Z} \mid x \mod 2 = 1\}$$

Conversely, for any given subset X of $\mathbb{Z}$, there is a smallest interval approximating it. This interval is called the *abstraction* of X, and noted $\alpha_{\texttt{Itv}}(X)$:

$$\alpha_{\texttt{Itv}}(X) = [\inf X, \sup X]$$

where we take the convention $\min \emptyset = +\infty$ and $\max \emptyset = -\infty$. Similarly, an abstraction for the parity $\alpha_{\texttt{Parity}}(X)$ can be defined by considering whether X is empty, contains only odd elements, only even elements or both odd and even elements.

We say that the pair $(\alpha_{\texttt{Itv}}, \gamma_{\texttt{Itv}})$ forms a *Galois connection* between the lattice of intervals, the *abstract domain*, and the lattice of subsets of $\mathbb{Z}$, the *concrete domain*. Formally, a Galois connection between a concrete domain X and an abstract domain Y is a pair of functions $\alpha : X \to Y$ and $\gamma : Y \to X$ such that:

$$\forall x \in X, \forall y \in Y, \quad \alpha(x) \leq y \Leftrightarrow x \leq \gamma(y)$$

In particular, this implies that α and γ are increasing, that $\forall x,\ x \leq \gamma(\alpha(x))$ and that $\forall x,\ \alpha(\gamma(x)) \leq x$. We use the following notation to denote that the pair (α, γ) is a Galois connection:

$$X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} Y$$

Galois connections answer formally to a natural problem: it is often impossible or impractical to compute anything directly on the semantics of a program. Therefore, we use approximations that will possibly miss properties of the program (i.e., they are not complete), but never compute false properties (i.e., they are sound). The benefit is that such an approximation makes the computation easier. Galois connections help us define the link between these approximations (the abstract domain) and the actual behaviors (the concrete domain). In practice, as we see in Section 3.1, not every abstract domain can be described by a lattice and a Galois connection, but, in the good cases, the definition of a Galois connection over an abstract domain simplifies its design and gives systematic methods for the definitions of the primitives of the domain.

When applying the abstraction function, we lose some information: for example, the two different sets of integers $\{1; 3\}$ and $\{1; 2; 3\}$ have the same abstraction $[1, 3]$ in the domain of intervals. This means that not all sets of integers are represented exactly by intervals, and that some sets will be over-approximated. Conversely, there is a redundancy in our representation of intervals: indeed, several intervals have the empty set as a concretization. The abstraction function removes this redundancy by always giving the best abstraction. That is, $\alpha(X)$ is the smallest abstract value approximating X (i.e., with a concretization larger that X).

2.1.3. Transfer Functions

Approximating concrete sets is not enough for analyzing programs: indeed, it is important to be able to reflect in these approximations the computations over concrete values and states appearing in the semantics of the program. For example, in the case of sets of integer values, we want to model the sum of two such sets, so that we can approximate the addition of integers appearing in the semantics of programs.

More formally, a *concrete transfer function* F is a monotonic function from one or several concrete domains to a concrete domain. For example, the concrete transfer function for the addition of integers takes two sets of integers and returns a set of integers. It is defined as follows:

$$A + B = \{a + b \mid a \in A \text{ and } b \in B\}$$

When the concrete domains are powerset lattices, which is most often the case, it is often possible to define transfer functions as functions on *elements* rather than on subsets. For example, in the case of integer addition, it is convenient to define the transfer function on integers (which is trivial: it is simply integer addition) rather than on sets of integers.

Concrete transfer functions are approximated by abstract transfer functions, which are usually easier to compute but less precise. Formally, if $F : C_1 \times \cdots \times C_n \to C$ is a concrete transfer function from concrete domains $C_1 \cdots C_n$ to concrete domain C, then an abstract transfer function $F^\sharp : A_1 \times \cdots \times A_n \to A$ is a *sound approximation* of F if:

$$\forall X_1 \dots X_n,\ F(\gamma_1(X_1), \dots, \gamma_n(X_n)) \leq \gamma(F^\sharp(X_1, \dots, X_n))$$

That is, the abstract transfer function always returns sound approximations when given sound approximations. Because (α, γ) is a Galois connection, this definition is equivalent to:

$$\forall X_1 \dots X_n,\ \alpha(F(\gamma_1(X_1), \dots, \gamma_n(X_n))) \leq F^\sharp(X_1, \dots, X_n)$$

This means that the Galois connection naturally gives us a candidate for an abstract transfer function approximating a given transfer function. Indeed, $\alpha(F(\gamma_1(X_1), \dots, \gamma_n(X_n)))$ is a sound approximation, which is better (i.e., smaller) than all others. The definition of this sound approximation is not constructive: it uses concrete values as intermediate computation steps. Hence, this best abstract transfer function is not always easy to compute, but can serve as a guide for measuring the precision of abstract transfer functions.

Join and meet in the concrete abstract domain can be seen as concrete transfer functions, and their counterparts in the abstract domain as their abstract approximations. However, they are not necessarily optimal as transfer functions and may need reductions to improve precision.

2.1.4. Reductions

A particular case of concrete transfer function is the identity: it does not change anything in the concrete, and hence can be applied to concrete values as often as necessary. In the abstract, the identity is of course a sound approximation of the concrete identity, but it is often neither the only one nor the best one. A *reduction* operator is a sound approximation of the concrete identity: it can be applied soundly on any abstract values at any time. The abstract value returned by a reduction operator is always a sound approximation of what their argument is a sound approximation of.

Reductions are useful because some abstract domains have redundancies and hence contain several abstract values with the same concretizations. Among these abstract values,

some are better than others because they express more properties of the concrete or have a more compact representation. Reductions typically compute such better approximations.

When using lattices and Galois connections, there is always a best reduction, just like there is always a best transfer function: the best reduction of an abstract value X is $\alpha(\gamma(X))$. Hence, when this best reduction is easy to compute, we should apply it on every computed abstract value.

In our example of intervals, there are several approximations for the empty set: any $[a, b]$ with $b < a$ is such an approximation. Among them, $[+\infty, -\infty]$ is the smallest one when considering the ordering of intervals using their bounds. It is preferable to use it rather than others, because it is the only one encoding the fact that any integer in the empty set is larger than $+\infty$ and smaller than $-\infty$: this is the one carrying the most information about the empty set, and, in particular, the only one being smaller than any other, possibly non empty, interval. Therefore, the best reduction operator ρ for our intervals is:

$$\rho([a,b]) = \begin{cases} [a,b] & \text{when } a \leq b \\ [+\infty, -\infty] & \text{otherwise} \end{cases}$$

2.1.5. Products of Abstract Domains

An abstract domain is an approximation of a concrete domain. In order to get a more precise approximation, we need to combine abstract domains. For example, we can combine the information provided by an interval with parity information to get a more precise approximation of a subset of $\mathbb{Z}$.

It is easy to define the product of two lattices L_1 and L_2. It is given by the set $L_1 \times L_2$, equipped with the pointwise order relation and the pointwise meet and join operators:

$$\begin{aligned} (x_1, y_1) \leq (x_2, y_2) &\equiv x_1 \leq x_2 \text{ and } y_1 \leq y_2 \\ (x_1, y_1) \vee (x_2, y_2) &= (x_1 \vee x_2, y_1 \vee y_2) \\ (x_1, y_1) \wedge (x_2, y_2) &= (x_1 \wedge x_2, y_1 \wedge y_2) \end{aligned}$$

Similarly, if we have two Galois connections (α_1, γ_1) and (α_2, γ_2) for the same concrete domain but for two different abstract domains, we can define a Galois connection (α, γ) for the product of the two abstract domains:

$$\alpha(x) = (\alpha_1(x), \alpha_2(x)) \qquad \gamma(x, y) = \gamma(x) \wedge \gamma(y)$$

The next step is to define transfer functions for the combination of two abstract domains. The easiest method to define them is in a pointwise manner: we apply transfer functions on each component without any interaction between abstract domains. For a unary abstract transfer function $F^\sharp$ over a product of abstract domains, this gives:

$$F^\sharp(X_1, X_2) = (F_1^\sharp(X_1), F_2^\sharp(X_2))$$

If we use these transfer functions for the implementation of the combination of the two abstract domains we get the *direct product* of the two abstract domains.

Direct products are usually easy to implement, but not precise. Indeed, they provide no cooperation mechanism between the components of the product. For example, consider the direct product of the interval abstract domain and the parity abstract domain, and consider the abstract transfer functions approximating the absolute value. Applied to the interval $[-1, 1]$, it returns $[0, 1]$, and applied to the parity `Odd`, it returns `Odd`. Therefore, the abstract

transfer function for the direct product, applied to the abstract value $([-1,1], \texttt{Odd})$, returns $([0,1], \texttt{Odd})$. However, the interval $[0,1]$ is not as precise as possible, because 0, being an even number, need not be included in the interval. A cooperation between the two abstract domains would compute the better abstract value $([1,1], \texttt{Odd})$.

The problem here is that using the product of two abstract domains creates redundancies. Therefore, we should use reduction functions in order to eliminate them. Typically, such a reduction function for the product of two abstract domains will transfer as much information as possible from one domain to the other. If we use systematically the best reduction function after each computation, the resulting abstract domain is called the *reduced product* of the two abstract domains.

2.2. Analyzing `Toy` Programs

The intuition behind static analysis by abstract interpretation is that the analyzer behaves like an interpreter, but using a non-standard semantics. This so-called abstract semantics is designed so that it always terminates: the static analyzer can always compute it in finite time. The "runs" of the abstract semantics are in close connection with the standard concrete semantics of the program, via a Galois connection. In particular, the abstract semantics always represents a larger set of behaviors than the actual set of behaviors of the program. Hence, we are able to deduce properties of the executions of programs from the results of this "interpretation in the abstract". However, not all the properties of all programs can be deduced from the abstract semantics, because the abstract semantics potentially over-approximates the concrete semantics.

In this section, we describe the formalism behind such abstract semantics. In this introduction, the abstract semantics is based on intervals: that is, instead of executing the program with integer values for variables, we execute it with interval values for variables. The values of a variable in an actual run are guaranteed to be in the concretization of the corresponding interval in the abstract run.

Because the concrete runs of the program can be unbounded in execution time or even non-terminating, there cannot be a simple one-to-one correspondence between the concrete runs and the abstract runs. Therefore, we do not directly approximate the set of traces of the program, but rather the set of reached states, called the *collecting semantics*. This semantics can be characterized using a set of equations which can be themselves easily reflected in the abstract domain. We explain first this traditional method, then give another presentation, which gives very similar results, but can be implemented more easily in the case of structured languages.

2.2.1. Collecting Semantics

A static analyzer based on abstract interpretation computes an approximation of all the behaviors of the given program. The behaviors of the program are often represented by traces (i.e., sequences of states), and hence the objective of the static analyzer is to compute an over-approximation of the set of traces of the given program.

Traditionally, the first step towards this approximation is the definition of the collecting semantics of the program. In the case of an analysis for safety such as ours, the *collecting semantics* of a program is the set of states it can reach. This is a first approximation of the semantics of the program, because we cannot determine, for example, whether the program always terminates using the collecting semantics. Nonetheless, the collecting semantics can be used for determining interesting properties of the program, such as whether it can

produce an error.

For `Toy` programs, a state can be decomposed into an environment representing the values of the variables and a queue of statements to be executed, representing the current position in the code. For a given `Toy` program, the possible queues are in finite number and can be materialized in the program syntax as *control points*. Control points represent the positions in the program syntax at which the execution can be. For example, we can tag the factorial program seen earlier with control points noted by circled numbers:

$$
\begin{array}{ll}
x \Rightarrow & \textcircled{1} r := 1; \\
& \textcircled{2}\texttt{while}\ \textcircled{3}\ x\ \texttt{do} \\
& \quad \textcircled{4} r := x \times r; \\
& \quad \textcircled{5} x := x + -1 \\
& \textcircled{6}\texttt{done}; \\
& \textcircled{7}\texttt{return}\ r
\end{array}
$$

Using this set of control points, noted $\mathcal{CP}$, we can represent the set of states the program can reach (i.e., the collecting semantics) as a function $S : \mathcal{CP} \to \mathcal{P}(\mathcal{V} \to \mathbb{Z})$ from control points to sets of environments: a control point is associated to the set of environments the state can have at this point.

The collecting semantics can be characterized by a set of equations we can translate in the abstract domains. Consider our running example of factorial function, and assume that S is its collecting semantics.

- For the control point ①, we have $S(\textcircled{1}) = \{\rho^0_{x \leftarrow v}\}$, where $\rho^0_{x \leftarrow v}$ is the environment such that $\rho^0_{x \leftarrow v}(x) = v$ and $\rho^0_{x \leftarrow v}(y) = 0$ if $y \neq x$. Indeed, $\rho^0_{x \leftarrow v}$ is the only initial environment according to the operational semantics.

- For the control point ②, we have $S(\textcircled{2}) = [\![r := 1]\!](S(\textcircled{1}))$, where $[\![r := 1]\!]$ is the *concrete assignment transfer function* for $r := 1$. Formally, the concrete transfer function for an assignment $x := e$ gives the set of possible environments after the assignment, assuming that the environment before the assignment is in the given set:

$$[\![x := e]\!](X) = \{\rho' \mid \exists \rho \in X,\ (\forall y \neq x,\ \rho'(y) = \rho(y)) \text{ and } \rho \vdash e \Downarrow \rho'(x)\} \tag{2.1}$$

- Similarly, we have $S(\textcircled{5}) = [\![r := x \times r]\!](S(\textcircled{4}))$ and $S(\textcircled{6}) = [\![x := x + 1]\!](S(\textcircled{5}))$.

- The control point ③ is a join point, merging the control flow coming from ② and ⑥. Hence, the possible environments are exactly those coming from these two control points. Thus, we have $S(\textcircled{3}) = S(\textcircled{2}) \cup S(\textcircled{6})$.

- For the control point ④, we have $S(\textcircled{4}) = [\![x \neq 0]\!](S(\textcircled{3}))$, where $[\![x \neq 0]\!]$ is the *concrete assumption transfer function* for the assumption $x \neq 0$. The concrete assumption transfer function filters the environments verifying the given condition. Formally, the concrete transfer function for the assumption $e \neq 0$ is given by:

$$[\![e \neq 0]\!](X) = \{\rho \mid \rho \in X \text{ and } \exists n \neq 0,\ \rho \vdash e \Downarrow n\} \tag{2.2}$$

- Similarly, for control point ⑦, we have $S(\textcircled{7}) = [\![x = 0]\!](S(\textcircled{3}))$, where:

$$[\![e = 0]\!](X) = \{\rho \mid \rho \in X \text{ and } \rho \vdash e \Downarrow 0\} \tag{2.3}$$

We can merge all these equations into one equation over S. We write $S = F(S)$, where F is an increasing function on the lattice $\mathcal{CP} \to \mathcal{P}(\mathcal{V} \to \mathbb{Z})$, defined using the previously defined equations. It can be proven that the collecting semantics is the smallest solution of the fixpoint equation $F(S) = S$.

As a side remark, the collecting semantics can be seen as an abstraction of the semantics of the program [Cou02]. Indeed, if we equip both sets of trace and sets of states with the powerset lattice structure, we can see the transformation from the set of traces to the collecting semantics as a Galois connection. This Galois connection is often called the *state abstraction*.

2.2.2. Approximating the Collecting Semantics

Many properties of the concrete behaviors of a program can be stated on its collecting semantics, an element of $\mathcal{CP} \to \mathcal{P}(\mathcal{V} \to \mathbb{Z})$ giving the reachable states of the program. However, the sets of environments (i.e., elements of $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$) present in the collecting semantics are still potentially infinite and hence complex objects. Therefore, an additional layer of approximation is needed in order to write a static analyzer that uses only objects on which it knows how to compute.

This new abstraction approximates sets of environments using *abstract environments*. In many cases[2], an abstract domain $\mathcal{E}$ of abstract environments is equipped with a lattice structure and a Galois connection between abstract environments and the powerset lattice of concrete environments:

$$\mathcal{P}(\mathcal{V} \to \mathbb{Z}) \underset{\alpha_{\mathsf{Env}}}{\overset{\gamma_{\mathsf{Env}}}{\leftrightarrows}} \mathcal{E}$$

Then, to approximate the collecting semantics, we use a map from control points to abstract environments: each abstract environment approximates the set of concrete environments the program can have when reaching a given control point. This is an example of an *abstract domain functor*, building an abstract domain for the collecting semantics based on an abstract domain for environments. That is, using the Galois connection $(\alpha_{\mathsf{Env}}, \gamma_{\mathsf{Env}})$ for environments, we can build a Galois connection $(\alpha_{\mathsf{Sem}}, \gamma_{\mathsf{Sem}})$ for the collecting semantics:

$$\begin{array}{rcl} \mathcal{CP} \to \mathcal{P}(\mathcal{V} \to \mathbb{Z}) & \underset{\alpha_{\mathsf{Sem}}}{\overset{\gamma_{\mathsf{Sem}}}{\leftrightarrows}} & \mathcal{CP} \to \mathcal{E} \\ \alpha_{\mathsf{Sem}}(S) & = & \mathsf{cp} \mapsto \alpha_E(S(\mathsf{cp})) \\ \gamma_{\mathsf{Sem}}(S^\sharp) & = & \mathsf{cp} \mapsto \gamma_E(S^\sharp(\mathsf{cp})) \end{array}$$

2.2.3. Approximating Sets of Environments: Non-Relational Abstractions

To build an abstract domain for environments of `Toy`, we need a *numerical abstract domain*, handling numerical properties of environments. In contrast, in Verasco, as we explain in Section 3.2, we need a more complex hierarchy of abstractions in order to take into account the more complex structure of C#minor memory.

A simple way of building a numerical abstract domain is by using a *non-relational abstraction*, which is another abstract domain functor. This functor uses an abstract domain concretizing to sets of integers, such as intervals or parity, and builds an abstract domain for numerical environments. To this end, this abstract domain associates to each variable

[2] As we explain in Chapter 3, this is not always true, and we often weaken this formalism in actual abstract interpreters such as Verasco.

an abstract value approximating the values the variable has in the approximated environments. More formally, assume we have a numerical abstract domain $\mathcal{Z}$ in Galois connection with $\mathcal{P}(\mathbb{Z})$:

$$\mathcal{P}(\mathbb{Z}) \underset{\alpha_{\mathbb{Z}}}{\overset{\gamma_{\mathbb{Z}}}{\leftrightarrows}} \mathcal{Z}$$

From this Galois connection, we can build a Galois connection between the lattice $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$ of sets of concrete environments and the lattice $\mathcal{E} = \mathcal{V} \to \mathcal{Z}$ of abstract environments:

$$\begin{array}{rcl} \mathcal{P}(\mathcal{V} \to \mathbb{Z}) & \underset{\alpha_{\mathsf{Env}}}{\overset{\gamma_{\mathsf{Env}}}{\leftrightarrows}} & \mathcal{V} \to \mathcal{Z} \\ \alpha_{\mathsf{Env}}(X) & = & v \mapsto \alpha_{\mathbb{Z}}(\{\rho(v) \mid \rho \in X\}) \\ \gamma_{\mathsf{Env}}(X^\sharp) & = & \{\rho \mid \forall v,\ \rho(v) \in \gamma_{\mathbb{Z}}(X^\sharp(v))\} \end{array}$$

This non-relational abstraction is not exact: it can miss potentially interesting properties of numerical environments. That is, even if the abstract domain $\mathcal{Z}$ on integers were exact (e.g., $\mathcal{Z} = \mathcal{P}(\mathbb{Z})$, with the identity Galois connection), the derived non-relational abstract domain for numerical environments would not be able to represent every property over numerical environments. In particular, it is not possible to represent relations between the values of different variables: for instance, it is impossible, with a non-relational abstract domain, to establish the fact that $x - y \leq 2$ for two program variables x and y. Relational abstract domains exist: a popular example is the abstract domain of octagons, whose implementation in Verasco is described in Chapter 8.

In this introductory chapter, we limit our choice of numerical abstract domain to the non-relational abstract domain of intervals. Hence, to summarize, our abstract domain for approximating the collecting semantics of the program uses elements of $\mathcal{CP} \to \mathcal{V} \to I$, where $\mathcal{CP}$, the set of control points, and $\mathcal{V}$, the set of program variables are finite. This lattice is simple enough that we can represent its elements in the static analyzer (using, e.g., finite maps) and implement operations using these abstract values. These operations include comparison, join, meet and transfer functions.

2.2.4. Abstract Semantics

The series of abstract domains we have defined is a tool to compute an approximation of the semantics of programs. We now have to compute this approximation by reflecting in the abstract domains the fixpoint equation $F(S) = S$ characterizing the collecting semantics. Assume we have an abstract transfer function $F^\sharp$, called *abstract semantics* approximating the concrete transfer function F:

$$\forall X, F(\gamma_{\mathsf{Sem}}(X)) \leq \gamma_{\mathsf{Sem}}(F^\sharp(X)) \tag{2.4}$$

The property we use to get an approximation of the collecting semantics S is that post-fixpoints of $F^\sharp$ (i.e., abstract values $S^\sharp$ such that $F^\sharp(S^\sharp) \leq S^\sharp$) are approximations of the least fixpoint of F (i.e., the collecting semantics). Indeed, further assume that we have such a post-fixpoint $S^\sharp$ of $F^\sharp$. Then, $F(\gamma_{\mathsf{Sem}}(S^\sharp)) \subseteq \gamma_{\mathsf{Sem}}(F^\sharp(S^\sharp)) \subseteq \gamma_{\mathsf{Sem}}(S^\sharp)$, thus $\gamma_{\mathsf{Sem}}(S^\sharp)$ is a post-fixpoint of F. But a general property of least fixpoints of increasing functions in complete lattices is that a post-fixpoint is always larger than the least fixpoint (this is a consequence of the proof of the Knaster–Tarski theorem). Thus, we have $S \subseteq \gamma_{\mathsf{Sem}}(S^\sharp)$; so $S^\sharp$ is an approximation of the collecting semantics.

Therefore, it remains two problems: first, we need to find an abstract transfer function $F^\sharp$ approximating F, the abstract semantics and second, we need to find an algorithm to

compute a post-fixpoint of $F^\sharp$.

The abstract semantics $F^\sharp$ is built by combining approximations of the initial environment and of the transfer functions for assignment, assumption and union. Concrete union is soundly approximated by the abstract join. Approximating the initial environment is just as easy: we know the interval $[0, 0]$ for every variable, except the program parameter. Moreover, an interval for this paramter is typically given as a parameter of the static analyzer. We describe here the approximation of the assignment transfer function, but omit the approximation of assumption, which is more technical. We refer the reader to Section 6.1.4 for the description of such an algorithm in Verasco.

In order to approximate soundly the concrete transfer function for assignments, we need to get an interval for the possible values of an expression when the environment is in the concretization of a given abstract environment: that is, given an interval for every variable (i.e., an abstract environment), what is an interval for the given expression? Again, we can answer this question with a pair of a concrete and an abstract transfer function. For a given expression e, the concrete transfer function $[\![e]\!]$ associates to a set of concrete environments the set of values e can have in one such environment:

$$[\![e]\!](X) = \{v \mid \exists \rho \in X,\ \rho \vdash e \Downarrow v\} \tag{2.5}$$

An abstract transfer function $[\![e]\!]^\sharp$ approximating this concrete transfer function returns an interval for the expression when given an abstract environment. Recall that abstract environments associate an interval to each variable: they are elements of $\mathcal{V} \to I$. A possible such abstract transfer function is defined recursively on the structure of e:

$$\begin{aligned}
[\![n]\!]^\sharp(E) &= [n, n] \quad \text{when } n \in \mathbb{Z} \\
[\![x]\!]^\sharp(E) &= E(x) \quad \text{when } x \in \mathcal{V} \\
[\![-e]\!]^\sharp(E) &= -^\sharp([\![e]\!]^\sharp(E)) \\
[\![e_1 + e_2]\!]^\sharp(E) &= ([\![e_1]\!]^\sharp(E)) +^\sharp ([\![e_1]\!]^\sharp(E)) \\
[\![e_1 \times e_2]\!]^\sharp(E) &= ([\![e_1]\!]^\sharp(E)) \times^\sharp ([\![e_1]\!]^\sharp(E)) \\
[\![e_1 \div e_2]\!]^\sharp(E) &= ([\![e_1]\!]^\sharp(E)) \div^\sharp ([\![e_1]\!]^\sharp(E))
\end{aligned}$$

where the negation, addition, multiplication and division of intervals are themselves abstract transfer functions approximating their concrete counterparts. For example, the negation and addition of intervals can be defined by:

$$-^\sharp[a, b] = [-b, -a] \qquad\qquad [a_1, b_1] +^\sharp [a_2, b_2] = [a_1 + a_2, b_1 + b_2]$$

We can prove that these two definitions are optimal, provided the given intervals are reduced: they correspond exactly to the best transfer function provided by the Galois connection for intervals. We do not give the definitions of the abstract multiplication and division of intervals: they are technical but do not present any theoretical difficulty.

It should be noted that this transfer function for expressions is not optimal. That is, in some situations, the interval it will return for an expression is not the best possible one. For example, if we have the interval $[-1, 1]$ for the variable x, then it will compute the interval $[-1, 1]$ for the expression $x \times x$, while $[0, 1]$ is a smaller, but still sound interval.

Using $[\![e]\!]^\sharp$, we can define $[\![x := e]\!]^\sharp$, the abstract transfer function for assignments, which we need for defining the abstract semantics $F^\sharp$. This transfer function computes an abstract environment approximating the set of possible concrete environments after an assignment,

given an approximation of the set of concrete environments before the assignment:

$$[\![x := e]\!]^\sharp(E) \;=\; y \mapsto \begin{cases} [\![e]\!]^\sharp(E) & \text{if } y = x \\ E(y) & \text{otherwise} \end{cases}$$

For all these abstract transfer functions, we can prove soundness theorems such as:

$$[\![e]\!](\gamma_{\texttt{Env}}(E)) \subseteq \gamma_{\texttt{Itv}}([\![e]\!]^\sharp(E)) \qquad [\![x := e]\!](\gamma_{\texttt{Env}}(E)) \subseteq \gamma_{\texttt{Env}}([\![x := e]\!]^\sharp(E))$$

Then, we can define the abstract semantics $F^\sharp$: it is defined by following the fixpoint equation characterizing the collecting semantics. We prove the soundness theorem of the abstract semantics:

$$F(\gamma_{\texttt{Sem}}(X)) \subseteq \gamma_{\texttt{Sem}}(F^\sharp(X))$$

which lets us deduce, as we have seen, that a post-fixpoint of the abstract semantics $F\sharp$ is necessarily a sound approximation of the collecting semantics.

We do not detail the computation of such a post-fixpoint, which requires some engineering to be done efficiently [Bou93]. Instead, we will explain in Section 2.2.6 the fixpoint computation in another setting, which has the other advantage of being closer to the methods we actually use in Verasco. The important thing to understand here is that the computation of such a post-fixpoint can be done using always terminating algorithms, and hence the static analyzer can actually do this computation.

To illustrate these definitions, we give here a possible post-fixpoint of the abstract semantics for the factorial program that could have been computed by a static analyzer, when given the interval $[2, 10]$ for the possible values of the parameter x:

$x \Rightarrow$ ①$r := 1$;
②`while` ③ x `do`
④$r := x \times r$;
⑤$x := x + -1$
⑥`done`;
⑦`return` r

$$\begin{aligned}
① &\mapsto \{x \mapsto [2,10]; r \mapsto [0,0]\} \\
② &\mapsto \{x \mapsto [2,10]; r \mapsto [1,1]\} \\
③ &\mapsto \{x \mapsto [0,10]; r \mapsto [1,+\infty]\} \\
④ &\mapsto \{x \mapsto [1,10]; r \mapsto [1,+\infty]\} \\
⑤ &\mapsto \{x \mapsto [1,10]; r \mapsto [1,+\infty]\} \\
⑥ &\mapsto \{x \mapsto [0,9]; r \mapsto [1,+\infty]\} \\
⑦ &\mapsto \{x \mapsto [0,0]; r \mapsto [1,+\infty]\}
\end{aligned}$$

In particular, we can see that this post-fixpoint is not able to infer any better result interval than $[1, +\infty]$: in fact, a relational abstract domain is needed to compute a better one.

2.2.5. Deducing Properties of the Program

The role of a static analyzer is to prove properties of a program without executing it: this can be done using the abstract semantics. For example, if using our static analyzer for `Toy`, a simple property that we can compute is an interval for the returned value, by applying the abstract environment of the final control point to the abstract transfer function of the result expression.

In the case of Verasco, we are mostly interested in the absence of erroneous behavior. The mode of operation of Verasco is to emit an *alarm* whenever it is not able to prove the absence of an error at some place in the code. Then, the guarantee provided by its main soundness theorem states that if it does not return any alarm for a given program, then this program is free of erroneous behavior. This means there are two kinds of alarms Verasco can emit:

true alarms, which correspond to real bugs in the programs, and *false alarm*, which are consequences of the incompleteness of the analysis, and Verasco cannot distinguish between these two kinds of alarms. In contrast, as a consequence of the soundness of the analysis, there is no "false negatives", where the analyzer would return no alarm on an incorrect program.

In the `Toy` language, erroneous behaviors are divisions by zero. Therefore, error detection for `Toy` programs consists in emitting an alarm for each possible division by zero: using our approximation for the collecting semantics, we compute an interval for each divisor appearing in the program. If the interval for a divisor contains zero, then the static analyzer is not able to prove that this division is valid, and therefore emits an alarm.

2.2.6. Structural Abstract Interpretation

This approach for designing static analyzers by approximating the collecting semantics works, but it has several problems. First, control points are difficult to define and to uniquely identify in languages whose semantics is not defined in terms of control points. It becomes even more complicated when dealing with function calls if we want to inline them on-the-fly. Second, the large amount of required control points makes the data structures needed to store all the abstract environments use a lot of memory. Third, solving the post-fixpoint equation of the abstract semantics is not easy and necessitates complex algorithms [Bou93].

The reason for these problems is that, by using this notion of control point, we essentially forgot the syntactical structure of the program: we would get the same abstract semantics if the program were transformed to a control flow graph before the analysis.

In this section, we describe how we can define another kind of abstract interpretation on `Toy` programs, which uses the syntactic structure of the program and therefore avoids the problems we mentioned. The abstract interpreter of Verasco follows the same ideas. When we use the collecting semantics, we approximate the small-step operational semantics of `Toy`. Instead, in this section, we will approximate the axiomatic semantics of `Toy`, defined in Section 1.1.2.

The axiomatic semantics uses logical predicates for pre- and post-conditions. These logical predicates can be seen, equivalently, as sets of environments for which they are valid. Therefore, we reuse the abstract domain for approximating environments defined above to approximate pre- and post-conditions. More precisely, for any statement s, we define an abstract transfer function $[\![s]\!]^\sharp_{\texttt{Stmt}}$, taking as parameter an abstract environment (an approximation of the pre-condition) and returning either an abstract environment (an approximation of the post-condition) or an error, noted `ERR`, when the absence of error in s could not be established. The soundness of this abstract transfer function is stated relatively to the axiomatic semantics:

$$\frac{[\![s]\!]^\sharp_{\texttt{Stmt}}(E_{\texttt{pre}}) = E_{\texttt{post}} \neq \texttt{ERR}}{\{\rho \in \gamma_{\texttt{Env}}(E_{\texttt{pre}})\} \;\; s \;\; \{\rho \in \gamma_{\texttt{Env}}(E_{\texttt{post}})\}}$$

We do a slight abuse in notations by letting environments ρ appear in pre- and post-conditions. Finally, we can apply the abstract transfer function of the body of the program to the initial abstract environment: if the result is not `ERR`, then it is guaranteed that the program does not fail, and the returned abstract environment approximates the final environment of the program.

Apart from the case of loops, the definition of $[\![s]\!]^\sharp_{\texttt{Stmt}}$ is straightforward: it literally corresponds to programming an interpreter for `Toy`, but using abstract environments instead

of concrete ones:

$$[\![x := e]\!]^\sharp_{\texttt{Stmt}}(E) = \begin{cases} [\![x := e]\!]^\sharp(E) & \text{if, for any divisor } d \text{ appearing in } e,\ 0 \notin \gamma_{\texttt{Itv}}([\![d]\!]^\sharp(E)) \\ \texttt{ERR} & \text{otherwise} \end{cases}$$

$$[\![s_1; s_2]\!]^\sharp_{\texttt{Stmt}}(E) = \begin{cases} \texttt{ERR} & \text{if } [\![s_1]\!]^\sharp(E) = \texttt{ERR} \\ [\![s_2]\!]^\sharp([\![s_1]\!]^\sharp(E)) & \text{otherwise} \end{cases}$$

The proofs of soundness of these definitions follow naturally from the Hoare logic.

Analyzing loops

The definition of $[\![\texttt{while } e \texttt{ do } s \texttt{ done}]\!]^\sharp_{\texttt{Stmt}}$ is more complicated: indeed, in order to use the rule for loops (1.9) of the Hoare logic, we need to weaken the pre-condition in order to get an invariant valid as a post-condition of the loop body. Another way of seeing the problem is that we cannot "execute" the loop in the static analyzer, because it might not terminate. Instead, we need to infer an abstract value $I^\sharp$ whose concretization can serve as an invariant. Reflecting (1.9) in terms of $[\![\cdot]\!]^\sharp_{\texttt{Stmt}}$, we get the following conditions that $I^\sharp$ should verify in order to compute $[\![\texttt{while } e \texttt{ do } s \texttt{ done}]\!]^\sharp_{\texttt{Stmt}}(E)$:

$$E \leq I^\sharp \tag{2.6}$$

$$[\![s]\!]^\sharp_{\texttt{Stmt}}([\![e \neq 0]\!]^\sharp(I^\sharp)) \leq I^\sharp \tag{2.7}$$

$$\text{For any divisor } d \text{ appearing in } e,\ 0 \notin \gamma_{\texttt{Itv}}([\![d]\!]^\sharp(I^\sharp)) \tag{2.8}$$

$$[\![s]\!]^\sharp_{\texttt{Stmt}}([\![e \neq 0]\!]^\sharp(I^\sharp)) \neq \texttt{ERR} \tag{2.9}$$

Then, assuming that we have such an abstract environment $I^\sharp$, we can use:

$$[\![\texttt{while } e \texttt{ do } s \texttt{ done}]\!]^\sharp_{\texttt{Stmt}}(E) = [\![e = 0]\!]^\sharp(I^\sharp) \tag{2.10}$$

Therefore, the algorithm for computing $[\![\texttt{while } e \texttt{ do } s \texttt{ done}]\!]^\sharp_{\texttt{Stmt}}(E)$ proceeds as follows: first, we search for a value for $I^\sharp$ verifying the two inequations (2.6) and (2.7). This abstract environment should be as small as possible, because it directly impacts the precision of the analysis. Then, we check whether (2.8) and (2.9) hold for this solution. If so, we return the abstract environment described by (2.10); otherwise we return ERR.

It remains to compute an abstract environment $I^\sharp$ satisfying (2.6) and (2.7). These conditions state that $I^\sharp$ is a post-fixpoint of the function $X \mapsto E \vee [\![s]\!]^\sharp_{\texttt{Stmt}}([\![e = 0]\!]^\sharp(X))$. The usual way of finding a post-fixpoint of increasing functions in finite lattices is to apply the Kleene's fixed-point theorem: we iterate the function starting from E until reaching a fixpoint. This method cannot be directly applied to our case, because of two problems: first, the iteration may not terminate because the abstract domain could have infinite ascending chains and second, the iterated function is not necessarily increasing.

Abstract interpretation provides a general solution to force the iteration to reach a post-fixpoint in finite time: at each iteration, if a constraint stored in the abstract domain is unstable (i.e., weakens compared to the previous iteration), then it should be *extrapolated* (i.e., either removed or transformed into a weaker constraint). Quickly, no unstable constraint remains, and we have a post-fixpoint. More formally, we use a new operator defined on abstract environments called *widening* and noted $\triangledown$. It should verify the following three

properties:

$$\forall\ A\ B,\quad A \leq A \triangledown B \tag{2.11}$$

$$\forall\ A\ B,\quad B \leq A \triangledown B \tag{2.12}$$

$$\text{For all sequences } (A_i), \text{ the sequence } (A_i^{\triangledown}) \text{ defined by } A_0^{\triangledown} = A_0 \text{ and } A_{i+1}^{\triangledown} = A_i^{\triangledown} \triangledown A_{i+1} \text{ is ultimately stationary.} \tag{2.13}$$

Then, we compute, in the static analyzer, the following sequence:

$$\begin{cases} X_0 & = E \\ X_{n+1} & = X_n \triangledown [\![s]\!]^\sharp_{\texttt{Stmt}}([\![e = 0]\!]^\sharp(X_n)) \end{cases} \tag{2.14}$$

until one iteration returns `ERR`, in which case the post-fixpoint computation fails and we have no other choice than returning `ERR` for the loop, or until we reach a post-fixpoint. The properties of the widening ensure that such a post-fixpoint is necessarily reached after a finite number of iterations, and that the post-fixpoint in question is larger than E.

Apart from the three properties above, the design of widening operators is a heuristic task. For the abstract environments we use for `Toy`, a widening operator can be defined by applying pointwise a widening operator for intervals. The typical widening operator for intervals [CC76] is obtained by the following definition:

$$[a_1, b_1] \triangledown [a_2, b_2] = \left[\begin{cases} a_1 & \text{if } a_1 \leq a_2 \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b_1 & \text{if } b_2 \leq b_1 \\ +\infty & \text{otherwise} \end{cases} \right] \tag{2.15}$$

Using a widening operators makes the post-fixpoint iteration terminate, but it can lead to a dramatic loss in precision, because unstable constraints are extrapolated quickly. For example, if we use this widening for intervals for the analysis of our factorial program, we infer as invariant the interval $[-\infty, 10]$ for x instead of $[0, 10]$, and, therefore, any information on the result of the computation is lost: the final interval for r is $[-\infty, +\infty]$.

In order to mitigate this loss in precision, we can use another widening operator, or use additional decreasing iterations after the fixpoint is computed. Both techniques will possibly lead to a slower iteration process.

2.3. State of the Art in Static Analysis by Abstract Interpretation

Static analysis by abstract interpretation is an active research area. Progresses were first achieved in abstract interpretation theory. While this fundamental research continued, the development of such static analysis tools based on abstract interpretation started, and several of them are now regularly used by industry for improving the safety of software. The research needed toward the formal verification of such tools started during the 2000s and this thesis is the continuation of this work. In this section, we present prior work that we consider representative of the state of the art in the domain of static analysis by abstract interpretation. We first consider mature unverified tools and then we present several attempts toward their formal verification.

2.3.1. Static Analyzers Based on Abstract Interpretation

Static analyzers based on abstract interpretation that aim at finding bugs in software can be categorized into two categories. First, unsound tools such as Coverity [BBC+10] or .Net CodeContracts static checker [FL10] find bugs in software but do not *guarantee* the absence of bug in a given program. Indeed, when these tools face features of the input language that are particularly difficult to analyze (such as aliasing or reentrancy), they do optimistic assumptions on the input programs. Therefore, they can miss behaviors, but are capable of analyzing a large variety of programs while still generating a reasonable amount of false alarms. Their unsoundness does not mean that they are not useful: a program passing all those tests without generating an alarm is generally of great quality and less likely to contain bugs. Second, some tools, such as Astrée [BCC+03, BCC+02], Fluctuat [DGP+09] or Frama-C Value [Fv] claim to be sound, which means they should never miss incorrect programs. Sound tools will generally either need more user interaction or produce more false alarms, but when they do not return any alarm, they provide a formal guarantee of the absence of some classes of bugs in the given program. Verasco is definitely in the second category: not only we claim Verasco provides such guarantees, but we give a formal proof of this property.

Astrée. Astrée [BCC+03, BCC+02] is a sound static analyzer based on abstract interpretation. It is targeted at the verification of large critical real-time software, written in C. Therefore, it has limited support for features such as recursion or dynamic memory allocation which are not used in this particular type of programs. On the other hand, it contains many numerical abstract domains able to prove the complex numerical invariants appearing in these codes. Astrée has been successfully used to verify the absence of runtime errors in the fly-by-wire systems embedded in the Airbus A340 and A380 airplanes. The design of Verasco is largely inspired from that of Astrée.

Frama-C Value. Frama-C is a platform for specifying and verifying C code. Several Frama-C plugins exist, enabling the user to use several different technologies to prove properties of her program. One of these plugins, *Value* [Fv], is a sound static analyzer based on abstract interpretation. One of its strength is the fact that it can be combined with other verification techniques through Frama-C in order to circumvent its potential imprecision.

Fluctuat. Fluctuat [DGP+09] is a static analyzer based on abstract interpretation. Its main objective is the study of the influence of rounding errors of floating-point computations. It is able to compute bounds of the difference between the actual result of a program and its theoretical result when executed using reals instead of floating-point numbers. Fluctuat also tracks the origin of rounding errors (i.e., the contribution from each instruction). It makes it easy for the programmer to improve the overall precision by targetting the largest sources of imprecision first.

2.3.2. Mechanization of Abstract Interpretation

We claim that Verasco is currently the most sophisticated and realistic attempt at verifying a static analyzer based on abstract interpretation. Nevertheless, this is not a new idea, and several prior attempts can be compared to ours.

Monniaux [Mon98] made one of the first attempts at formally verifying abstract interpretation, as part of his master's thesis. He formally verified many results about lattice theory, fixpoint computation and Galois connections. He then implemented two toy static

analyzers: the first analyzed regular expressions and the second targeted a simple imperative language. The first one is fully formally verified and extractable to OCaml, but the lack of time prevented him to finish the formal verification of the second one.

Pichardie and his colleagues [CJPR04, CJPS05, Pic05, Pic08, BCJP09, CP10] implemented and verified several static analyzers based on abstract interpretation. Their work include the formal verification of a static analyzer for a simple imperative language and for the Java bytecode. Their analyzers are built in a composable way, by using the module system of Coq. Each concept is described as a module interface, and each component as a module or functor; the static analyzers are constructed by composing these modules. Similarly to the non-mechanized formalization of the Astrée static analyzer, Pichardie explains that the formalism of Galois connections is unnecessarily complicated to formalize in Coq, and prefers a formalism where only a concretization function γ is used, like we do in Verasco. A large part of this work is dedicated to the termination proof of fixpoint computations. We avoid these proofs, that are not necessary for the final soundness theorem: we prefer using the fuel trick that lets us define in Coq non-terminating functions. Finally, the main difference between Verasco and this earlier work is the complexity of the analyzer: Verasco targets a more complex language, C#minor; it uses a more complex combination of numerical abstract domains, including relational abstractions, while the analyzer of Pichardie has only simple non-relational numerical abstract domains such as signs, intervals and congruences; and Verasco is able to infer complex memory properties, which is not possible with the analyzers of Pichardie.

Bertot [Ber09] formalized a static analyzer based on abstract interpretation for a simple imperative language. This analyzer can be instantiated by several non-relational abstract domains, such as intervals or parity. It uses a weakest pre-condition calculus as the description of the semantics of this language. This approach is comparable to our use of a Hoare logic for proving the soundness of our abstract iterator. The output of the analyzer is a version of the input program annotated with logical assertions: we find this idea interesting, and Verasco may, in the future, output such an annotated version of its input. This would enable us to leverage the information acquired about the program during the analysis for other purposes (such as optimizations or uses in other specialized analyzers).

Several authors prototyped static analyzers for educational purposes. Leroy [Ler12] describes a static analyzer based on abstract interpretation for a simple imperative language. His development includes standard abstract interpretation mechanisms, such as backward analysis of expressions and fixpoint iteration using widening and decreasing iterations. Similarly to Verasco, the emphasis is on proving the soundness of the analysis, but no result is guaranteed about the optimality of abstract transfer functions. As a result, no definition of the abstraction function α is given. Another similarity with Verasco is the fact that all the concrete domains are fixed to be powerset lattices, simplifying the definition and use in formal proofs of the concretization function γ.

Nipkow [Nip12, NK14] describes a static analyzer based on abstract interpretation on a simple language of annotated commands. These annotations let him conveniently define the operational semantics, the collecting semantics and the abstract interpreter with the same tools. As in the work by Bertot [Ber09], the output of the analyzer is an annotated version of the input program. Nipkow describes his framework independently of the non-parametric numerical abstract domain used and then instantiated it by sign analysis, constant propagation, parity and intervals. His formalization includes fixpoint iteration with widening and narrowing, together with their technical termination proofs.

Blazy, Laporte, Maroneze and Pichardie [BLMP13] designed, implemented and formally verified a static analyzer based on abstract interpretation. Their implementation is the ancestor of Verasco. Therefore, it shares many of its design decisions. A major design

difference between this analyzer and Verasco is the fact that it targets CFG, an intermediate language specially crafted for this purpose, instead of C#minor. This intermediate language uses control flow graphs instead of structured syntax. Hence, they needed an external unverified fixpoint iterator (based on Bourdoncle's algorithm [Bou93]) in order to solve the complex system of inequations arising from the abstract interpretation of such languages. Their static analyzer stores in memory all the intermediate abstract states provided by the external solver and checks that this is indeed a valid solution to the system of inequations. Other improvements in Verasco include better abstractions of the structure of the memory, and a more sophisticated combination of numerical abstract domain, including relational abstract domains and a generic functor for taking into account the boundedness of integers.

Cho, Kang, Choi and Yi [CKCY13] implemented a formally verified validator, called SparrowBerry, for the results of the analysis computed by Sparse Sparrow, an unverified static analyzer for C. The static analysis is performed ahead of time on a CFG representation of the input program by an unverified static analyzer. Then, their formally verified tool checks the validity of the computed abstract values with respect to the semantics of the program. This approach presents an important problem compared to Verasco. Indeed, Verasco contains a large variety of abstract domains that would need to be implemented twice: once in the unverified analyzer and once in the validator. If, as in SparrowBerry, the abstract domain hierarchy is simple, this is feasible, but for complex hierarchies such as the one used in Verasco, we think it would be extremely difficult to keep equivalent the two implementations. Moreover, the memory consumption of the data structures needed to transfer the information from the analyzer to the validator can be large, and, for structured languages such as Verasco, it is unclear how this information could be represented at all.

Bodin, Jensen and Schmitt [BJS15] show how a formally verified static analyzer can be automatically derived from a pretty-big step semantics of the input language. This approach could have simplified the work needed to define the abstract iterator of Verasco (see Section 4.3), but a pretty-big step semantics for C#minor would still be needed to be written and formally verified, which is far from easy.

Darais and Van Horn [DVH15] give a new approach to formalize Galois connections in verified static analyzers, that would avoid the use of classical axioms that seem unavoidable when defining abstraction functions α[3]. In Verasco, the use of classical axioms is not a real obstacle. The reason why we are not using full Galois connections is twofold: first, some abstract domains, such as polyhedra or symbolic abstract domains, do not have abstraction functions α, and second, the use of a fully well-behaved Galois connection setting would assume the existence of a well-defined lattice structure on abstract values, which is far from trivial for complex combinations of abstract domains. In Verasco, such properties, which are important for guaranteeing some sort of optimality of the abstract transfer functions, would be a waste of formalization effort, because the emphasis is on the soundness of the analyzer.

[3]Indeed, the definition of α in a proof assistant such as Coq proses problems, because this function is most often non-constructive.

II

Formally Verified Abstract Interpretation

Chapter 3

A Modular Architecture

Before digging into the details of Verasco, we need to have a glimpse of the framework we used for formalizing abstract interpretation together with a view of its architecture. We explain that some adaptations to the general abstract interpretation theory are needed for implementing realistic static analyzers in general, and for our Coq formalization in particular. We present the general framework we use to represent abstract domains using Coq type classes. Then, we describe the architecture of Verasco: we use a very modular architecture, that lets us design each part independently from the others. We try to follow a strict methodology, where each abstract domain is an implementation of an interface that aims at being as simple as possible.

3.1. Abstract Interpretation in Practice

When it was first designed, the abstract interpretation theory was mainly a mathematical framework without a concrete implementation [CC76, CC77, CC79]. Since then, many static analyzers have been implemented following the approach of abstract interpretation. While abstract interpretation is clearly a basis for these works, some adaptations need to be made for designing a realistic static analyzer. A high-level survey of some possible frameworks can be found in [CC92]. Astrée [BCC+03, CCF+06] also uses a relaxation, and early works on formalizing abstract interpretation by Pichardie [Pic05] already described many possible relaxations. Some of those are needed due to the lack of good properties of abstract domains (such as polyhedron, which do not have best abstractions). Some others are guided by our design strategy: proving only what is needed for the main soundness theorem. In this section, we describe our choices, together with justifications.

3.1.1. Galois Connections and Lattice Structure

Typically, in abstract interpretation theory, concrete and abstract domains are equipped with a lattice structure with an order relation $\leq$. A Galois connection is used to describe the link between an abstract domain and a corresponding concrete domain. It consists in two monotonic functions: α from the concrete domain to the abstract domain, and γ from the abstract domain to the concrete domain. It is supposed to verify:

$$\alpha(x) \leq y \iff x \leq \gamma(y) \tag{3.1}$$

The formalism of Galois connections is not practical for some realistic abstract domains: some of them, such as polyhedra or many symbolic abstract domains, do not have best approximations for some concrete sets. Thus, we will not be able to properly define the abstraction function α. The complete lattice structure suffers from similar drawbacks. Indeed, many abstract domains do not have least upper bounds and greatest upper bounds for arbitrary families[1]. Moreover, from a software engineering perspective, even for abstract domains featuring these nice properties (such as intervals), it may be desirable not to choose best abstractions for some abstract transfer functions when the best approximation is too complicated to compute.

The lattice and Galois connection formalisms are definitely useful to design precise abstract domains. As an example, one can prove that some abstract operation is the most precise possible for some abstract domain using the α function. However, they are not needed for soundness: in Verasco, we decided to take the shortest path to a correct implementation with its correctness theorem. That is, for every abstract domain, we only provide a concretization function γ, a comparison operator $\sqsubseteq$ and some abstract operations.

The concretization function γ

The γ function takes an element of the abstract domain and returns its corresponding element in the concrete domain. For all the abstract domains we implemented, the corresponding concrete domain is a powerset lattice (i.e., the concrete domain is the set of the subsets of a larger set, ordered by inclusion): we added this constraint to the type of concretization functions. They are implemented using a Coq type class:

```
Class gamma_op (A B: Type) : Type :=
  γ : A -> 𝒫 B.
```

where `𝒫 B` is a notation for `B -> Prop`, the type of sets of elements of type `B`. The intuitive meaning of `γ` is that for an abstract value `x`, the Coq term `γ x` corresponds to the set of concrete values represented by `x`.

The use of type classes in Verasco is very frequent: this helps us keep the code easy to read by letting Coq inferring a large amount of implicit information from the context. For example, if `x` is an integer and `i` an interval, we simply write `γ i x`, or equivalently using a dedicated notation, `x ∈ γ i` for stating that `x` belongs to `i`. Coq will automatically choose the right instance of the type class to give a meaning to this statement. This is especially useful when defining abstract domains generically.

As an example, for some abstract domain `A`, we define `A+⊥` as `A` augmented with a bottom element with empty concretization:

```
Inductive botlift (A:Type) : Type :=
| Bot
| NotBot (x:A).
Notation "t +⊥" := (botlift t) (at level 39).

Instance gamma_bot A B (G: gamma_op A B) : gamma_op (A+⊥) B :=
  (fun x =>
     match x with
     | Bot => fun _ => False
     | NotBot x => γ x
     end).
```

[1] This is the case for many linear abstract domains: polyhedra [CH78] have least upper bounds and greatest upper bounds only for finite families, and zonotopes do not have a binary least upper bound operator [GLGP12].

For example, Coq is able to use this instance to automatically understand the meaning of `x ∈ γ i` when `i` has type `iitv+⊥` (where `iitv` is the type of intervals, for which we already have defined an instance of `gamma_op`).

Comparison operators

Requiring a full-fledged order relation from a lattice in each abstract domain would require proving reflexivity, transitivity and antisymmetry. It appears that these properties are not needed for proving the soundness of the analysis, and moreover they can be tedious to prove (especially for transitivity), or even sometimes wrong (typically for antisymmetry). Instead, we use a *comparison operator*, which is not necessarily an order relation, defined as a Coq type class:

```
Class leb_op (A:Type) : Type :=
  leb : A -> A -> bool.
Notation "x ⊑ y" := (leb x y) (at level 39).
```

The only property that is needed about $\sqsubseteq$ for the correctness of the analyzer is the inclusion of concrete values (noted $\subseteq$) when the comparison returns `true`:

```
Class leb_op_correct A B {L:leb_op A} {G:gamma_op A B} : Prop :=
  leb_correct : ∀ a1 a2,   (a1 ⊑ a2) = true  ->  (γ a1) ⊆ (γ a2).
```

An instance of this type class would state that, for implicitly given comparison operator and concretization function, if the comparison operator returns `true`, then we have an inclusion of the concretizations. It is worth noting that when the comparison operator is an order relation, the instance of `leb_op_correct` captures exactly the monotonicity of γ.

Instances of `leb_op` and `leb_op_correct` for `A` can naturally be lifted to instances for `A+⊥`. We do this once and for all using generic type classes instances:

```
Instance leb_bot (A:Type) (L:leb_op A) : leb_op (A+⊥) :=
  (fun x y =>
     match x, y with
     | NotBot x, NotBot y => leb x y
     | Bot, _ => true
     | _, Bot => false
     end).

Instance leb_bot_correct A B (G:gamma_op A B) (L:leb_op A) :
  leb_op_correct A B -> leb_op_correct (A+⊥) B.
Proof. [...] Qed.
```

Join and meet

Since we do not have a proper order relation, we aim for a weak form of least upper bounds and greatest lower bounds, sufficient for our purpose. To guarantee the soundness of the analyzer, we only need to be able to build approximations of the concrete union and intersection. That is, we define $\sqcup$ and $\sqcap$ such that, for all x and y in the abstract domain:

$$\gamma(x) \cup \gamma(y) \subseteq \gamma(x \sqcup y) \tag{3.2}$$

$$\gamma(x) \cap \gamma(y) \subseteq \gamma(x \sqcap y) \tag{3.3}$$

This is different from least upper bounds and greatest lower bounds for several reasons: first, these statements do not claim any optimality property about $\sqcup$ nor $\sqcap$. This is important, since we want to be able to define non-optimal operators when optimal ones are too tedious

to implement and prove sound, or even do not exist. Second, these weak requirements let us prove the soundnesses of $\sqcup$ and $\sqcap$ independently from $\sqsubseteq$[2]: we will be able to define sound join and meet operators that will be actually *more precise* than the hypothetical least upper bound and greatest lower bound. This is typically the case when defining reduced products, as is discussed is Section 3.3.

Similarly to $\sqsubseteq$, the operators $\sqcup$ and $\sqcap$ and their specifications are defined as type classes, with Coq notations:

```
Class join_op (A B:Type) : Type :=
  join : A -> A -> B.
Notation "x □ y" := (join x y) (at level 40).
Class join_op_correct A B C
    {J:join_op A B} {GA:gamma_op A C} {GB:gamma_op B C} : Prop :=
  join_correct : ∀ x y, (γ x) ∪ (γ y) ⊆ γ (x □ y).

Class meet_op (A B:Type) : Type :=
  meet : A -> A -> B.
Notation "x □ y" := (meet x y) (at level 40).
Class meet_op_correct A B C
    {J:meet_op A B} {G:gamma_op A C} {GB:gamma_op B C} : Prop :=
  meet_correct : ∀ x y, (γ x) ∩ (γ y) ⊆ γ (x □ y).
```

It is worth noting that the result type of the meet and join functions need not be the same type as their parameters. Indeed, it is often the case that we want to be able to return an abstract value with a type with one additional $\bot$ element. For example, as we see in Section 6.2, the abstract domain of intervals has the implicit invariant that all the manipulated abstract values are non-empty intervals of type `iitv`. To define meet and still enforce this invariant, we make it return a value in `iitv+⊥`, but still taking parameters in `iitv` to avoid considering trivial cases. We can generically handle those trivial cases by defining instances of `join_op` and `meet_op` (and their proofs of soundness):

```
Instance join_bot_res A (J:join_op A A) : join_op A (A+⊥) :=
  [...].
Instance join_bot_args A (J:join_op A (A+⊥)) : join_op (A+⊥) (A+⊥) :=
  [...].
Instance meet_bot_args A (M:meet_op A (A+⊥)) : meet_op (A+⊥) (A+⊥) :=
  [...].
```

Bottom and top elements

When a contradiction is deduced by an abstract domain, the contradiction has to be propagated in all the abstract domains representing the same set of concrete values: this mechanism is a very basic kind of abstract domain cooperation. To implement the propagation of contradictions, we want the bottom element of abstract domains to be canonical and easily identifiable: we use the `_+⊥` types, presented earlier.

A convenient way of writing and proving code using these types is to see `+⊥` as an error monad. We use several monads in Verasco: in order to be able to define some generic monadic constructs, we use a `monad` type class. This also lets us use a common `do` notation *à la* Haskell for all our monads:

```
Class monad (M:Type -> Type) : Type :=
  { ret: ∀ {A:Type}, A -> M A;
    bind: ∀ {A B:Type}, M A -> (A -> M B) -> M B }.
Notation "'do' X <- A ; B" :=
```

[2] In particular, we do not require $x \sqsubseteq x \sqcup y$ nor $y \sqsubseteq x \sqcup y$, nor that $z \sqsubseteq x \sqcap y$ when z is an upper bound of x and y.

```
  (bind A (fun X => B))
  (at level 200, X ident, A at level 100, B at level 200).
```

Then, the +⊥ monad can be defined as an error monad:

```
Instance monad_botlift : monad botlift :=
{ ret := NotBot;
  bind A B f a :=
    match a with
    | Bot => Bot
    | NotBot x => f x
    end }.
```

Many algorithms in Verasco are programmed using the +⊥ monad, so a generic lemma for proving their soundness is of great help. Assume we have to prove that a call to `bind` returns a sound abstract value, that is `y ∈ γ (bind a f)` for some `y`, `f` and `a`, under the hypothesis that `a` is sound, i.e., `x ∈ γ a`. We encounter this situation each time we have to prove the soundness of a function using `bind`. We need to prove the corresponding property on `f` and deduce, using the following lemma, the soundness of the call to `bind`:

```
Lemma botbind_spec {A A' B B'} {GA : gamma_op A A'} {GB : gamma_op B B'} :
  ∀ (f:A->B+⊥) (x:A') (y:B'),
    (∀ a:A,   x ∈ γ a -> y ∈ γ (f a)) ->
    (∀ a:A+⊥, x ∈ γ a -> y ∈ γ (bind a f)).
```

The situation for top is different. We recall that $\top$ is the safe abstract value that can be used in any case. When lattices and Galois connections are used, it is the largest abstract value. In our situation, we use a simpler specification, stating that all concrete values are in the concretization of $\top$. Sometimes, there is clearly an element in the abstract domain that can be used as $\top$. For these cases, we use the following type classes and notation:

```
Class top_op (A:Type) : Type := top : A.
Notation "⊤" := top (at level 40).
Class top_op_correct A B {T:top_op A} {G:gamma_op A B} : Prop :=
  top_correct : ∀ x, x ∈ γ (⊤).
```

Similarly to $\sqcup$ and $\sqcap$, its specification, `top_op_correct`, uses γ instead of actually stating that this is a top element for $\sqsubseteq$.

Sometimes, there is no top element to be used in an abstract domain. In this situation we can use the combinator +⊤, defined in a similar way as +⊥:

```
Inductive toplift (A: Type) :=
| All : top_op (toplift A)
| Just : A -> toplift A.
Notation "x +⊤" := (toplift x) (at level 39).
```

It is easy to define corresponding instances to `gamma_op`, `meet_op`, `join_op`, `top_op`, and prove their soundnesses.

Similarly, we define standard instances of those type classes for pairs of abstract values. Depending on the context, a pair of abstract values can either concretize to a set of pairs of concrete values or to the intersection of the concretizations of components.

3.1.2. Widening Operator

The widening operator plays an important role when searching for invariants. As we see in Section 4.3, they are needed for inferring loop invariants and `goto` invariants at the function level.

The problem of finding invariants can be summarized as follows: given an abstract transformer F (a function transforming an abstract value into another abstract value) that represents the abstract interpretation of a loop body, find an abstract value I (the invariant), such that:

$$\gamma(F(I)) \subseteq \gamma(I) \tag{3.4}$$

That is, the invariant is preserved by one iteration of the loop. Additionally, we want to take into account the initial state of the loop. That is, for some given I_0, we also want:

$$\gamma(I_0) \subseteq \gamma(I) \tag{3.5}$$

Under these two constraints, finding a smaller I means better precision for the static analysis: as often, a tradeoff between precision and ease of computation has to be found. As an example, using $I = \top$ is certainly sound and easy to compute, but mostly useless.

The typical way to solve this problem in abstract interpretation is to use a binary widening operator $\triangledown$ and to compute the sequence defined by:

$$\begin{cases} X_0 & = I_0 \\ X_{n+1} & = X_n \triangledown F(X_n) \end{cases} \tag{3.6}$$

where $\triangledown$ satisfies regularity properties:

$$\forall\ A\ B, \quad \gamma(A) \subseteq \gamma(A \triangledown B) \tag{3.7}$$

$$\forall\ A\ B, \quad \gamma(B) \subseteq \gamma(A \triangledown B) \tag{3.8}$$

$$\begin{array}{l}\text{For all sequences } (A_i), \text{ the sequence } (A_i^{\triangledown}) \text{ defined by } A_0^{\triangledown} = A_0 \\ \text{and } A_{i+1}^{\triangledown} = A_i^{\triangledown} \triangledown A_{i+1} \text{ is ultimately stationary.}\end{array} \tag{3.9}$$

The property (3.7) maintains (3.5) during the iteration; (3.9) ensures that a stationary point will be eventually reached; and (3.8) ensures that this fixpoint verifies (3.4). As a result, this is a valid invariant.

In our quest for proof simplification, we want to minimize the number of proof obligations. The sole purpose of (3.9) is to prove the termination of the static analyzer, which is not essential for soundness. We do not formally prove such a property: instead, as we detail in Section 4.3, we use the *fuel* technique and avoid these tedious proofs[3]. It is worth noting that using fuel does not mean that termination does not hold for our abstract domains. We have proved, *on paper*, that such a property holds, so that the analyzer terminates in practice and in "paper theory". However, this is not ensured by the formal proofs.

Similarly, it is possible to avoid the formal proof of (3.8): instead, we can use the comparison operator $\sqsubseteq$ to directly check whether (3.4) holds at each step. If, for some n, we have $F(X_n) \sqsubseteq X_n$, then we know that X_n is a valid loop invariant. We still need to check on paper that there exists such an n. An easy solution is to enforce it when reaching a stationary point: a sufficient condition would be that, if $I = I \triangledown F(I)$, then $F(I) \sqsubseteq I \triangledown F(I)$. More generally without speaking of F, we check on paper the following property for our abstract domains:

$$\forall\ A\ B, \quad A = A \triangledown B \implies B \sqsubseteq A \triangledown B \tag{3.10}$$

This approach has another advantage: we can reach an invariant before reaching the stationary point. In this case, the analysis takes less time and the invariant is likely to be

[3]Given the amount of proofs dedicated to termination in [Pic05], we believe doing so saves us substantial amount of proof effort.

more precise.

Concerning (3.7), a simple solution to avoid its proof would be to replace F with $\lambda X.F(X \sqcup I_0)$ in the iteration (3.6). This would make (3.7) unnecessary. However, to avoid this unnecessary join at every iteration, we decided to keep this property formalized in Coq: in practice, proving (3.7) is not difficult, since widenings are implemented very similarly to joins, so that a large part of these proofs can be shared with the correctness of $\sqcup$.

Reductions after widening

A delicate problem with widening iterations is their bad behavior with respect to reductions [CCF+06, Section 6.4]. Indeed, some abstract domains use a reduction operation after each computation. Typically, a reduction does internal deductions in an abstract domain, and returns a refined abstract value, without changing its concretization. However, applying such a reduction after a widening can break termination. This is typically the case for octagon closure, as explained in Antoine Miné's PhD thesis [Min04, Section 3.7.2].

Reductions are important: they are used internally in some abstract domains (such as octagons) and, as we see in Section 3.3, as a mean of communication between abstract domains. A simple solution would involve forbidding reductions after widening. This is unsatisfactory for two reasons: some abstract domains require abstract values to be always reduced; and it would disallow abstract domains to take advantage of possibly better extrapolation strategies from other abstract domains (since communications after widening would not be possible).

We propose an original solution to the problem of loss of termination due to the use of reductions after widening: we observe that the result of a widening operation is used in two different places. It is used as a parameter for F, the abstract transfer function of the loop body, and as the first parameter of the next use of the widening. There is actually no reason to use the same values, and we propose to use two distinct abstract values. Our widening operation returns a pair, and we write the two components of the pair as follows:

$$A \triangledown_1 B = \texttt{fst}(A \triangledown B) \tag{3.11}$$
$$A \triangledown_2 B = \texttt{snd}(A \triangledown B) \tag{3.12}$$

The first component, which should not be reduced, is used as the first parameter for the next widening; the second component, on which reduction is allowed, is the precondition of the loop body for this iteration. The iteration is defined as follows:

$$\begin{cases} X_0 = Y_0 = I_0 \\ (Y_{n+1}, X_{n+1}) = Y_n \triangledown F(X_n) \end{cases} \tag{3.13}$$

We iterate this scheme until $F(X_n) \sqsubseteq X_n$. Properties (3.7), (3.10) and (3.9) are transformed into, respectively:

$$\forall\ A\ B, \quad \gamma(A) \subseteq \gamma(A \triangledown_1 B)\ \wedge\ \gamma(A) \subseteq \gamma(A \triangledown_2 B) \tag{3.14}$$
$$\forall\ A\ B, \quad A = A \triangledown_1 B\ \Longrightarrow\ B \sqsubseteq A \triangledown_2 B \tag{3.15}$$
$$\text{For all sequences } (A_i), \text{ the sequence } (A_i^\triangledown) \text{ defined by } A_0^\triangledown = A_0 \text{ and } A_{i+1}^\triangledown = A_i^\triangledown \triangledown_1 A_{i+1} \text{ is ultimately stationary.} \tag{3.16}$$

where only (3.14) needs to be formally proved in Coq.

Widening implementation

Requiring $\triangledown$ to return a pair has one additional advantage in our implementation: it offers the freedom of not using the same return type for $\triangledown_1$ and $\triangledown_2$. This can be helpful to implement non-trivial widening strategies needing additional information about the state of iteration. For example, one may want to implement an abstract domain that widens only every two iterations; this is made possible by storing an additional Boolean in the type of the first component.

To summarize, a widening operator should have type `A -> B -> A*B`, where `A` is the type of abstract values used for the iteration state, and `B` is the type of abstract values used for the analysis of program statements. An increasing function of type `B -> A` is needed to initialize the iteration, effectively computing Y_0 from I_0. Here are the type class definitions for widenings and their specifications:

```
Class widen_op (A B:Type) : Type :=
  { init_iter : B -> A;
    widen : A -> B -> A * B }.
Notation "x ▽ y" := (widen x y) (at level 40).
Class widen_op_correct A B C
    {W:widen_op A B} {GA:gamma_op A C} {GB:gamma_op B C} : Prop :=
  { init_iter_correct : ∀ x, γ x ⊆ γ (init_iter x);
    widen_incr : ∀ (x:A) y, γ x ⊆ γ (x ▽ y) }.
```

In this specification, the use of `γ (x ▽ y)` hides a somewhat involved typeclass inference mechanism: by using the generic instance of `gamma_op` for pairs, `γ (x ▽ y)` is interpreted as the intersection of the concretization of the two components of `x ▽ y`.

3.2. Abstract Domains Hierarchy

The global hierarchy of the Verasco static analyzer is described in Figure 3.1. It is divided into several modules that handle different features of the C programming language. Each module implements an interface, usually an abstract domain interface, described using either type classes or plain Coq records. An interface contains both functions (such as abstract transfer functions) and their specifications. This methodology helps us reason about the static analyzer, by making each module mostly independent from the others. If one would like to modify or re-implement such a module, she would have to learn only very little about the rest of the code.

At the top of the hierarchy, we use the CompCert compiler front-end, up to the C#minor intermediate language. Using a CompCert intermediate language has several advantages. First, because we share with CompCert its formal semantics, we can combine formal guaranties provided by Verasco together with formal guaranties provided by CompCert up to the generated assembly code. This combination leads to a safety theorem about the generated assembly code: that is, any C#minor that passes the analysis without raising an alarm compiles to assembly code that is free of run-time errors. This answers a common issue with C static analyzers: the semantics of C is so subtle and contains so many implementation-defined behaviors that it is unclear to know whether the analyzer has taken into account all the choices (and bugs) of the compiler. For example, the evaluation order of expressions or the memory layout of C structures may or may not be treated consistently in the analyzer and in the compiler. The second advantage of using the CompCert front-end is to share with CompCert the large effort of specifying and proving such a front-end. Indeed, C#minor is free of several high-level features of the C language, such as side effects in expressions, operator overloading, unspecified expression evaluation order, or multiple loop constructs.

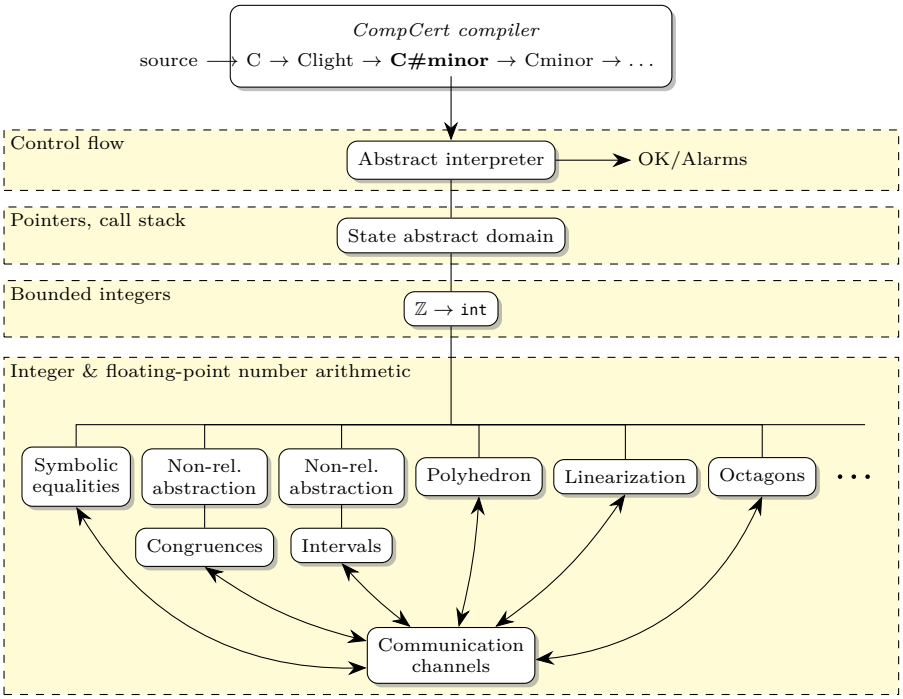


Figure 3.1: Verasco abstract domains hierarchy

The next component is the abstract interpreter. It iterates over the C#minor code, inferring abstract states at every program point, and checking for run-time errors. This component and its proofs are described in detail in Chapter 4.

The abstract states used by the abstract interpreter are computed using a state abstract domain that concretizes to stack and memory states. It includes a points-to domain for precisely handling pointers, an abstract domain for tracking run-time types, and some specialized domain for tracking allocation, deallocation and memory permissions. It has been designed, implemented and proved correct by Laporte as part of his PhD thesis [Lap15]. We do not give details about its content, and refer the reader to his dissertation instead. However, we will have a look at its interface in Section 3.2.1.

This state abstract domain is itself parameterized by a numerical abstract domain, capable of inferring numerical properties for the program. It is decomposed into several abstract domains, each of them handling a specific kind of properties. The whole combination is able to infer relational properties of numerical environments, containing either bounded machine integers or IEEE754 floating point numbers. Its interface is described in Section 3.2.2. It is composed of an abstract domain dedicated to the handling of boundedness of machine integers, described in Chapter 5, itself parameterized by a combination of abstract domains dealing with floating point numbers and ideal unbounded mathematical integers. Each of these domains share a common interface, and are combined using a special communication system described in Section 3.3. They include an abstract domain for symbolic equalities (Chapter 7); two non-relational abstract domains, intervals (Section 6.2) and integer congruences (Section 6.3); a polyhedron abstract domain using the *Verasco Polyhedral Library* developed independently by Fouilhé et al. [FMP13, FB14]; a linearization abstract

```
Class mem_dom (t iter_t:Type) := {
  leb_mem :> leb_op t;
  join_mem :> join_op t (t+⊥);
  widen_mem :> widen_op t (t+⊥);
  assign:
    ident -> expr -> t -> alarm_mon (t+⊥);
  assign_any:
    ident -> AbTy.t -> t -> alarm_mon (t+⊥);
  store:
    memory_chunk -> expr -> expr -> t -> alarm_mon (t+⊥);
  assume:
    expr -> t -> alarm_mon (t+⊥ * t+⊥);
  noerror:
    expr -> t -> alarm_mon unit;
  deref_fun:
    expr -> t -> alarm_mon (list (ident * fundef));
  push_frame:
    ident -> function -> list expr -> t -> alarm_mon (t+⊥);
  pop_frame:
    option (expr) -> option(ident) -> t -> alarm_mon (t+⊥);
  ab_malloc:
    expr -> option ident -> t -> alarm_mon (t+⊥);
  ab_free:
    expr -> t -> alarm_mon (t+⊥);
  ab_memcpy:
    Z -> Z -> expr -> expr -> t -> alarm_mon (t+⊥);
  init_mem:
    list (ident * globdef fundef unit) -> alarm_mon (t+⊥)
}.
```

Figure 3.2: State abstract domain interface

domain (Section 8.1); and an octagon abstract domain (Section 8.3). The non-relational abstract domains use a common adaptation layer to make them fit the relational interface (Section 6.1).

Every numerical abstract domain can be removed from the hierarchy, leading to a sound, but less precise analyzer. Even though the analyzer is useless in practice without some numerical domains such as intervals, it has better performance and still delivers reasonable precision without some other abstract domains, such as polyhedron or octagons.

3.2.1. State Abstract Domain Interface

The state abstract domain infers properties about the stack and memory state of a program. The implementation part of its interface is the `mem_dom` type class shown on Figure 3.2. As we have mentioned in Section 3.1.2, it depends on two types for abstract values. The type `t` is the usual type for abstract values, and `iter_t` is the type of iteration abstract states, containing potentially non-reduced abstract values and internal state for widening iterations strategies. This interface depends on a logging monad, `alarm_mon`, which tracks alarms (that is, potential bugs in the analyzed code) that could be triggered when executing a transfer function. This monad is another instance of the `monad` type class.

An instance of the `mem_dom` type class provides instances of `leb_op`, `join_op` and `widen_op`: a comparison operator, a join operator and a widening mechanism. It also contains abstract transfer functions for different concrete functions appearing in the C#minor semantics. Most of them depend on the type `expr` of C#minor expressions. A summary of their

behavior follows:

- `assign` writes the result of an expression to a temporary variable;
- `assign_any` writes non-deterministically any value of a given type to a temporary variable;
- `store` writes the result of an expression to a given memory location;
- `assume` returns two refinements of its input abstract state: the first one contains the additional assumption that the expression evaluates to a non-zero value, the second one assumes it evaluates to 0;
- `noerror` checks that an expression evaluate without raising an error;
- `deref_fun` returns the list of functions to which the result of an expression can point;
- `push_frame` and `pop_frame` handle function entry and leaving, that correspond, from the point of view of the state abstract domain, to pushing and popping[4] a frame onto the call stack;
- `ab_malloc`, `ab_free` and `ab_memcpy` are transfer functions abstracting the corresponding memory primitives;
- `init_mem` returns an abstract state corresponding to the state of the memory at program start-up time, given the global definitions of the C#minor program.

The specification coming with this interface relies on its concrete domain. It is a pair of a memory state (using CompCert memory model), and a representation of the call stack:

```
Definition concrete_state :=
  (list (ident * (temp_env * env))   *   mem)%type.
```

A stack frame is represented by the corresponding function identifier, and two local environments. Indeed, in C#minor, the variables in a stack frame are divided into two categories. On the one hand, *temporary variables* correspond to intermediate computation and variables whose address is not taken. It is not necessary to allocate memory for them and the CompCert back-end may store them in registers. Their values are directly stored in a map of type `temp_env`. On the other hand, *local variables* require to be explicitly stored in memory. The stack frame contains a map of type `env`, relating local variable identifiers to their addresses. Both are maps using identifiers as keys, but contain different data: `temp_env` contain values, while `env` contain addresses in the memory.

The state abstract domain has to provide concretization functions as instances to the `gamma_op` type class for both `t`, the type of regular abstract states and `iter_t`, the type of abstract states used during iteration with widening. A generic instance of the type class `gamma_op` exists for the `alarm_mon` monad: if some type `A` concretizes to some concrete type `B`, then `alarm_mon A` concretizes to `option B`, where `None` represents an error.

Several soundness theorems are established for the different parts of the interface, as shown in Figure 3.3. The specification of the comparison, join and widening operators relies on previously defined type classes. The specification of abstract transfer functions rely on concrete transfer functions, defined using the C#minor semantics. As usual in abstract

[4] If needed, `pop_frame` also performs the assignment of the given caller's temporary variable to the returned abstract value. This is the role of its `expr` and `ident` optional parameters.

```
Class mem_dom_spec {t iter_t: Type} (DOM: mem_dom t iter_t)
       (mem_gamma: gamma_op t concrete_state)
       (mem_gamma_iter: gamma_op iter_t concrete_state)
  : Prop := {
  leb_mem_correct:> leb_op_correct t concrete_state;
  join_mem_correct:> join_op_correct t (t+⊥) concrete_state;
  widen_mem_correct:> widen_op_correct iter_t (t+⊥) concrete_state;
  assign_sound: ∀ x e ab,
    Assign x e (γ ab)           ⊆ γ (assign x e ab);
  assign_any_sound: ∀ x ty ab,
    AssignAny x ty (γ ab)       ⊆ γ (assign_any x ty ab);
  store_sound: ∀ κ dst src ab,
    Store κ dst src (γ ab)      ⊆ γ (store κ dst src ab);
  assume_sound: ∀ e ab,
    Assume e (γ ab)             ⊆ γ (assume e ab);
  noerror_sound: ∀ e ab,
    Nonerror e (γ ab)           ⊆ γ (noerror e ab);
  deref_fun_sound: ∀ e ab,
    DerefFun e (γ ab)           ⊆ γ (deref_fun e ab);
  pop_frame_sound: ∀ ret rettemp ab,
    PopFrame ret rettemp (γ ab) ⊆ γ (pop_frame ret rettemp ab);
  push_frame_sound: ∀ fi f args ab,
    PushFrame fi f args (γ ab)  ⊆ γ (push_frame fi f args ab);
  malloc_sound: ∀ sz rettemp ab,
    Malloc sz rettemp (γ ab)    ⊆ γ (ab_malloc sz rettemp ab);
  free_sound: ∀ ptr ab,
    Free ptr (γ ab)             ⊆ γ (ab_free ptr ab);
  memcpy_sound: ∀ sz al dst src ab,
    Memcpy sz al dst src (γ ab) ⊆ γ (ab_memcpy sz al dst src ab);
  init_mem_sound: ∀ defs,
    Init_Mem defs               ⊆ γ (init_mem defs)
}.
```

Figure 3.3: State abstract domain specification

interpretation, they are specified as increasing functions over the lattice of concrete state sets.

For example, given a variable identifier `x`, an expression `e` and a set of concrete states `A`, the term `Assign x e A` denotes the set of options of possible concrete states after the assignment of `e` to `x` from a concrete state in `A`. The eventuality of an error during the assignment is represented by the fact that `None ∈ Assign x e A`. The soundness of the abstract transfer function `assign` is then specified by the inclusion `Assign x e (γ ab) ⊆ γ (assign x e ab)`.

3.2.2. Machine Numerical Domain Interface

An abstract domain implementing this interface infers properties about numerical environments: interactions with memory are no longer visible. In this section, concrete environments map variables to concrete numerical values, which can be either 32 bits or 64 bits integers or floating-point numbers:

```
Inductive mach_num : Type :=
| MNint : int -> mach_num
| MNint64 : int64 -> mach_num
| MNfloat : float -> mach_num
| MNsingle : float32 -> mach_num.
```

```
Inductive mach_num_ty : Type :=
| MNTint
| MNTint64
| MNTfloat
| MNTsingle.
```

Expressions from C#minor are not adapted for this layer of the static analyzer. Indeed, they contain memory-related operators, which are not handled by the numerical abstract domains. Thus, we also define a new kind of expressions and their semantics: they do not contain memory references, but still contain all the arithmetic operations of C#minor expressions:

```
Inductive mexpr : Type :=
| MEvar : mach_num_ty -> var -> mexpr
| MEconst : mach_num -> mexpr
| MEunop : unary_operation -> mexpr -> mexpr
| MEbinop : binary_operation -> mexpr -> mexpr -> mexpr.
```

References to the environment are expressed using `MEvar`, through an abstract type `var` of variable identifiers. They are typed: `MEvar` expressions are stuck if the identifier is not bound to a value of the appropriate type in the environment. Numerical constants of any type are represented using `MEconst`. Finally, unary and binary operations are factorized through `MEunop` and `MEbinop`. They feature the same arithmetic operations as C#minor: 32 bits and 64 bits integer arithmetic and bitwise operations, single and double precision floating-point arithmetic and conversion operators between the different types of values. To describe floating-point values, we rely on the Flocq formally verified library [BM11], already used in CompCert for specifying floating-point arithmetic [BJLM13, BJLM15].

The evaluation of these expressions is defined in big-step style using an inductive predicate:

```
Inductive eval_mexpr (ρ:var -> mach_num) : mexpr -> mach_num -> Prop :=
| eval_MEvar : ∀ id i ty,
  ρ id = i -> γ ty i ->
  eval_mexpr (MEvar ty id) i
[...].
```

A notable fact is that the state abstract domain guarantees that all the expressions it gives to the numerical layers are well typed. As a result, numerical abstract domains do not raise alarms when encountering type errors. This simplify the code of numerical abstract domains, but make the specification of numerical abstract domains slightly more complex. Indeed, error can no longer be defined as stuck expressions.

As a result, to fully define the semantics of these expressions, we also have to define erroneous expressions. This is done, again, using an inductive predicate in big-step style:

```
Inductive error_mexpr (ρ:var -> mach_num) : mexpr -> Prop :=
[...].
```

The errors tracked by this predicate are only arithmetic errors: division by zero, overflow when converting from a floating-point type to an integer type, and overflow in the count operand of shift expressions. This predicate does not take into account type errors, when there is a type mismatch between a sub-expression and its use. Similarly, the environments considered in this domain are total: there is no notion of undefined value; instead, an undefined value is represented at this level by any numerical value.

Using these expressions, we describe the interface and specification of machine numerical domains, as shown in Figure 3.4. Like in the previous section, we use a type class for this purpose. It can be divided into two parts: the first one is the programmatic interface, the second is its specification. Again, the specification depends on two concretization functions, on both the iteration abstract type `iter_t` and the type of usual abstract environments `t`. The programmatic interface contains type class instances for abstract domain operations: `top_op`, `leb_op`, `join_op` and `widen_op`. It contains the abstract transfer functions `assign`,

```
Class ab_machine_env (t iter_t: Type) : Type := {
  mach_top:> top_op (t+⊥);
  mach_leb:> leb_op t;
  mach_join:> join_op t (t+⊥);
  mach_widen:> widen_op iter_t (t+⊥);

  assign: var -> mexpr -> t -> t+⊥;
  forget: var -> t -> t+⊥;
  assume: mexpr -> bool -> t -> t+⊥;
  get_itv: mexpr -> t -> mach_num_itv+⊥;
  concretize_int: N -> mexpr -> t -> int_set+T+⊥;
  noerror: mexpr -> t -> bool;

  mach_gamma:> gamma_op t (var -> mach_num);
  mach_gamma_iter:> gamma_op iter_t (var -> mach_num);

  mach_top_correct:> top_op_correct (t+⊥) (var -> mach_num);
  mach_leb_correct:> leb_op_correct t (var -> mach_num);
  mach_join_correct:> join_op_correct t (t+⊥) (var -> mach_num);
  mach_widen_correct:>
    widen_op_correct iter_t (t+⊥) (var -> mach_num);
  assign_correct: ∀ x e ρ n ab,
    ρ ∈ γ ab ->
    n ∈ eval_mexpr ρ e ->
    ρ +[x => n] ∈ γ (assign x e ab);
  forget_correct: ∀ x ρ n ab,
    ρ ∈ γ ab ->
    ρ +[x => n] ∈ γ (forget x ab);
  assume_correct: ∀ e ρ ab b,
    ρ ∈ γ ab ->
    of_bool b ∈ eval_mexpr ρ e ->
    ρ ∈ γ (assume e b ab);
  get_itv_correct: ∀ e ρ ab,
    ρ ∈ γ ab ->
    (eval_mexpr ρ e) ⊆ γ (get_itv e ab);
  concretize_int_correct: ∀ n e ρ ab,
    ρ ∈ γ ab ->
    (eval_mexpr ρ e ∘ MNint) ⊆ γ (concretize_int n e ab);
  noerror_correct: ∀ e ρ ab,
    ρ ∈ γ ab ->
    noerror e ab = true ->
    error_mexpr ρ e ->
    False
}.
```

Figure 3.4: Machine numerical domain interface and specification

`forget` and `assume` and their specifications. The specifications of `forget` and `assign` use the notation `ρ +[x => n]` to denote the environment `ρ` updated by the new binding from `x` to `n`. Finally, it contains functions to query abstract environments. An interval for an expression can be queried using `get_itv`. If possible, `concretize_int` returns a finite set of integers an expression can evaluate to[5]. The eventuality of an arithmetic error arising when evaluating an expression is checked using `noerror`.

3.2.3. Ideal Numerical Domain Interface

Compared to the previous one, this ideal numerical level of Verasco has two main differences. First, the values we consider are simpler: they are either double precision floating-point numbers or ideal mathematical integers (i.e., elements of $\mathbb{Z}$). Second, the abstract domains do not need to track arithmetic errors: they are checked by the functor transforming the ideal numerical abstract domain to a machine numerical abstract domain (see Chapter 5).

This interface uses its own types for numerical values and their types:

```
Inductive ideal_num :=
| INf : float -> ideal_num
| INz : Z -> ideal_num.
```

```
Inductive ideal_num_ty :=
| INTz
| INTf.
```

We also define the type of *ideal expressions*, used in the whole numerical domain hierarchy. It is very similar to the type `mexpr` of machine numerical expressions, except that it is designed to compute over ideal numerical values:

```
Inductive iexpr : Type :=
| IEvar : ideal_num_ty -> var -> iexpr
| IEconst : ideal_num -> iexpr
| IEZitv : ZIntervals.itv -> iexpr
| IEunop : i_unary_operation -> iexpr -> iexpr
| IEbinop : i_binary_operation -> iexpr -> iexpr -> iexpr.
```

Typed variables are introduced with `IEvar`. `IEconst` wraps numerical constants. Non-determinism can be introduced by `IEZitv`, with an interval constraint in $\mathbb{Z}$. Arithmetic operations are built by using `IEunop` and `IEbinop`. These operators include arithmetic operations over $\mathbb{Z}$ and floating-point numbers, bitwise operations over $\mathbb{Z}$, comparisons, and conversions between both types.

Their semantics is defined in big-step style, using an inductive predicate:

```
Inductive eval_iexpr (ρ: var -> ideal_num): iexpr -> ideal_num -> Prop :=
| eval_IEvar: ∀ id i ty,
  ρ id = i -> i ∈ γ ty ->
  eval_iexpr (IEvar ty id) i
[...].
```

However, unlike machine expressions, we do not define a predicate of errors in ideal expressions. Indeed, all the arithmetic errors are checked by the machine arithmetic functor, so that ideal numerical domains only care about values of expressions when they evaluate with no error.

The interface for ideal numerical domains is shown in Figure 3.5. It is similar to the interface `ab_machine_env` shown on Figure 3.4, the major difference being that it works with ideal numerical values instead of machine numbers. Another difference lies in the fact that functions used to query abstract environments (e.g., `get_itv` in Figure 3.4) are embedded

[5]This is useful in the state abstract domain: when dereferencing a pointer, represented by an offset in a block of memory, this domain needs to know the values this offset can take.

in a *query channel* provided by the `idnc_get_query_chan` primitive. We explain further the contents and role of such channels in the next section.

3.3. Communication Between Numerical Abstract Domains

Several abstract numerical domains are needed in order to be able to check non-trivial programs without raising alarms. At one end of the spectrum, it is well known that the full reduced product [CC79], where each domain receives the maximal amount of information from others, is highly non-modular. It may not even exist or its computation may be too hard in practice. At the other extreme, direct products, where each domain reasons independently of the others are too weak; for example, in our analyzer, the interval domain is the only numerical domain able to deal with all numerical operations (including bitwise and floating-point operations), while other domains, such as octagons, can take advantage of intervals. We are seeking an intermediate solution where domains can communicate with each other, while keeping the analyzer easy to maintain from the point of view of its programmer. It should be easy to modify the hierarchy by adding, removing or replacing domains. Even though we do not expect the system to be as precise as the reduced product, we need the communication mechanism itself to be easily extensible, so that the sources of

```
Class ab_ideal_env_nochan (t iter_t:Type) : Type := {
  idnc_top:> top_op (t+⊥);
  idnc_leb:> leb_op t;
  idnc_join:> join_op t (t+⊥);
  idnc_widen:> widen_op iter_t (t+⊥);
  idnc_assign: var -> iexpr -> t -> t+⊥;
  idnc_forget: var -> t -> t+⊥;
  idnc_assume: iexpr -> bool -> t -> t+⊥;
  idnc_get_query_chan: t -> query_chan;

  idnc_gamma:> gamma_op t (var -> ideal_num);
  idnc_gamma_iter:> gamma_op iter_t (var -> ideal_num);
  idnc_top_correct:> top_op_correct (t+⊥) (var -> ideal_num);
  idnc_leb_correct:> leb_op_correct t (var -> ideal_num);
  idnc_join_correct:> join_op_correct t (t+⊥) (var -> ideal_num);
  idnc_widen_correct:> widen_op_correct iter_t (t+⊥) (var -> ideal_num);
  idnc_assign_correct: ∀ x e ρ n ab,
    ρ ∈ γ ab -> n ∈ eval_iexpr ρ e ->
    (ρ+[x => n]) ∈ γ (idnc_assign x e ab);
  idnc_forget_correct: ∀ x ρ n ab,
    ρ ∈ γ ab ->
    (ρ+[x => n]) ∈ γ (idnc_forget x ab);
  idnc_assume_correct: ∀ c b ρ ab,
    ρ ∈ γ ab ->
    (INz (if b then 1 else 0)) ∈ eval_iexpr ρ c ->
    ρ ∈ γ (idnc_assume c b ab);
  idnc_get_query_chan_correct: ∀ ab ρ,
    ρ ∈ γ ab ->
    ρ ∈ γ (idnc_get_query_chan ab)
}.
```

Figure 3.5: Ideal numerical domain interface and specification, without channels

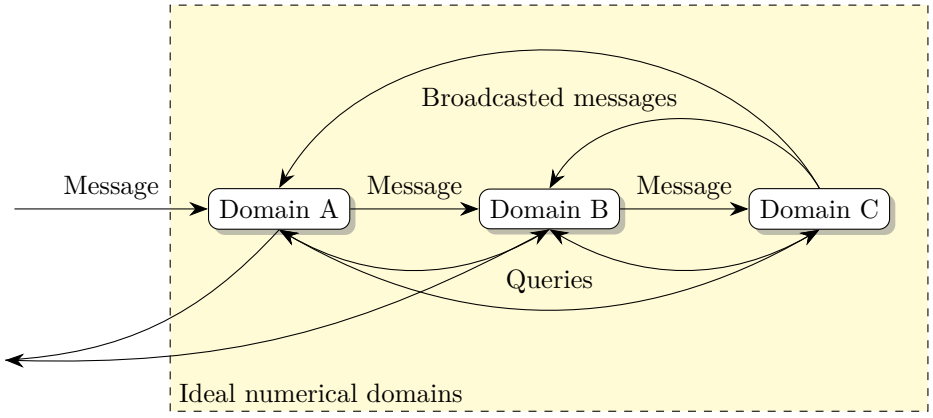


Figure 3.6: Query and message channels

imprecision can be mitigated.

We have followed some ideas of Astrée [CCF+06] to design a system of communication channels between numerical domains. We use this system to share information between domains modularly. Information can be either pulled by a domain from another domain using a query, or pushed by sending a message between domains.

In this section, we explain how the domains we used are combined using a product functor, taking two numerical abstract domains, and returning a numerical abstract domain, containing information from both domains[6]. A schematic view of the whole mechanism is depicted in Figure 3.6. Such domains use a different interface than the one presented in Figure 3.5, and a thin adaptation layer is used at the root of the hierarchy so that it meets the interface `ab_ideal_env_nochan`. We end our description by giving this internal interface in Section 3.3.3.

3.3.1. Message Channels

Message channels provide a kind of cooperation between domains. The general idea is that when an abstract domain assesses that some new fact may be useful to other domains, it sends a message containing this new information. A message is an element of a Coq inductive type, defined once and for all and shared by the whole hierarchy. It is customizable depending on the kind of information that needs to be shared. We define a concretization function for messages, which extends to message channels, defined as lists of messages. For example, if we need to define one category of messages, `Known_value_msg x v`, stating that the actual value of variable `x` is known to be `v`, we would write:

```
Inductive message : Type :=
| Known_value_msg : var -> ideal_num -> message
[...].

Instance message_gamma : gamma_op message (var -> ideal_num) :=
  fun m =>
    match m with
    | Known_value_msg x v => fun ρ => ρ x = v
```

[6] It is worth noting that this product is not commutative: the abstract operations are computed from left to right, and the information made available by the right component is not available when computing the left component.

```
  [...]
  end.
```

Each abstract operator returning an abstract environment (including abstract transfer functions and lattice operators) now also returns a message channel. Thus, we generalize the previous types of the abstract domain primitives. For `assign`, the type becomes:

```
assign: var -> iexpr -> t -> (t * message_chan)+⊥
```

This `assign` function has the exact same specification as in Figure 3.5: when it returns a pair `(ab, chan)` of an abstract environment and a message channel, its concretization is defined – using generic type classes – as the intersection of `γ(ab)` and `γ(chan)`[7].

Then, in every domain, a new primitive is implemented to process messages. The role of this primitive is to internalize the information brought by a message given as argument inside an abstract environment given as a second argument. Its type and specification follow:

```
process_msg: message -> t -> (t * message_chan)+⊥;
process_msg_correct: ∀ m ρ ab,
  ρ ∈ γ ab -> ρ ∈ γ m -> ρ ∈ γ (process_msg m ab)
```

It is worth noting that this function returns a message channel. While processing a message, it can decide to forward it, refine it before forwarding, send additional messages, or even discard it[8].

In the product functor, the abstract operations are executed in order, and then the messages produced by the first component are sent to the second component. To this end, we first define a function that extends `process_msg` to message lists (i.e., messages channels), by successively applying the `process_msg` function, and concatenating the returned messages channels:

```
Fixpoint process_msg_list (mchan:messages_chan)
                          (ab:A) (accu_mchan:messages_chan) :=
  match mchan with
  | nil => NotBot (ab, accu_mchan)
  | m::mchanq =>
     do (ab', mchan') <- process_msg m ab;
     process_msg_list mchanq ab' (mchan'++accu_mchan)%list
  end.
```

This piece of code uses the `do` notation for the `+⊥` monad defined in Section 3.1.1.

We can now define the abstract operations of the product functor. As an example, the definition for `assign` is:

```
assign x e ab qchan :=
  let '(a, b) := ab in
  do (a, mchana) <- assign x e a;
  do (b, mchanb) <- assign x e b;
  do (b, mchan) <- process_msg_list mchana b mchanb;
  ret ((a, b), mchan)
```

[7] There is no constraint on whether `γ(ab)` should be a subset of `γ(chan)`. While expected to be true in practice for most implementations of most operations, this property is wrong for typical implementations of the `process_msg` operation.

[8] Discarding may be useful, for example, when the domain already knows this piece of information, so the following abstract domains in the hierarchy should already have received this particular piece of information.

This way, any message produced by the first abstract domain is sent to all the following abstract domains. This idea is applied to all the described abstract operations that return an abstract environment: `assign`, `join`, `top`[9]…

Broadcasting messages

Note that in this setting, information cannot be propagated backwards in the chain of domains. To address this limitation, we introduce another sort of messages, called *broadcast messages*, which embed another message, and keep its concretization unchanged:

```
Inductive message : Type :=
[...]
| Broadcasted_msg : message -> message.

Instance message_gamma : gamma_op message (var -> ideal_num) :=
  fix message_gamma m :=
    match m with
    [...]
    | Broadcasted_msg m => message_gamma m
    end.
```

Domains simply forward those messages without reading their contents. Then, when such a message is returned by the last domain to the root of the hierarchy, it is unboxed and sent back to the whole hierarchy. To ensure termination, broadcasting messages produced during this second step are not processed at all.

Following the ideas of Section 3.1.2, the implementation of the widening operator for products of abstract domains apply the widening on both components, but forward messages only to the $\triangledown_2$ component, so that termination properties are maintained. Its type and implementation follow:

```
id_widen: iter_t -> t+⊥ -> iter_t * (t * messages_chan)+⊥;
id_widen x y :=
  let '(xa, xb) := x in
  let ya := do yab <- y; ret (fst yab) in
  let '(ita, wa) := id_widen xa ya in
  let yb := do yab <- y; ret (snd yab) in
  let '(itb, wb) := id_widen xb yb in
  ((ita, itb),
    do (a, mchana) <- wa;
    do (b, mchanb) <- wb;
    do (b, mchan) <- process_msg_list mchana b mchanb;
    ret ((a, b), mchan))
```

Code maintenance for message channels

From the point of view of the code maintainer, we believe message channels are very convenient. They provide a simple, powerful abstraction for domain cooperation, and they are very flexible. In order to add some new kind of communication, one can just add a new constructor to the `message` type, give it a concretization, and implement and prove correct the sender and the receiver. If removing or replacing an abstract domain is needed, modifying the hierarchy is possible without touching the `message` type: the messages that were processed by the changed domain will simply be ignored.

Messages used in Verasco

[9] Messages for the `top` operation are only provided for uniformity reasons: we do not expect them to improve precision.

```
Inductive message : Type :=
| Fact_msg        : iexpr -> bool -> message
| Known_value_msg : var -> ideal_num -> message
| Itv_msg         : var -> IdealIntervals.abs -> message
| Linear_zero_msg : T_linear_expr -> message
| Congr_msg       : var -> zcongr -> message
| Broadcasted_msg : message -> message.

Instance message_gamma : gamma_op message (var->ideal_num) :=
  fix message_gamma m ρ :=
    match m with
    | Fact_msg c b =>
       (INz (if b then 1 else 0)) ∈ eval_iexpr ρ c
    | Known_value_msg x v =>
       ρ x = v
    | Itv_msg x i =>
       ρ x ∈ γ i
    | Linear_zero_msg le =>
        eval_T_linear_expr ρ le 0
    | Congr_msg x zc =>
       match ρ x with
       | INz v => v ∈ γ zc
       | _ => False
       end
    | Broadcasted_msg m =>
       message_gamma m ρ
    end.
```

Figure 3.7: Messages in Verasco

The messages used in Verasco and their concretizations are listed in Figure 3.7. Here is a summary of their uses and meaning:

- `Fact_msg c b` messages are a replacement for the `assume` primitive that is traditionally used by abstract interpreters to refine an abstract environment using a Boolean predicate, typically when entering either branch of an `if` statement. They are only sent by higher layers of Verasco. They state that some Boolean expression evaluates to a known Boolean value.
- `Known_value_msg x v` is emitted when a domain knows exactly the value of variable `x`.
- `Itv_msg x i` and `Congr_msg x c` state that variable `x` is in interval `i` or has the congruence property `c`.
- `Linear_zero_msg le` states that the quasi-linear expression `le` evaluates to the real 0. As we will see, the linearization domain translates some messages to those, because they can be understood by linear numerical abstract domains.
- `Broadcasted_msg m` has already been mentioned.

3.3.2. Query Channels

Message channels are useful, but an additional mechanism is needed. Indeed, when a message comes from another domain, the receiver may not be able to internalize the information at the time of reception. For example, if the congruence domain sends a message stating that some variable is equal to 2 modulo 5, the interval domain is not able to express this

constraint using its own abstraction. Thus, in the future, if it learns that this variable lies in the interval $[2, 15]$, it will not be able to use the previously received congruence message to refine it to $[2, 12]$. A solution would be for the interval domain to query, when needed, the congruence information for this variable.

This example shows that sometimes, a domain may need to *query* another domain for information. That is, the initiative of sharing information may come from the receiving domain. This leads to our second kind of channels: query channels. A query channel is an element of a record type, whose fields are functions that can be called to query information. For example, in Verasco, some query returns the interval of values an expression may take. Query channels come with concretization functions; informally, an environment is in the concretization of a query channel if every query answer is valid for this environment:

```
Record query_chan : Type :=
  { get_itv: iexpr -> IdealIntervals.abs+⊤+⊥; [...] }.
Record query_chan_gamma (chan:query_chan) (ρ:var->ideal_num) : Prop :=
  { get_itv_correct: ∀ e, eval_iexpr ρ e ⊆ γ (chan.(get_itv) e);
    [...] }.
```

Each domain implements a function, `enrich_query_chan`, which refines the answers given by a query channel with the information contained in the abstract environment:

```
enrich_query_chan: t -> query_chan -> query_chan;
enrich_query_chan_correct: ∀ ab chan ρ,
  ρ ∈ γ ab ->
  ρ ∈ γ chan ->
  ρ ∈ γ (enrich_query_chan ab chan)
```

When a query is issued to the returned query channel, the channel can either decide to directly forward the query to the query channel given as argument (typically when the query has nothing to do with the kind of information the enriching domain stores), or answer the query directly, or a combination thereof. For example, an interval domain can forward a `get_itv` query to the base channel, and then intersect the result with an interval it computes itself.

The implementation of `enrich_query_chan` for products of domains is a straightforward composition of implementations for both components. Then, to compute the query channel associated with a hierarchy of abstract domains, it suffices to call `enrich_query_chan` at the root with a dummy query channel, whose concretization is the whole concrete domain. This way, the query will be handled first by the last domain and possibly forwarded all the way to the first domain in the hierarchy.

In order to enable domains to use the query channel, instead of an abstract environment, a pair of an abstract environment and a query channel is passed to abstract operations, abstracting the same set of concrete environments. This query channel contains the contribution of every abstract domains. The type of `assign` becomes:

```
assign: var -> iexpr -> t * query_chan -> (t * message_chan)+⊥;
```

On the other hand, when an abstract operation returns an abstract environment, an abstract domain may want to query the other domains about the concrete states whose abstraction is *currently being computed*. For example, when evaluating an abstract assignment, a domain may want to query other domains about the *new* value of the variable. For this reason, a query channel corresponding to the *result* of the operation is also given as an argument. Note that, as not all abstract domains have computed the operation, this query channel is partial and contains only the contributions of the domains located before in the hierarchy. The final type and specification of the `assign` function follows:

```
assign:
  var -> iexpr -> t * query_chan -> query_chan -> (t * message_chan)+⊥;

assign_correct: ∀ x e ρ n ab chan,
  ρ ∈ γ ab ->
  (ρ+[x => n]) ∈ γ chan ->
  n ∈ eval_iexpr ρ e ->
  (ρ+[x => n]) ∈ γ (assign x e ab chan)
```

Every abstract operation can be adapted similarly, except the comparison operator. Without query channels, its type would be `t -> t -> bool`, with the specification given in Section 3.1.1 by the `leb_op_correct`. So, logically, with query channels, its type would be:

```
t * query_chan -> t * query_chan -> bool
```

However, the query channel for the second argument would forbid any correct non-trivial implementation. Indeed, to prove an implementation returning `true` for some inputs, it would be necessary to prove that for those inputs, the concretization of the left hand side is included in the concretization of the query channel of the right hand side, which is impossible without enumerating every possible query. Thus, instead, the comparison operator does not take a query channel in its second argument:

```
id_leb: t * query_chan -> t -> bool
id_leb_correct: ∀ ab1_chan ab2 ρ,
  id_leb ab1 ab2 = true ->
  ρ ∈ γ ab1 -> ρ ∈ γ ab2
```

This signature can also be viewed another way: an abstract domain answers whether it is able to deduce the properties corresponding to the right hand side from properties of the left hand side abstract environment and query channel.

The implementation and proof of the product functor, combining both message channels and query channels are a bit tedious, but do not present noticeable difficulties. As an example, we give the implementation of the `assign` function, obtained simply by adding query channels to the implementation given in Section 3.3.1:

```
assign v e ab qchan :=
  let '((a, b), abqchan) := ab in
  do (a, mchana) <- assign v e (a, abqchan) qchan;
  let qchana := enrich_query_chan a qchan in
  do (b, mchanb) <- assign v e (b, abqchan) qchana;
  do (b, mchan) <- process_msg_list mchana (b, qchana) mchanb;
  ret ((a, b), mchan)
```

Code maintenance for query channels

Similarly to message channels, query channels are very convenient. Adding a new kind of query is easy: to this end, one has to add a field to the `query_chan` record, update its concretization function, and provide a dummy implementation to be passed to the first abstract domain of the chain. Similarly, it is robust to replacements or deletions in the hierarchy of domains.

Queries used in Verasco

Figure 3.8 lists the different queries used in the hierarchy of numerical domains of Verasco:

- `get_var_ty` tries to return the type of a given variable (i.e., integer or float). It is handled by a trivial auxiliary domain keeping track of variable types.

```
Record query_chan : Type :=
  { get_var_ty: var -> ideal_num_ty+⊤;
    get_itv: iexpr -> IdealIntervals.abs+⊤+⊥;
    get_congr: iexpr -> zcongr+⊥;
    get_eq_expr: var -> option ite_iexpr;
    linearize_expr: iexpr -> option T_linear_expr }.

Record query_chan_gamma (chan:query_chan) (ρ:var->ideal_num) :=
  { get_var_ty_correct: ∀ v, ρ v ∈ γ (chan.(get_var_ty) v);
    get_itv_correct:
      ∀ e, eval_iexpr ρ e ⊆ γ (chan.(get_itv) e);
    get_congr_correct:
      ∀ e, (eval_iexpr ρ e ∘ INz) ⊆ γ (chan.(get_congr) e);
    get_eq_expr_correct:
      ∀ x e, chan.(get_eq_expr) x = Some e ->
             eval_ite_iexpr ρ e (ρ x);
    linearize_expr_correct:
      ∀ e le, chan.(linearize_expr) e = Some le ->
      ∀ x, eval_iexpr ρ e x ->
           ∃ y, r_of_inum x = Some y /\ eval_T_linear_expr ρ le y
  }.
```

Figure 3.8: Queries used in Verasco

- `get_itv` and `get_congr` return an interval or a congruence relation for an expression given as parameter. They are mainly answered by the interval domain and the congruence domain. The octagon domain can also answer such queries in some cases.

- As we see in Chapter 7, `get_eq_expr` tries to return an equivalent expression of a given variable. It is answered by a dedicated abstract domain used, among others, to reconstruct Boolean expressions. Answers are in a slightly different expression type, that adds the possibility of if-then-else selector in prenex position.

- `linearize_expr` queries are answered by the linearization domain when linear abstract domains (e.g., octagons) need quasi-linear forms of expressions. When possible, an answer contains a quasi-linear expression evaluating to a real value corresponding to the return value of the given, possibly non-linear, expression.

A notable fact is that these queries are not only made by numerical abstract domains. The higher layers of Verasco use them when they need to get some information from the numerical domains. Typically, the abstract domain functor of Verasco dealing with machine-integer arithmetic will issue interval queries to analyze the overflow behavior of some arithmetic expressions. This is the reason why query channels appear in the interface `ab_ideal_env_nochan` listed in Figure 3.5.

3.3.3. Internal Interface for Numerical Abstract Domains

To conclude, we give in Figure 3.9 the final interface for numerical domains. It is very close to the type class `ab_ideal_env_nochan` given in Figure 3.5. It is in fact the same interface to which channels have been added in a systematic way. To this end, the type classes `top_op`, `leb_op`, `join_op`, `widen_op` and their specifications have been unfolded, leading to a more complex type class definition. However, the number and the complexity of primitives to be implemented is actually small, leading to simple implementations of abstract domains.

```
Class ab_ideal_env (t iter_t:Type) : Type := {
  id_top:
    query_chan -> (t * messages_chan)+⊥;
  id_leb:
    t * query_chan -> t -> bool;
  id_join:
    t * query_chan -> t * query_chan ->
    query_chan -> (t * messages_chan)+⊥;
  id_init_iter:
    t+⊥ -> iter_t;
  id_widen:
    iter_t -> (t * query_chan)+⊥ ->
    query_chan+⊥ -> iter_t * (t * messages_chan)+⊥;

  assign:
    var -> iexpr -> t * query_chan ->
    query_chan -> (t * messages_chan)+⊥;
  forget:
    var -> t * query_chan -> query_chan -> (t *  messages_chan)+⊥;

  process_msg:
    message -> t * query_chan -> (t * messages_chan)+⊥;
  enrich_query_chan:
    t -> query_chan -> query_chan;

  id_gamma:> gamma_op t (var -> ideal_num);
  id_gamma_iter:> gamma_op iter_t (var -> ideal_num);
  id_top_correct: ∀ chan ρ,
    ρ ∈ γ chan -> ρ ∈ γ (id_top chan);
  id_leb_correct: ∀ ab1 ab2 ρ,
    id_leb ab1 ab2 = true -> ρ ∈ γ ab1 -> ρ ∈ γ ab2;
  id_join_correct: ∀ ab1 ab2 chan ρ,
    ρ ∈ γ ab1 \/ ρ ∈ γ ab2 -> ρ ∈ γ chan ->
    ρ ∈ γ (id_join ab1 ab2 chan);
  id_init_iter_sound: ∀ ab,
    γ ab ⊆ γ (id_init_iter ab);
  id_widen_incr: ∀ ab1 ab2 chan ρ,
    ρ ∈ γ ab1 -> ρ ∈ γ chan -> ρ ∈ γ (id_widen ab1 ab2 chan);

  assign_correct: ∀ x e ρ n ab chan,
    ρ ∈ γ ab -> (ρ+[x => n]) ∈ γ chan ->
    n ∈ eval_iexpr ρ e -> (ρ+[x => n]) ∈ γ (assign x e ab chan);
  forget_correct: ∀ x ρ n ab chan,
    ρ ∈ γ ab -> (ρ+[x => n]) ∈ γ chan -> (ρ+[x => n]) ∈ γ (forget x ab chan);

  process_msg_correct: ∀ m ρ ab,
    ρ ∈ γ ab -> ρ ∈ γ m -> ρ ∈ γ (process_msg m ab);
  enrich_query_chan_correct: ∀ ab chan ρ,
    ρ ∈ γ ab -> ρ ∈ γ chan -> ρ ∈ γ (enrich_query_chan ab chan)
}.
```

Figure 3.9: Internal ideal numerical domain interface and specification

3.3.4. Related Work on Abstract Domains Combination

Combining abstract domains in order to get more precise abstract interpreters is not a new idea: Cousot and Cousot proposed this idea in one of the very first papers on abstract interpretation [CC79]. They described the direct product and the reduced product. However, as we explained, neither product is adapted to our setting: direct product is not precise enough, and reduced product is not practical, leading to non-modular, and hard to maintain code.

The Nelson-Oppen procedure [NO79] is a decision procedure for combinations of decidable logic theories. Several authors [CCM11, GT06] tried to adapt this method to abstract interpretation. Compared to ours, this method has several drawbacks in the context of static analysis. First, its theoretical foundations requires that function and predicate symbols of both theories be disjoint (except for the equality predicate), so the only information shared are (dis-)equalities between expressions. Second, the reduction phase used by this procedure has a higher computational complexity. Last, because they rely on saturating the set of shared (dis-)equalities, these methods do not include a way for an abstract domain to query others.

Corstesi, Le Charlier and Van Hentenryck [CLCVH94] introduced the notion of *open product*, which is a refinement of the direct product, based on queries. This idea is close to our notion of query channels. However, their extension to abstract operations, with additional query parameters, has several limitations. First, it prevents the implementation of operations to query other domains about the result of the operation. For this kind of communication, they rely on a refinement step after the operation. In our implementation of the weak reduced product between intervals and congruences, implementing such a refinement would require us to do a query for each variable, which is not practical. Second, their notion of query is limited, as they return only Booleans. Third, unlike in our system, it seems like this communication system is not used for binary operations, such as join, widening or comparison. Last, they do not provide any mean of communication on the initiative of the sender, which our messages do.

The inspiration of our work mainly comes from the combination of domains used by the Astrée static analyzer [CCF+06]. This analyzer features both *input channels* (the equivalent of our query channels) and *output channels* (the equivalent of our message channels). One of our main contributions is their formalization using the Coq proof assistant. Another difference is their use of the same formalism to describe the implementation of both input and output channels, based on two operators: $\text{EXTRACT}_{D^\#}$ and $\text{REFINE}_{D^\#}$. The operation $\text{EXTRACT}_{D^\#}$ can be seen as an equivalent to `enrich_query_chan` for query channels, and $\text{REFINE}_{D^\#}$ as an equivalent to `process_msg` for message channels. However, we do not have a refining operation solely based on query channels, nor an operation that would generically extract messages from an abstract environment. For the same reasons as for refinements in open products, we believe these should be integrated in abstract operations, and not part of the interface.

Chapter 4

Iterating Over the C#minor Language

The abstract interpreter of Verasco is the part of the analyzer that deals with the control flow of the program. As explained in Chapter 3, it uses a state abstract domain as parameter so that its main role is to apply the transfer functions of the state abstract domain according to the syntax of the program. Before entering into the details of the implementation of this component in Section 4.3, we have to explain the tools we use to prove properties on C#minor programs in Section 4.1 and Section 4.2.

In Verasco, we chose not to analyze source code directly. We rather analyze the C#minor intermediate language of CompCert. Indeed, we believe it is a good compromise between the lack of structure of low-level languages and the complexity of high-level languages. We describe the syntax and semantics of this language and explain this choice in Section 4.1.

The small-step operational semantics of C#minor is adapted to the simulation proofs of CompCert, but not convenient for verifying programs in practice: at first, we investigated the possibility of a direct proof of the abstract interpreter from the small step semantics of C#minor; but we realized that an intermediate component, consisting in an axiomatic semantics, closer to the interpreter structure, would make the proofs easier to understand and maintain.

We present in Section 4.2 the axiomatic semantics we used to prove the soundness of the abstract interpreter. This semantics uses Hoare *heptuples*, with two pre-conditions and four post-conditions to state properties about statements with complex control flow, with advanced control constructs such as `goto`s, `exit` statements or `return` statements anywhere in a function.

Finally, we explain the design and proof of soundness of the abstract interpreter in Section 4.3. We chose to define it structurally over the syntax tree of the language, rather than finding a global fixpoint on the control flow graph. Because the control structures of the language are quite complex, there are some peculiarities in the definition of the abstract interpreter.

4.1. The C#minor Language

4.1.1. Syntax

C#minor is an intermediate language of the CompCert compiler: it shares some constructs with C, but simplify many of them. Its syntax is presented in Figure 4.1. It is structured in

Expressions:

e	::=	t	reading of a temporary variable
		$\mid$ `&`x	address of a variable
		$\mid cst$	integer and floating-point constants
		$\mid op_1(e) \mid op_2(e, e')$	arithmetic operations
		$\mid$ `load`(τ, e)	memory load with size τ

Statements:

s	::=	`skip`	
		$\mid t := e$	assignment
		$\mid t :=?_{\texttt{int64}\mid\texttt{float}}$	non-deterministic assignment
		$\mid$ `store`(τ, e_1, e_2)	memory store with size τ
		$\mid t^? := e(\vec{e}) : sig$	function call
		$\mid (s_1; s_2)$	sequence
		$\mid$ `if` e `then` s_1 `else` s_2	conditional
		$\mid \texttt{switch}_{\texttt{int}\mid\texttt{int64}}\ e\ s_{tbl}$	multiway branch
		$\mid$ `loop` s	infinite loop
		$\mid$ `block` s	block supporting `exit`
		$\mid$ `exit` n	escape from the $n+1$ enclosing `block`s
		$\mid L : s$	define label L
		$\mid$ `goto` L	jump to label L
		$\mid$ `return` $\mid$ `return` e	function return

Jump tables:

s_{tbl}	::=	$[]$	
		$\mid$ `case` $n : s; s_{tbl}$	regular case
		$\mid$ `default` $: s; s_{tbl}$	default case

Functions:

f	::=	$name\ (\vec{p}) : sig\ \{$	
		`vars` $\overrightarrow{x[size]}$	local variables
		`temps` $\vec{t}$	temporary variables
		s	function body
		$\}$	

Figure 4.1: The syntax of C#minor

functions, statements and expressions. A program is composed of function definitions and global variable declarations. Local variables are of two kinds: addressable (their address can be taken with the & operator) and temporary (not resident in memory).

Expressions have no side effects: assignments, memory stores and function calls are statements. The arithmetic, logical, comparison, and conversion operators are roughly those of C, but without overloading: for example, distinct operators are provided for integer multiplication and floating-point multiplication. Likewise, there are no implicit casts: all conversions between numerical types are explicit.

C#minor has a rich collection of control structures. They encode at a lower level the variety of control structures of C. Like in C, they include both conditional branching (`if`) and multiway branching (`switch`). The different kinds of loops are encoded using one simpler infinite loop statement. Normal loop exit and special `break` and `continue` statements are encoded into a specific `block`/`exit` construct: `exit` n jumps to the end of the $(n+1)$-th enclosing `block` statement. C#minor features `goto` statements, directly inherited from C. Finally, like in C, C#minor includes a `return` statement with an optional expression parameter. Such a statement can be placed anywhere in a function.

The $t :=?_\tau$ construct is a non-deterministic assignment of any value of type `int64` or `float`. In the Coq development, it is represented by a call to a dedicated external function.

In this document, we slightly simplified the syntax of C#minor: indeed, in CompCert, it also contains builtins and external functions. However, the Verasco interpreter most often returns an alarm when encountering one of these. Exceptions to this rule are `malloc`, `free`, `memcpy` and the Verasco-specific functions `verasco_any_int64` and `verasco_any_double`, that have been represented by the $t :=?_\tau$ construct. The formalization of the analysis of external functions did not cause any notable difficulty.

4.1.2. Motivation for C#minor

The front-end of CompCert performs a lot of work to compile to the C#minor language, which is significantly easier to analyze than plain C:

- It pulls side effects out of expressions using temporary variables. This makes expressions pure, and much simpler to reason about.
- It resolves overloading completely, making all arithmetic operations explicit.
- It encodes all the loop constructs of C into infinite loops and `block`/`exit` constructs.
- It removes all the sources of non-determinism except non-deterministic assignment: in particular, it chooses an evaluation order for impure C expressions.

Why stop at C#minor and not use another, lower-level intermediate representation from CompCert? The subsequent passes of the compiler transform the program too much and lose important information. In particular, the local memory layout of Cminor features only one memory block for all the local variables of a function, so it would no longer be possible to detect overflows of local arrays. Also, an imprecision on a pointer on one variable may result in a store operation trashing the memory of other variables, hence a large loss of precision. Moreover, the structured control flow provided by this high-level language allows the interpreter to follow the statement structure in the interpreter, instead of reconstructing it using dedicated algorithms [Bou93].

Values:

v ::=	`undef`	undefined value
	| `int` n | `int64` n	32-bits or 64 bits machine integer
	| `single` f | `float` f	32-bits or 64-bits floating-point
	| `ptr` b δ	pointer to block b at offset δ

Program states:

S ::=	$\mathcal{S}(f, s, k, \rho, E, M)$	regular state
	$\mid \mathcal{C}(f, \vec{v}, k, M)$	call state
	$\mid \mathcal{R}(v, k, M)$	return state

Continuations:

k ::=	`stop`	initial configuration
	$\mid (s; k)$	continue with s, then do as k
	| `endblock` k	leave a `block`, then do as k
	| `returnto`$(t^?, f, \rho, E, k)$	return to caller

Figure 4.2: C#minor semantic objects

4.1.3. Formal Semantics

The dynamic semantics of C#minor is given in small-step style by a transition relation[1] $S \to S'$ between program states, an initial state and a set of final states. We say that a state S is *stuck* or *erroneous* if S is not final and there is no S' such that $S \to S'$. A program has an undefined semantics (i.e., "goes wrong") when either it does not have an initial state or when it can reach an erroneous state. That is, given the initial state S_0, there exists a sequence of states $S_1, \ldots, S_n$ such that $\forall\ 0 \leq i < n,\ S_i \to S_{i+1}$ and S_n is stuck.

The role of Verasco is to prove that a given program does not have an undefined semantics. That is, if it raises no alarm, the main theorem guarantees that it has an initial state and it cannot reach a stuck state.

Similarly to the Cminor language, whose precise description is given in [Ler09b], regular states $\mathcal{S}$ are composed of the definition of the function being currently executed, the statement of interest, a continuation describing actions to be performed next, and several environments: ρ gives in-memory addresses to local variables, E associates values to temporary variables and M is the memory state. The continuation k contains both the context of the current statement and the current call stack. The syntax of continuations and program states is given in Figure 4.2.

The transition relation depends on the big-step semantics for expressions, noted $\rho, E, M \vdash e \Downarrow v$, where the possible values v are described in Figure 4.2. We omit its definition: our iterator mainly forwards expressions to the state abstract domain without traversing them, so the code and proof of this part of the analyzer is mostly independent of this semantics. We refer the interested reader to [Ler09b] for the semantics of Cminor expressions, which is very close.

The small-step semantics of C#minor is close to the one of Cminor [Ler09b]. The differences lie in the different handling of `switch` statements, the squashing of the local variables in Cminor and a few historical changes in CompCert. We describe this semantics rule by

[1] In fact, transitions are labeled to take into account interaction with the external world. However, this has currently no importance for Verasco.

rule, trying to give insight to the reader unfamiliar with CompCert language semantics.

Assignments and `store`

The rules for assignments and store modify the corresponding environment according to the result of expressions evaluations:

$$\frac{\rho, E, M \vdash e \Downarrow v}{\mathcal{S}(f, t := e, k, \rho, E, M) \to \mathcal{S}(f, \texttt{skip}, k, \rho, E\{t \leftarrow v\}, M)} \tag{4.1}$$

$$\frac{v \text{ has type } \tau}{\mathcal{S}(f, t :=?_\tau, k, \rho, E, M) \to \mathcal{S}(f, \texttt{skip}, k, \rho, E\{t \leftarrow v\}, M)} \tag{4.2}$$

$$\frac{\rho, E, M \vdash e_1 \Downarrow \texttt{ptr}\ b\ \delta \qquad \rho, E, M \vdash e_2 \Downarrow v \qquad \texttt{Mem.store}(M, \tau, b, \delta, v) = \lfloor M' \rfloor}{\mathcal{S}(f, \texttt{store}(\tau, e_1, e_2), k, \rho, E, M) \to \mathcal{S}(f, \texttt{skip}, k, \rho, E, M')} \tag{4.3}$$

The rule for `store` depends on `Mem.store`. This *partial* function is included in the CompCert memory model [App14, Chapter 32]. If successful, it stores a given value with a given size at a given location in the memory, and returns the resulting memory state. This can produce an error for illegal or misaligned memory accesses.

Sequence

The rule defining the semantics of the sequence is:

$$\mathcal{S}(f, (s_1; s_2), k, \rho, E, M) \to \mathcal{S}(f, s_1, (s_2; k), \rho, E, M) \tag{4.4}$$

Executing a sequence of statements amounts to executing the first statement and queuing the second statement in the continuation. When the statement s_1 terminates, i.e., reduced to `skip`, it remains to execute s_2 then k. This behavior is captured by a second rule:

$$\mathcal{S}(f, \texttt{skip}, (s; k), \rho, E, M) \to \mathcal{S}(f, s, k, \rho, E, M) \tag{4.5}$$

Conditional statement

The rules for the conditional statement reduce the statement to one of its branches depending on whether the condition evaluates to 0 or a non-zero value:

$$\frac{\rho, E, M \vdash c \Downarrow \texttt{int}\ n \qquad n \neq 0}{\mathcal{S}(f, \texttt{if}\ c\ \texttt{then}\ s_1\ \texttt{else}\ s_2, k, \rho, E, M) \to \mathcal{S}(f, s_1, k, \rho, E, M)} \tag{4.6}$$

$$\frac{\rho, E, M \vdash c \Downarrow \texttt{int}\ 0}{\mathcal{S}(f, \texttt{if}\ c\ \texttt{then}\ s_1\ \texttt{else}\ s_2, k, \rho, E, M) \to \mathcal{S}(f, s_2, k, \rho, E, M)} \tag{4.7}$$

Infinite loop

A `loop` statement reduces to its body, but queuing a copy of itself in the continuation, so that it is executed again when the body ends, as described by rule (4.5):

$$\mathcal{S}(f, \texttt{loop}\ s, k, \rho, E, M) \to \mathcal{S}(f, s, (\texttt{loop}\ s; k), \rho, E, M) \tag{4.8}$$

Local exceptions: `block` and `exit`

Entering a `block` statement adds an `endblock` marker in the continuation to recall that a `block` context was entered:

$$\mathcal{S}(f, \texttt{block}\ s, k, \rho, E, M) \to \mathcal{S}(f, s, \texttt{endblock}(k), \rho, E, M) \tag{4.9}$$

On the one hand, if the execution of the `block` body terminates normally, we leave the block, and the marker disappears:

$$\mathcal{S}(f, \texttt{skip}, \texttt{endblock}(k), \rho, E, M) \to \mathcal{S}(f, \texttt{skip}, k, \rho, E, M) \tag{4.10}$$

On the other hand, if the execution reaches an `exit` statement, we unwind the continuation until we find the right number of enclosing `block`s. That is, if we encounter a sequence continuation, we have to ignore the queued statement (this corresponds to the statements that we are "jumping" over); if we encounter an `endblock` continuation, we either decrement the counter or restart the execution at this point if the counter is 0:

$$\mathcal{S}(f, \texttt{exit}\ n, (s; k), \rho, E, M) \to \mathcal{S}(f, \texttt{exit}\ n, k, \rho, E, M) \tag{4.11}$$

$$\mathcal{S}(f, \texttt{exit}\ 0, \texttt{endblock}(k), \rho, E, M) \to \mathcal{S}(f, \texttt{skip}, k, \rho, E, M) \tag{4.12}$$

$$\mathcal{S}(f, \texttt{exit}(n+1), \texttt{endblock}(k), \rho, E, M) \to \mathcal{S}(f, \texttt{exit}\ n, k, \rho, E, M) \tag{4.13}$$

Multiway branching: `switch`

To define the semantics of the `switch` statement, we first have to define two helper functions. The first one, `seq_of_tbl`, converts a jump table into a regular statement, by removing labels and using sequences:

$$\begin{aligned}
&\texttt{seq_of_tbl}\ [] = \texttt{skip} \\
&\texttt{seq_of_tbl}(\texttt{case}\ n : s; s_{tbl}) = (s; \texttt{seq_of_tbl}\ s_{tbl}) \\
&\texttt{seq_of_tbl}(\texttt{default} : s; s_{tbl}) = (s; \texttt{seq_of_tbl}\ s_{tbl})
\end{aligned}$$

The second one, $\texttt{select\_switch}_n(s_{tbl})$, returns the first matching entry (and all the following ones) for a given matched integer n, or [] if none match (and there is no `default` label):

$$\begin{aligned}
&\texttt{select_switch_default}\ [] = [] \\
&\texttt{select_switch_default}(\texttt{case}\ n : s; s_{tbl}) = \texttt{select_switch_default}(s_{tbl}) \\
&\texttt{select_switch_default}(\texttt{default} : s; s_{tbl}) = (\texttt{default} : s; s_{tbl}) \\
&\texttt{select_switch_case}_n(\texttt{case}\ n : s; s_{tbl}) = \lfloor \texttt{case}\ n : s; s_{tbl} \rfloor \\
&\texttt{select_switch_case}_n(\texttt{case}\ n' : s; s_{tbl}) = \texttt{select_switch_case}_n(s_{tbl}) \qquad \text{if } n \neq n' \\
&\texttt{select_switch_case}_n(\texttt{default} : s; s_{tbl}) = \texttt{select_switch_case}_n(s_{tbl}) \\
&\texttt{select_switch}_n(s_{tbl}) = \begin{cases} \texttt{select_switch_case}_n(s_{tbl}) & \text{if defined} \\ \texttt{select_switch_default}(s_{tbl}) & \text{otherwise} \end{cases}
\end{aligned}$$

Using these functions, we can state the rule for `switch`: it simply reduces such a statement

to the matched case:

$$\frac{\tau \in \{\texttt{int}, \texttt{int64}\} \qquad \rho, E, M \vdash e \Downarrow \tau\ n}{\mathcal{S}(f, \texttt{switch}_\tau\ e\ s_{tbl}, k, \rho, E, M) \to \mathcal{S}(f, \texttt{seq\_of\_tbl}(\texttt{select\_switch}_n\ s_{tbl}), k, \rho, E, M)} \tag{4.14}$$

Unstructured control: labels and `goto`

Labels are ignored by the regular control flow:

$$\mathcal{S}(f, L : s, k, \rho, E, M) \to \mathcal{S}(f, s, k, \rho, E, M) \tag{4.15}$$

To execute a `goto` statement, we first have to unwind the continuation until reaching the base continuation for the current call frame. This is the role of the `call_cont` function:

$$\begin{aligned}
&\texttt{call_cont}(s; k) = \texttt{call_cont}(k) \\
&\texttt{call_cont}(\texttt{endblock}(k)) = \texttt{call_cont}(k) \\
&\texttt{call_cont}(k) = k \qquad \text{otherwise}
\end{aligned}$$

After this unwinding, we find in the current function the statement to be executed *together with a continuation describing its context*, so that the same continuation is used for a given statement when entering it normally or via a goto. The `find_label` partial function computes the new statement and continuation[2]:

$$\texttt{find\_label}(L, (s_1; s_2), k) = \begin{cases} \texttt{find\_label}(L, s_1, (s_2; k)) & \text{if defined} \\ \texttt{find\_label}(L, s_2, k) & \text{otherwise} \end{cases}$$

$$\texttt{find\_label}(L, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2, k) = \begin{cases} \texttt{find\_label}(L, s_1, k) & \text{if defined} \\ \texttt{find\_label}(L, s_2, k) & \text{otherwise} \end{cases}$$

$$\texttt{find\_label}(L, \texttt{switch}_\tau\ e\ s_{tbl}, k) = \texttt{find\_label}(L, \texttt{seq\_of\_tbl}(s_{tbl}), k)$$

$$\texttt{find\_label}(L, \texttt{loop}\ s, k) = \texttt{find\_label}(L, s, (\texttt{loop}\ s; k))$$

$$\texttt{find\_label}(L, \texttt{block}\ s, k) = \texttt{find\_label}(L, s, \texttt{endblock}(k))$$

$$\texttt{find\_label}(L, L : s, k) = \lfloor s, k \rfloor$$

$$\texttt{find\_label}(L, L' : s, k) = \texttt{find\_label}(L, s, k) \text{ when } L \neq L'$$

Using these definitions, the rule for `goto` is:

$$\frac{\texttt{find\_label}(L, f.\texttt{body}, \texttt{call\_cont}(k)) = \lfloor s', k' \rfloor}{\mathcal{S}(f, \texttt{goto}\ L, k, \rho, E, M) \to \mathcal{S}(f, s', k', \rho, E, M)} \tag{4.16}$$

Function call and initial state

Calling a function is a rather complex operation: the machine has to evaluate the function pointer, dereference it and check that the signature matches, then evaluate the parameters and finally setup the new frame of the called function. Moreover, a part of this logic (but not all of it) is common with the setup of the context of the `main` function. To avoid duplication, in many CompCert languages, this process is divided into two steps. The

[2] In order to keep this definition structurally recursive in the Coq development, the definition of `seq_of_tbl` is inlined, making `find_label` a mutually recursive definition. We give here a simpler, equivalent definition.

first step corresponds to the computation made in the caller context, while the second step corresponds to the setup of the new stack frame in the context of the callee.

More precisely, the first step evaluates and dereferences the function pointer, checks the signature and evaluates the parameters. It ends in a special "call state" denoted using $\mathcal{C}$ (cf. Figure 4.2).

$$\frac{\rho, E, M \vdash e \Downarrow \texttt{ptr}\ b\ 0 \qquad \rho, E, M \vdash \vec{e} \Downarrow \vec{v} \qquad \texttt{funct}(b) = \lfloor f' \rfloor \qquad f'.\texttt{sig} = sig}{\mathcal{S}(f, t^? := e(\vec{e}) : sig, k, \rho, E, M) \to \mathcal{C}(f', \vec{v}, \texttt{returnto}(t^?, f, \rho, E, k), M)} \tag{4.17}$$

The `funct` function searches the global environment for the definition of the function at the given memory address.

The second step consists in setting up the new environment of the called function. It first checks there is no name clashes in the local and temporary variables and parameters. It allocates one block of memory for each local variable, and sets up the environment of temporary variables with undefined values and parameter values. This whole process is complex and mostly handled by the state abstract domain, so we do as if there were a partial function `env_setup`, whose definition is not shown, that does all this work. The second rule is:

$$\frac{\texttt{env\_setup}(f, \vec{v}, M_0) = \lfloor \rho, E, M \rfloor}{\mathcal{C}(f, \vec{v}, k, M_0) \to \mathcal{S}(f, f.\texttt{body}, k, \rho, E, M)} \tag{4.18}$$

The initial state of the program depends on a `main` function and an initial memory $M_{\texttt{init}}$, that are both defined by CompCert. Given those, the initial state is $\mathcal{C}(\texttt{main}, nil, \texttt{stop}, M_{\texttt{init}})$.

Function return and final states

Similarly, the process of returning from a function is decomposed into two steps. The first one evaluates the return value and frees the local variables; the second one returns to the calling context, optionally assigning the return value to a temporary variable.

The first step can be divided into three cases, leading to three different rules: either the function returns after having reached the end of its body, or it returns with a `return` statement without a parameter, or it returns with a `return` statement with a parameter. The three rules are:

$$\frac{\texttt{Mem.free\_env}(\rho, M) = \lfloor M' \rfloor \qquad k = \texttt{stop} \vee k = \texttt{returnto}(\ldots)}{\mathcal{S}(f, \texttt{skip}, k, \rho, E, M) \to \mathcal{R}(\texttt{undef}, k, M')} \tag{4.19}$$

$$\frac{\texttt{Mem.free\_env}(\rho, M) = \lfloor M' \rfloor}{\mathcal{S}(f, \texttt{return}, k, \rho, E, M) \to \mathcal{R}(\texttt{undef}, \texttt{call\_cont}(k), M')} \tag{4.20}$$

$$\frac{\rho, E, M \vdash e \Downarrow v \qquad \texttt{Mem.free\_env}(\rho, M) = \lfloor M' \rfloor}{\mathcal{S}(f, \texttt{return}\ e, k, \rho, E, M) \to \mathcal{R}(v, \texttt{call\_cont}(k), M')} \tag{4.21}$$

The second step is much simpler:

$$\mathcal{R}(v, \texttt{returnto}(t^?, f, \rho, E, k), M) \to \mathcal{S}(f, \texttt{skip}, k, \rho, E\{t^? \leftarrow v\}, M) \tag{4.22}$$

It simply models the return to the calling context stored in the continuation, and, optionally, the assignment of the temporary with the return value.

The valid final states are those of the form $\mathcal{R}(\texttt{int}\ r, \texttt{stop}, M)$, where r is the return status of the program.

Quasi-determinism

A key property of this semantics is that the only source of non-determinism is the non-deterministic assignment $t :=?_\tau$. It is essential for the proof technique used in CompCert. It also simplifies many aspects of the axiomatic semantics of C#minor.

Theorem 4.1.1 (Determinism of C#minor expressions)**.** *Let ρ, E, M be respectively a local environment, a temporary environment and a memory state. Let e be a C#minor expression and v_1 and v_2 values such that $\rho, E, M \vdash e \Downarrow v_1$ and $\rho, E, M \vdash e \Downarrow v_2$. Then $v_1 = v_2$.*

Theorem 4.1.2 (Determinism of C#minor)**.** *There is only one initial state. Moreover, let S be a C#minor state and S_1 and S_2 two states such that $S \to S_1$ and $S \to S_2$. Suppose S is not of the form $\mathcal{S}(f, t :=?_\tau, k, \rho, E, M)$. Then $S_1 = S_2$.*

4.2. An Axiomatic Semantics for C#minor

As explained earlier, an axiomatic semantic is more convenient than an operational semantic for verifying programs in practice. This is the reason why we designed an axiomatic semantics for C#minor. We first present the semantics itself by giving the rules allowing proving programs. Then we give its properties, together with their proof sketches. These properties include the soundness of the program logic with respect to the operational semantics, as well as deduction principles such as consequence rules, generalized conjunction and disjunction rules, and lemmas for proving loop unrolling and decreasing iterations techniques.

4.2.1. Semantic Rules

Because C#minor contains many advanced control structures, this logic has a complex structure: in general, there are two kinds of ingoing control flows and four kinds of outgoing control flows. Indeed, a statement can be entered either normally or using a `goto` statement to a label in the statement. On the other hand, a statement can be exited either normally, using a `goto` statement, using a `return` statement, or using an `exit` statement.

As a result, instead of traditional Hoare triples, we use Hoare "heptuples", with two pre-conditions and four post-conditions. This technique has already been used, for example, by Appel and Blazy in the context of the Cminor intermediate language [AB07]. In our setting, the Hoare judgment depends on an environment for local variables noted ρ (from local variable identifiers to memory cells). Its general form is as follows:

$$\rho \vdash \left\{ \begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array} \right\} \; s \; \left\{ \begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array} \right\} \tag{4.23}$$

The logical predicates $P[E, M]$, $P_{lbl}[L, E, M]$, $Q[E, M]$, $Q_{ret}[v, M]$, $Q_{exit}[n, E, M]$ and $Q_{goto}[L, E, M]$ depend on the variables between brackets. Here, E represents an environment for temporary variables (from temporary variable identifiers to values), M represents a memory state, L a label, v a value and n a natural integer.

Similarly, we have a Hoare judgment for functions:

$$\vdash \{P\}\ f\ \{Q\} \tag{4.24}$$

In this judgment, $P[\vec{v}, M]$ depends on the memory and on the values of parameters, while $Q[v, M]$ depends on the memory and on the return value of the function.

As we explain later, a third judgment, specific to jump tables, has four pre-conditions and four post-conditions. In Coq, the Hoare judgment for statements, the Hoare judgment for functions and the Hoare judgment for jump tables are defined using three large mutual coinductive types. Coinductiveness makes it possible to build proofs for recursive programs. Even if Verasco does not use this feature of the semantics, we think this is an interesting feature. We use Coq predicates directly for pre- and post-conditions: for example, the pre-condition P_{lbl} is a Coq term of type `ident -> temp_env -> mem -> Prop`.

It is worth noting that the semantics does not include the consequence rule itself. Instead, we prove that it is admissible given the other rules. Indeed, using the consequence rule in a coinductive setting would be unsound: it allows proving any specification for any program.

The `skip` statement

The rule for the `skip` statement simply states that the normal pre-condition should imply the normal post-condition:

$$\frac{P \Rightarrow Q}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\} \texttt{skip} \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.25}$$

A simpler rule would be to use the same predicate for P and Q, and replace other pre- and post-conditions with $\top$ and $\bot$, respectively. However, this additional level of generality is needed to prove the consequence rule.

Assignments and `store`

The rules for assignments and `store` are built such that the precondition both implies that the expressions will evaluate without error and that the postcondition will hold after the operation:

$$\frac{\forall EM,\ P[E,M] \Longrightarrow \left[\begin{array}{l} \exists v,\ \ \rho, E, M \vdash e \Downarrow v \\ \quad \wedge\ Q[E\{t \leftarrow v\}, M] \end{array}\right]}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ t := e\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.26}$$

$$\frac{\forall EM,\ P[E,M] \Longrightarrow \forall v : \tau,\ Q[E\{t \leftarrow v\}, M]}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ t :=?_\tau\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.27}$$

$$\frac{\forall EM,\ P[E,M] \Longrightarrow \left[\begin{array}{l} \exists b \delta v_2 M',\ \rho, E, M \vdash e_1 \Downarrow \texttt{ptr}\ b\ \delta \\ \qquad\qquad \wedge\ \rho, E, M \vdash e_2 \Downarrow v \\ \qquad\qquad \wedge\ \texttt{Mem.store}(M, \tau, b, \delta, v) = \lfloor M' \rfloor \\ \qquad\qquad \wedge\ Q[E, M'] \end{array}\right]}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ \texttt{store}(\tau, e_1, e_2)\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.28}$$

It is worth noting that this formulation of the rules is simplified by the determinism of the semantics of C#minor expressions. Otherwise, we would have to separately state that the

postcondition holds for *any* evaluation of expressions.

Sequence

The rule for the sequence is inspired from usual Hoare logic: the normal post-condition of the first statement is the normal pre-condition of the second statement. All the other predicates are shared:

$$\frac{\begin{array}{c}\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s_1\ \left\{\begin{array}{cc} R & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\} \\ \rho \vdash \left\{\begin{array}{c} R \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s_2\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\} \end{array}}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ (s_1; s_2)\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.29}$$

Conditional

Conditional statements are treated similarly:

$$\frac{\begin{array}{c}\rho \vdash \left\{\begin{array}{c} P_1 \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s_1\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\} \\ \rho \vdash \left\{\begin{array}{c} P_2 \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s_2\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\} \\ \forall EM,\ P[E,M] \Longrightarrow \left[\begin{array}{l} \exists n,\ \rho, E, M \vdash e \Downarrow \texttt{int}\ n \\ \quad \wedge\ n \neq 0 \Rightarrow P_1[E,M] \\ \quad \wedge\ n = 0 \Rightarrow P_2[E,M] \end{array}\right. \end{array}}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.30}$$

Again, we use the determinism of C#minor expressions in order to avoid quantifying over all the possible evaluations of the condition.

Infinite loop

Like in any axiomatic semantic, proving a Hoare tuple for an infinite loop involves providing an invariant I, implied by the pre-condition:

$$\frac{\rho \vdash \left\{\begin{array}{c} I \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s\ \left\{\begin{array}{cc} I & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\} \qquad P \Rightarrow I}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ \texttt{loop}\ s\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.31}$$

The control flow cannot escape normally from a `loop` statement: therefore, an arbitrary post-condition Q can be used.

Local exceptions: `block` and `exit`

The rule for `block` injects the normal post-condition into the `exit` post-conditions, after having shifted `exit` post-conditions by one:

$$\frac{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s\ \left\{\begin{array}{ccc} Q & & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to \lambda n.\ \left[\begin{array}{ll} Q & \text{if } n = 0 \\ Q_{exit}[n-1] & \text{if } n > 0 \end{array}\right] & \\ & & \texttt{goto} \to Q_{goto} \end{array}\right\}}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ \texttt{block}\ s\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.32}$$

The rule for the `exit` n statement imposes an implication between the pre-condition and the `exit` post-condition:

$$\frac{P \Rightarrow Q_{exit}[n]}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ \texttt{exit}\ n\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.33}$$

Unstructured control: labels and `goto`

Similarly to `exit` and `block`, the specification for a labeled statement is a matter of plumbing between the normal pre-condition, the `lbl` pre-condition and the `goto` post-condition:

$$\frac{\rho \vdash \left\{\begin{array}{c} P_{lbl}[L] \vee P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ L : s\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.34}$$

$$\frac{P \Rightarrow Q_{goto}[L]}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ \texttt{goto}\ L\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.35}$$

Multiway branching: `switch`

The rule for the `switch` statement depends on a third Hoare judgment for jump tables. There are many ways of entering into a partial jump table: either by a label, or by falling through from the previous case, or by jumping to a `case` label or by jumping to a `default` label. Thus, in this judgment, we have four pre-conditions and the usual four post-conditions. Its general form is as follows:

$$\rho \vdash \left\{\begin{array}{cc} \texttt{case} \to P_{case} & \texttt{def} \to P_{def} \\ \texttt{lbl} \to P_{lbl} & \texttt{ft} \to P_{ft} \end{array}\right\}\ s_{tbl}\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\} \tag{4.36}$$

This new judgment has its own inference rules, one for each constructor of jump tables. They use the same ideas as the one we already presented, hence we omit detailed

explanations:

$$\frac{P_{def} \Rightarrow Q \qquad P_{ft} \Rightarrow Q}{\rho \vdash \begin{Bmatrix} \texttt{case} \to P_{case} & \texttt{def} \to P_{def} \\ \texttt{lbl} \to P_{lbl} & \texttt{ft} \to P_{ft} \end{Bmatrix} [] \begin{Bmatrix} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{Bmatrix}} \tag{4.37}$$

$$\frac{\begin{array}{c} \rho \vdash \begin{Bmatrix} P_{case}[n] \vee P_{ft} \\ \texttt{lbl} \to P_{lbl} \end{Bmatrix} s \begin{Bmatrix} R & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{Bmatrix} \\ \rho \vdash \begin{Bmatrix} \texttt{case} \to P_{case} & \texttt{def} \to P_{def} \\ \texttt{lbl} \to P_{lbl} & \texttt{ft} \to R \end{Bmatrix} s_{tbl} \begin{Bmatrix} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{Bmatrix} \end{array}}{\rho \vdash \begin{Bmatrix} \texttt{case} \to P_{case} & \texttt{def} \to P_{def} \\ \texttt{lbl} \to P_{lbl} & \texttt{ft} \to P_{ft} \end{Bmatrix} \texttt{case}\ n : s;\ s_{tbl} \begin{Bmatrix} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{Bmatrix}} \tag{4.38}$$

$$\frac{\begin{array}{c} \rho \vdash \begin{Bmatrix} P_{def} \vee P_{ft} \\ \texttt{lbl} \to P_{lbl} \end{Bmatrix} s \begin{Bmatrix} R & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{Bmatrix} \\ \rho \vdash \begin{Bmatrix} \texttt{case} \to P_{case} & \texttt{def} \to \bot \\ \texttt{lbl} \to P_{lbl} & \texttt{ft} \to R \end{Bmatrix} s_{tbl} \begin{Bmatrix} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{Bmatrix} \end{array}}{\rho \vdash \begin{Bmatrix} \texttt{case} \to P_{case} & \texttt{def} \to P_{def} \\ \texttt{lbl} \to P_{lbl} & \texttt{ft} \to P_{ft} \end{Bmatrix} \texttt{default} : s; s_{tbl} \begin{Bmatrix} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{Bmatrix}} \tag{4.39}$$

The rule for the `switch` statement follows:

$$\frac{\begin{array}{c} \rho \vdash \begin{Bmatrix} \texttt{case} \to P_{case} & \texttt{def} \to P_{def} \\ \texttt{lbl} \to P_{lbl} & \texttt{ft} \to \bot \end{Bmatrix} s_{tbl} \begin{Bmatrix} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{Bmatrix} \\ \forall EM,\ P[E,M] \Longrightarrow \begin{bmatrix} \exists n,\ \rho, E, M \vdash e \Downarrow \tau\ n \\ \quad \wedge\ P_{case}[n, E, M] \\ \quad \wedge\ \texttt{select\_switch\_case}_n\ s_{tbl} = \emptyset \Rightarrow P_{def}[E, M] \end{bmatrix} \end{array}}{\rho \vdash \begin{Bmatrix} P \\ \texttt{lbl} \to P_{lbl} \end{Bmatrix} \texttt{switch}_\tau\ e\ s_{tbl} \begin{Bmatrix} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{Bmatrix}} \tag{4.40}$$

Returning from a function

Returning from a function by reaching the end of its body is directly handled by the rule for functions. For `return` statements, we use two dedicated rules that relate the normal ingoing control flow with the `ret` outgoing control flow:

$$\frac{\forall EM,\ P[E,M] \Longrightarrow \begin{bmatrix} \exists M'\ \texttt{Mem.free\_env}(\rho, M) = \lfloor M' \rfloor \\ \quad \wedge\ Q_{ret}[\texttt{undef}, E, M'] \end{bmatrix}}{\rho \vdash \begin{Bmatrix} P \\ \texttt{lbl} \to P_{lbl} \end{Bmatrix} \texttt{return} \begin{Bmatrix} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{Bmatrix}} \tag{4.41}$$

$$\frac{\forall EM,\ P[E,M] \Longrightarrow \left[\begin{array}{l}\exists v M',\ \rho, E, M \vdash e \Downarrow v \\ \qquad \wedge\ \texttt{Mem.free\_env}(\rho, M) = \lfloor M' \rfloor \\ \qquad \wedge\ Q_{ret}[v, E, M']\end{array}\right.}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl}\end{array}\right\}\ \texttt{return}\ e\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto}\end{array}\right\}} \tag{4.42}$$

Function call

Just like in the operational semantics, our axiomatic semantics decompose function call and return into two parts. The first part concerns only the calling process from the point of view of the caller. It uses the Hoare triple for the called function in the premise, and contains the Hoare tuple for the call statement as a conclusion:

$$\frac{\begin{array}{l}\forall EM, \qquad\qquad\qquad\qquad\qquad P[E,M] \\ \qquad\qquad\qquad\qquad\qquad\qquad \Downarrow \\ \exists b \vec{v} f,\ \rho, E, M \vdash e \Downarrow \texttt{ptr}\ b\ 0 \\ \qquad \wedge\ \rho, E, M \vdash \vec{e} \Downarrow \vec{v} \\ \qquad \wedge\ \texttt{funct}(b) = \lfloor f \rfloor \quad \wedge \quad f.\texttt{sig} = sig \\ \qquad \wedge\ \vdash \{\lambda \vec{v'} M'.\ \vec{v'} = \vec{v} \wedge M' = M\}\ f\ \{\lambda v' M'.\ Q[E\{t^? \leftarrow v'\}, M']\}\end{array}}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl}\end{array}\right\}\ t^? := e(\vec{e}) : sig\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto}\end{array}\right\}} \tag{4.43}$$

A second rule builds the specification of a function from specifications of its body. Not only it needs to take into account the function call and return process, but it also needs to make sure that pre- and post-conditions for special control flow are coherent. Namely, it ensures that the ingoing `lbl` control flow meets the outgoing `goto` control flow, that there is no jump to an unknown label, and that no `exit` statement will escape the function:

$$\frac{\begin{array}{l}\forall \vec{v} M_0, \qquad\qquad\qquad\qquad\qquad P[\vec{v}, M_0] \\ \qquad\qquad\qquad\qquad\qquad\qquad \Downarrow \\ \exists \rho E M I,\ \texttt{env\_setup}(f, \vec{v}, M_0) = \lfloor \rho, E, M \rfloor \\ \qquad \wedge\ (\forall L E M' k,\ I[L, E, M'] \Rightarrow \texttt{find\_label}(L, f.\texttt{body}, k) \text{ is defined.}) \\ \qquad \wedge\ \text{let } Q_{ret}[E', M'] := \\ \qquad\qquad \exists M_{ret},\ \texttt{Mem.free\_env}(\rho, M') = \lfloor M_{ret} \rfloor \quad \wedge \quad Q[\texttt{undef}, M_{ret}] \\ \qquad\qquad \text{in} \\ \qquad\qquad \rho \vdash \left\{\begin{array}{c}\lambda E' M'.\ E' = E \wedge M' = M \\ \texttt{lbl} \to I\end{array}\right\}\ f.\texttt{body}\ \left\{\begin{array}{cc} Q_{ret} & \texttt{ret} \to Q \\ \texttt{exit} \to \bot & \texttt{goto} \to I\end{array}\right\}\end{array}}{\vdash \{P\}\ f\ \{Q\}} \tag{4.44}$$

4.2.2. Consequence Rules

As usually in Hoare logic, the consequence rule allows us to weaken the post-condition and strengthen the pre-condition of a given judgment. In our setting, this rule is not part of the system, but can be deduced from others, by rewriting the whole proof of the judgment. There is one version for each three judgment, and they are all proved by mutual coinduction without any hidden difficulties. Moreover, given the large numbers of pre-

and post-conditions, we have two separate sets of rules: one for pre-conditions and one for post-conditions:

Theorem 4.2.1 (Consequence rules for statements)**.** *The following rules are admissible:*

$$\frac{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\} \qquad P' \Rightarrow P \qquad P'_{lbl} \Rightarrow P_{lbl}}{\rho \vdash \left\{\begin{array}{c} P' \\ \texttt{lbl} \to P'_{lbl} \end{array}\right\}\ s\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.45}$$

$$\frac{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\} \qquad Q \Rightarrow Q' \qquad Q_{ret} \Rightarrow Q'_{ret} \qquad Q_{exit} \Rightarrow Q'_{exit} \qquad Q_{goto} \Rightarrow Q'_{goto}}{\rho \vdash \left\{\begin{array}{c} P' \\ \texttt{lbl} \to P'_{lbl} \end{array}\right\}\ s\ \left\{\begin{array}{cc} Q' & \texttt{ret} \to Q'_{ret} \\ \texttt{exit} \to Q'_{exit} & \texttt{goto} \to Q'_{goto} \end{array}\right\}} \tag{4.46}$$

Theorem 4.2.2 (Consequence rules for functions)**.** *The following rules are admissible:*

$$\frac{\vdash \{P\}\ f\ \{Q\} \qquad P' \Rightarrow P}{\vdash \{P'\}\ f\ \{Q\}} \tag{4.47}$$

$$\frac{\vdash \{P\}\ f\ \{Q\} \qquad Q \Rightarrow Q'}{\vdash \{P\}\ f\ \{Q'\}} \tag{4.48}$$

Theorem 4.2.3 (Consequence rules for jump tables)**.** *The following rules are admissible:*

$$\frac{\rho \vdash \left\{\begin{array}{cc} \texttt{case} \to P_{case} & \texttt{def} \to P_{def} \\ \texttt{lbl} \to P_{lbl} & \texttt{ft} \to P_{ft} \end{array}\right\}\ s_{tbl}\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\} \qquad P'_{case} \Rightarrow P_{case} \qquad P'_{def} \Rightarrow P_{def} \qquad P'_{lbl} \Rightarrow P_{lbl} \qquad P'_{ft} \Rightarrow P_{ft}}{\rho \vdash \left\{\begin{array}{cc} \texttt{case} \to P'_{case} & \texttt{def} \to P'_{def} \\ \texttt{lbl} \to P'_{lbl} & \texttt{ft} \to P'_{ft} \end{array}\right\}\ s_{tbl}\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.49}$$

$$\frac{\rho \vdash \left\{\begin{array}{cc} \texttt{case} \to P_{case} & \texttt{def} \to P_{def} \\ \texttt{lbl} \to P_{lbl} & \texttt{ft} \to P_{ft} \end{array}\right\}\ s_{tbl}\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\} \qquad Q \Rightarrow Q' \qquad Q_{ret} \Rightarrow Q'_{ret} \qquad Q_{exit} \Rightarrow Q'_{exit} \qquad Q_{goto} \Rightarrow Q'_{goto}}{\rho \vdash \left\{\begin{array}{cc} \texttt{case} \to P_{case} & \texttt{def} \to P_{def} \\ \texttt{lbl} \to P_{lbl} & \texttt{ft} \to P_{ft} \end{array}\right\}\ s_{tbl}\ \left\{\begin{array}{cc} Q' & \texttt{ret} \to Q'_{ret} \\ \texttt{exit} \to Q'_{exit} & \texttt{goto} \to Q'_{goto} \end{array}\right\}} \tag{4.50}$$

4.2.3. Conjunction and Disjunction Rules

Conjunction and disjunction rules are used in the proof of the fixpoint iterator for loops and gotos (especially for proving the validity of loop unrolling and of the decreasing phase).

Recall the standard conjunction and disjunction rules from the literature:

$$\frac{\vdash \{P_1\}\ S\ \{Q_1\} \qquad \vdash \{P_2\}\ S\ \{Q_2\}}{\vdash \{P_1 \wedge P_2\}\ S\ \{Q_1 \wedge Q_2\}} \qquad \frac{\vdash \{P_1\}\ S\ \{Q_1\} \qquad \vdash \{P_2\}\ S\ \{Q_2\}}{\vdash \{P_1 \vee P_2\}\ S\ \{Q_1 \vee Q_2\}}$$

However, it can be noted that with the consequence rules, these rules are derivable from simpler ones, where pre- or post-conditions are kept equal:

$$\frac{\vdash \{P\}\ S\ \{Q_1\} \qquad \vdash \{P\}\ S\ \{Q_2\}}{\vdash \{P\}\ S\ \{Q_1 \wedge Q_2\}} \qquad \frac{\vdash \{P_1\}\ S\ \{Q\} \qquad \vdash \{P_2\}\ S\ \{Q\}}{\vdash \{P_1 \vee P_2\}\ S\ \{Q\}}$$

For the sake of generality, we proved more general rules, where the disjunction is replaced with an existential quantification over an arbitrary type and conjunction is replaced with an universal quantification over an arbitrary *inhabited* type[3]. These more general rules in the context of the Hoare statement for functions is proved by the following theorem:

Theorem 4.2.4 (Conjunction and disjunction rules for functions)**.** *The following rules are admissible[4]:*

$$\frac{\forall x : T,\ \vdash \{P\}\ f\ \{Q(x)\} \qquad \exists x : T, \top}{\vdash \{P\}\ f\ \{\textstyle\bigwedge_{x:T} Q(x)\}} \tag{4.51}$$

$$\frac{\forall x : T,\ \vdash \{P(x)\}\ f\ \{Q\}}{\vdash \{\textstyle\bigvee_{x:T} P(x)\}\ f\ \{Q\}} \tag{4.52}$$

Note that, for the generalized conjunction rule, the constraint stating that T is inhabited is important. Indeed, if used with an empty type, we would have the triple $\vdash \{P\}\ f\ \{\top\}$ for *any* function f and predicate P. This is clearly wrong, because, as we will see, our Hoare tuples ensure the safety of programs, which may not be enforced by P.

We have similar rules for statements:

Theorem 4.2.5 (Conjunction and disjunction rules for statements)**.** *The following rules are admissible:*

$$\frac{\forall x : T, \rho \vdash \begin{Bmatrix} P \\ \texttt{lbl} \to P_{lbl} \end{Bmatrix}\ s\ \begin{Bmatrix} Q(x) & \texttt{ret} \to Q_{ret}(x) \\ \texttt{exit} \to Q_{exit}(x) & \texttt{goto} \to Q_{goto}(x) \end{Bmatrix} \qquad \exists x : T, \top}{\rho \vdash \begin{Bmatrix} P \\ \texttt{lbl} \to P_{lbl} \end{Bmatrix}\ s\ \begin{Bmatrix} \bigwedge_{x:T} Q(x) & \texttt{ret} \to \bigwedge_{x:T} Q_{ret}(x) \\ \texttt{exit} \to \bigwedge_{x:T} Q_{exit}(x) & \texttt{goto} \to \bigwedge_{x:T} Q_{goto}(x) \end{Bmatrix}} \tag{4.53}$$

$$\frac{\forall x : T, \rho \vdash \begin{Bmatrix} P(x) \\ \texttt{lbl} \to P_{lbl}(x) \end{Bmatrix}\ s\ \begin{Bmatrix} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{Bmatrix}}{\rho \vdash \begin{Bmatrix} \bigvee_{x:T} P(x) \\ \texttt{lbl} \to \bigvee_{x:T} P_{lbl}(x) \end{Bmatrix}\ s\ \begin{Bmatrix} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{Bmatrix}} \tag{4.54}$$

First proof attempt

The first idea that comes to mind to prove these statements is by "merging" the proof trees of the premises. That is, by coinduction, we build a proof tree for the conclusion, using

[3] The less general rules are obtained by taking $T = \texttt{bool}$.

[4] We used the notations $\bigwedge_{x:T} Q(x)$ for $\forall x{:}T,\ Q(x)$ and $\bigvee_{x:T} P(x)$ for $\exists x{:}T,\ P(x)$.

as pre- and post-condition the conjunction (or disjunction) of the predicates used in the proof trees of the premises. However, such a proof method turned out to be harder than expected: indeed, some sort of axiom of choice is needed[5].

In order to understand why this proof uses the axiom of choice, let us try to handle the case of a sequence statement $(s_1; s_2)$ for the disjunction rule (4.54). We assume the rule has been proved for s_1 and s_2, and try to prove it for $(s_1; s_2)$. We have the following:

$$\forall x : T, \rho \vdash \left\{ \begin{array}{c} P(x) \\ \texttt{lbl} \to P_{lbl}(x) \end{array} \right\} \ (s_1; s_2) \ \left\{ \begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array} \right\} \tag{4.55}$$

and (4.54) for s_1 and s_2. We want to prove:

$$\rho \vdash \left\{ \begin{array}{c} \bigvee_{x:T} P(x) \\ \texttt{lbl} \to \bigvee_{x:T} P_{lbl}(x) \end{array} \right\} \ (s_1; s_2) \ \left\{ \begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array} \right\} \tag{4.56}$$

The idea would be to use the rule for the sequence (4.29), and use a combination of the coinduction hypotheses and of the consequence rules as premises. However, the question of which intermediate predicate R to use in (4.29) is tricky. Intuitively, we would want to take the disjunction of all the Rs used in the proof trees given by (4.55) (which necessarily use (4.29) at their head). However, this would need to extract *computationally* a predicate R for each x, or, equivalently, a function that returns R if given x. The existence of such an informational function from the family of non-informational proof trees can only by proved using the axiom of choice. The same problem appears for all the constructs where the proof tree is not syntax-directed, and both for the conjunction and disjunction rules.

Proof sketch

In the previous proof attempt, we were unable to apply the rule for the sequence because of the lack of a predicate that would be both a pre-condition for s_2 and a post-condition for s_1. There is, however, a simple candidate. Indeed, for a given set of post-conditions for a statement s, there is a *weakest pre-condition* for which the Hoare tuple is valid. Usually, this pre-condition is built syntactically, but we have a simple semantic way of building it, using higher order logic: it is the disjunction of all the valid pre-conditions.

More formally, let Q, Q_{ret}, Q_{exit} and Q_{goto} be post-conditions. We note $H(P, P_{lbl})$ the predicate defined as:

$$H(P, P_{lbl}) \equiv \rho \vdash \left\{ \begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array} \right\} \ s \ \left\{ \begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array} \right\} \tag{4.57}$$

We define P^{wp} as:

$$P^{wp} \equiv \bigvee_{P \mid H(P,\bot)} P \tag{4.58}$$

or equivalently, using only the logical constructs available in Coq:

$$P^{wp}[E, M] \equiv \exists P, \ H(P, \bot) \wedge P[E, M] \tag{4.59}$$

[5] In Verasco, we are not a priori against the idea of using such an axiom. However, we have another proof of this theorem that does not uses it; and, at the time we proved it, we did not figured out that the axiom of choice could actually solve the issue we are discussing here.

Similarly, we define P^{wp}_{lbl}:

$$P^{wp}_{lbl}[L, E, M] \equiv \exists P_{lbl},\ H(\bot, P_{lbl}) \wedge P_{lbl}[L, E, M] \tag{4.60}$$

Those ideas can also be used for the preconditions of the other two Hoare judgments.

Suppose now that we could prove these predicates are valid pre-conditions:

$$\rho \vdash \left\{\begin{array}{c} P^{wp} \\ \texttt{lbl} \to P^{wp}_{lbl} \end{array}\right\}\ s\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\} \tag{4.61}$$

The generalized disjunction rule (4.54) follows immediately using the consequence rule for preconditions. As an example, for the normal pre-condition, it suffices to prove:

$$\bigvee_{x:T} P(x) \Longrightarrow P^{wp} \tag{4.62}$$

which is immediate by using $P(x)$ as a witness and reapplying the consequence rule.

We prove (4.61) and the other analog lemmas for functions and jump tables by mutual coinduction with a slightly stronger coinduction hypothesis. Namely, we prove by coinduction that any pair of pre-conditions that implies P^{wp} and P^{wp}_{lbl} are valid. Rephrasing, we prove by coinduction that the following rule is admissible:

$$\frac{P \Rightarrow P^{wp} \qquad P_{lbl} \Rightarrow P^{wp}_{lbl}}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.63}$$

We sketch the proof only for the sequence and `loop` cases:

- For the sequence case $s = (s_1, s_2)$, we apply the sequence rule, using the weakest pre-condition of s_2 as intermediate predicate. The two premises of the sequence rule are proved using the coinduction hypotheses. It remains to prove the premises of the coinduction hypotheses: they are easily deduced from the premises and the consequence rules.
- For the `loop` s case, we apply the `loop` rule, using the weakest invariant that allows deducing the post-conditions. This invariant is defined similarly to P^{wp}. We then use the coinduction hypothesis and prove its premises using the consequence rules.

The proof of the conjunction rule is dual: instead of building the weakest pre-conditions, we build the strongest post-conditions and prove similar lemmas by coinduction. Note, however, that we need to make sure the precondition is strong enough to ensure at least the safety of the statement. For this reason, the dual of (4.63) has an additional hypothesis stating that the pre-conditions are valid for a dummy post-condition:

$$\frac{\begin{array}{c} Q \Rightarrow Q^{sp} \qquad Q_{ret} \Rightarrow Q^{sp}_{ret} \qquad Q_{exit} \Rightarrow Q^{sp}_{exit} \qquad Q_{goto} \Rightarrow Q^{sp}_{goto} \\ \rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s\ \left\{\begin{array}{cc} Q^0 & \texttt{ret} \to Q^0_{ret} \\ \texttt{exit} \to Q^0_{exit} & \texttt{goto} \to Q^0_{goto} \end{array}\right\} \end{array}}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.64}$$

This new premise becomes the inhabitedness premise of (4.51).

4.2.4. Loop Unrolling and Decreasing Iterations

Loop unrolling and decreasing iterations are used in Section 4.3.3 to improve the precision of the analysis of loops. They cannot be directly handled using the reasoning rule for infinite loops (4.31), because both reasoning schemes consists in using a different invariant.

Basically, loop unrolling consists in stating that `loop` s and (s; `loop` s) are equivalent, and proving a Hoare tuple for (s; `loop` s). This reasoning method is actually admissible in our logic:

Theorem 4.2.6 (Loop unrolling)**.** *The following rule is admissible:*

$$\frac{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s\ \left\{\begin{array}{cc} R & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\} \qquad \rho \vdash \left\{\begin{array}{c} R \\ \texttt{lbl} \to \bot \end{array}\right\}\ \texttt{loop}\ s\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}}{\rho \vdash \left\{\begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ \texttt{loop}\ s\ \left\{\begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\}} \tag{4.65}$$

This theorem is actually a bit stronger than stating that `loop` s and (s; `loop` s) are equivalent, because we can use $\bot$ instead of P_{lbl} as a precondition for `loop` s in the premise.

Its proof is a straightforward use of (4.31) and of the disjunction rule on the loop body: it suffices to use $R \vee I$ instead of I as loop invariant.

The decreasing iterations scheme is the dual of loop unrolling: instead of giving a separate proof for the first iterations, we give a separate proof for the *last* iterations. However, there is no way, in the syntax of C#minor, to distinguish the last iteration of a loop. For this reason, the statement of this theorem is more complex. It uses a notion of unstable invariant, which is true at every iteration but not necessarily strong enough to be stable. Formally, given a statement s and a local environment ρ, we say that I is an *unstable invariant* for pre- and post-conditions P, P_{lbl}, Q_{ret}, Q_{exit} and Q_{goto}, if there exists a predicate J such that $J \Rightarrow I$, $P \Rightarrow J$, and:

$$\rho \vdash \left\{\begin{array}{c} J \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s\ \left\{\begin{array}{cc} J & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array}\right\} \tag{4.66}$$

The reason for introducing this notion is that unstable invariants can be extended by one iteration of the loop:

Lemma 4.2.7. *If I is an unstable invariant for P, P_{lbl}, Q_{ret}, Q_{exit} and Q_{goto}, and the following Hoare judgment is derivable:*

$$\rho \vdash \left\{\begin{array}{c} I \\ \texttt{lbl} \to P_{lbl} \end{array}\right\}\ s\ \left\{\begin{array}{cc} I' & \texttt{ret} \to Q'_{ret} \\ \texttt{exit} \to Q'_{exit} & \texttt{goto} \to Q'_{goto} \end{array}\right\} \tag{4.67}$$

then, I' is an unstable invariant for P, P_{lbl}, Q'_{ret}, Q'_{exit} and Q'_{goto}.

The proof of this result is a use of the consequence rules and of the conjunction rule with $J \wedge I'$.

The following lemma can be used at the end to build the Hoare judgment for `loop` s:

Lemma 4.2.8. *If there exists an unstable invariant for P, P_{lbl}, Q_{ret}, Q_{exit} and Q_{goto}, then*

the following Hoare judgment is derivable (for any Q):

$$\rho \vdash \left\{ \begin{array}{c} P \\ \texttt{lbl} \to P_{lbl} \end{array} \right\} \ \texttt{loop}\ s \ \left\{ \begin{array}{cc} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{array} \right\} \tag{4.68}$$

Its proof is a trivial unfolding of definitions. Similar ideas can be used at the function level for invariants for labels, but they involve more complex definitions, which we omit here.

4.2.5. Soundness of the Hoare Logic

To prove the soundness of our set of rules, we define, for each of its three judgments, a semantic counterpart, defined in term of the operational semantics[6]. The Hoare judgments, denoted by $\vdash$, imply the semantic counterpart, denoted by $\vDash_n$. Then, from the semantic judgment for the `main` function, we deduce a whole-program safety theorem expressed directly in terms of the operational semantics.

A key point is that the final aim of such a judgment is only to prove the safety of the program. To that end, we first define a `safe` predicate, stating that some state will not go into an erroneous state after any number of steps:

$$\texttt{safe}\ S \ \equiv \ \forall S', \ \neg(S \to^* S' \ \wedge \ S' \text{ is stuck}) \tag{4.69}$$

We begin with the semantic Hoare judgment for functions, and we will deduce analogously those for statements and jump tables. The semantic counterpart for a function f states the safety of any call state of the form $\mathcal{C}(f, \vec{v}, k, M)$ under the hypothesis that $\vec{v}$ and M verify the pre-condition. This is too strong, though, because the Hoare logic cannot guarantee anything about what happens after the function returns: one can very well prove a Hoare judgment for a function, if this function is called with a continuation k that will fail just after, we will have trouble proving that the call state is safe. For this reason, the semantic judgment includes one additional hypothesis, stating that any return state $\mathcal{R}(v', k, M')$ sharing the same continuation k and for which v' and M' verify the post-condition, is safe. This hypothesis is called a *continuation safety hypothesis*, and the whole proof is done in continuation passing style: we prove that some state is safe by first proving it is safe for the first few steps, and invoke the continuation safety hypothesis for the end of the execution. Thus, the semantic judgment for function looks like:

$$\vDash \{P\}\ f\ \{Q\} \equiv \left[\begin{array}{l} \forall k, \ (k = \texttt{stop} \ \vee \ k = \texttt{returnto}(\ldots)) \\ \qquad \wedge \ (\forall v' M', \ Q[v', M'] \Rightarrow \texttt{safe}\ \mathcal{R}(v', k, M')) \\ \Longrightarrow \forall \vec{v} M, \ P[\vec{v}, M] \Rightarrow \texttt{safe}\ \mathcal{C}(f, \vec{v}, k, M) \end{array} \right. \tag{4.70}$$

The proof is done by using the technique of *step-indexing* [AM01]. That is, we use an induction scheme on the length of the potential erroneous trace. To this end, we defined the $\texttt{safe}_n$ predicate, indexed on n, a bound on the length of considered traces. We also have an indexed definition of the semantic judgment, which uses an intermediate definition

[6] Another approach, traditionally used in the literature [AB07], would be to define the Hoare judgments directly using their semantic counterparts. Then, each of the rules defined in Section 4.2.1 can be proved, as well as the consequence and disjunction rules. We can even use the step indexing mechanism as a form of coinduction. Some of the proofs are simpler using this other approach, because they sometimes do not need to enumerate all the language constructs. Yet, we could not see any way to extend the proof method to support the conjunction rule.

$K_{ret,n}[Q,k]$ for the continuation safety hypothesis:

$$\texttt{safe}_n\ S \ \equiv\ \forall n' S',\ \neg(S \to^{n'} S' \ \wedge\ S' \text{ is stuck } \ \wedge\ n' < n) \tag{4.71}$$

$$K_{ret,n}[Q_{ret},k] \ \equiv\ \forall v M,\ Q_{ret}[v,M] \Rightarrow \texttt{safe}_n\ \mathcal{R}(v, \texttt{call\_cont}(k), M) \tag{4.72}$$

$$\vDash_n \{P\}\ f\ \{Q\} \ \equiv\ \begin{bmatrix} \forall k,\ (k = \texttt{stop} \ \vee\ k = \texttt{returnto}(\ldots)) \wedge K_{ret,n}[Q,k] \\ \Longrightarrow \forall \vec{v} M,\ P[\vec{v},M] \Rightarrow \texttt{safe}_n\ \mathcal{C}(f,\vec{v},k,M) \end{bmatrix} \tag{4.73}$$

The semantic judgments for statements and for jump tables use one continuation safety hypothesis for each post-condition. The definition of these hypotheses are as follows:

$$K_n[Q,f,k,\rho] \equiv \forall E M,\ Q[E,M] \Rightarrow \texttt{safe}_n\ \mathcal{S}(f, \texttt{skip}, k, \rho, E, M) \tag{4.74}$$

$$K_{exit,n}[Q_{exit},f,k,\rho] \equiv \forall x E M,\ Q_{exit}[x,E,M] \Rightarrow \texttt{safe}_n\ \mathcal{S}(f, \texttt{exit}\ x, k, \rho, E, M) \tag{4.75}$$

$$\begin{aligned} K_{goto,n}[Q_{goto},f,k,\rho] \equiv \forall L k_0 E M,\ & Q_{goto}[L,E,M] \wedge \texttt{call\_cont}(k) = \texttt{call\_cont}(k_0) \\ & \Rightarrow \texttt{safe}_n\ \mathcal{S}(f, \texttt{goto}\ L, k_0, \rho, E, M) \end{aligned} \tag{4.76}$$

Then, the semantic judgment for statements follows:

$$\rho \vDash_n \left\{ \begin{matrix} P \\ \texttt{lbl} \to P_{lbl} \end{matrix} \right\}\ s\ \left\{ \begin{matrix} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{matrix} \right\}$$

$$\Vert\Vert$$

$$\begin{aligned} \forall f k,\ & K_n[Q,f,k,\rho] \wedge K_{ret,n}[Q_{ret},k] \\ & \qquad \wedge K_{exit,n}[Q_{exit},f,k,\rho] \wedge K_{goto,n}[Q_{goto},f,k,\rho] \\ \Longrightarrow\ & (\forall E M, \qquad P[E,M] \Rightarrow \texttt{safe}_n\ \mathcal{S}(f,s,k,\rho,E,M)) \\ \wedge\ & (\forall L E M s' k', \quad P_{lbl}[L,E,M] \wedge \texttt{find\_label}(L,s,k) = \lfloor s', k' \rfloor \\ & \qquad \Rightarrow \texttt{safe}_n\ \mathcal{S}(f,s',k',\rho,E,M)) \end{aligned}$$

and for jump tables:

$$\rho \vDash_n \left\{ \begin{matrix} \texttt{case} \to P_{case} & \texttt{def} \to P_{def} \\ \texttt{lbl} \to P_{lbl} & \texttt{ft} \to P_{ft} \end{matrix} \right\}\ s_{tbl}\ \left\{ \begin{matrix} Q & \texttt{ret} \to Q_{ret} \\ \texttt{exit} \to Q_{exit} & \texttt{goto} \to Q_{goto} \end{matrix} \right\}$$

$$\Vert\Vert$$

$$\begin{aligned} \forall f k,\ & K_n[Q,f,k,\rho] \wedge K_{ret,n}[Q_{ret},k] \\ & \qquad \wedge K_{exit,n}[Q_{exit},f,k,\rho] \wedge K_{goto,n}[Q_{goto},f,k,\rho] \\ \Longrightarrow\ & (\forall E M x s'_{tbl}, P_{case}[x,E,M] \wedge \texttt{select\_switch\_case}_x\ s_{tbl} = \lfloor s'_{tbl} \rfloor \\ & \qquad \Rightarrow \texttt{safe}_n\ \mathcal{S}(f, \texttt{seq\_of\_tbl}\ s'_{tbl}, k, \rho, E, M) \\ \wedge\ & (\forall E M s'_{tbl}, \qquad P_{def}[E,M] \wedge \texttt{select\_switch\_default}\ s_{tbl} = s'_{tbl} \\ & \qquad \Rightarrow \texttt{safe}_n\ \mathcal{S}(f, \texttt{seq\_of\_tbl}\ s'_{tbl}, k, \rho, E, M) \\ \wedge\ & (\forall E M, \qquad P_{ft}[E,M] \Rightarrow \texttt{safe}_n\ \mathcal{S}(f, \texttt{seq\_of\_tbl}\ s_{tbl}, k, \rho, E, M) \\ \wedge\ & (\forall L E M s' k', \quad P_{lbl}[L,E,M] \wedge \texttt{find\_label}(L,s,k) = \lfloor s', k' \rfloor \\ & \qquad \Rightarrow \texttt{safe}_n\ \mathcal{S}(f,s',k',\rho,E,M)) \end{aligned}$$

Using these definitions, we can prove the main soundness theorem for the axiomatic semantics:

Theorem 4.2.9 (Soundness)**.** *For each of the three syntactic classes of C#minor (functions, statements and jump tables), the Hoare judgment* $\vdash$ *imply their semantic counterpart* $\vDash_n$ *for any* n*.*

Applying this theorem on the `main` function with the `stop` continuation, and noticing a final state is always safe gives the following corollary:

Corollary 4.2.10 (Whole-program soundness)**.** *Assume that the program has an initial state with initial memory* M_{init}*. Moreover, suppose the following Hoare triple is provable:*

$$\vdash \{\lambda \vec{v} M.\ \vec{v} = \texttt{nil} \wedge M = M_{init}\}\ \texttt{main}\ \{\lambda v M.\ \exists r,\ v = \texttt{int}\ r\}$$

Then, the program cannot reach an erroneous state. That is, if S_0 *is the initial state:*

$$\forall S,\ S_0 \to^* S \implies \neg(S \text{ is stuck})$$

Proof sketch

The proof of the soundness theorem uses two induction schemes[7]: first, it uses the strong induction principle on n as is usual with step-indexing. This scheme is used to prove simultaneously the three parts of the theorem. Second, the proof uses the mutual induction scheme on statements and jump tables. We do not give the entire proof here. As for previous theorems, we focus on two cases: the sequence and the `loop` statement. So, let n be the step index, and let's assume the theorem is true for any $n' < n$ and only for sub-statements of considered statement when $n' = n$.

We first consider the sequence case $(s_1; s_2)$. We assume that the sequence rule (4.29) has been used with the intermediate predicate R to prove a Hoare judgment for $(s_1; s_2)$. We also assume the continuations safety hypotheses K_n, $K_{ret,n}$, $K_{exit,n}$ and $K_{goto,n}$ hold, and prove the two conclusions of the semantic judgment for $(s_1; s_2)$. They are:

$$\forall EM,\ P[E, M]\ \Rightarrow \texttt{safe}_n\ \mathcal{S}(f, (s_1; s_2), k, \rho, E, M) \tag{4.77}$$

$$\begin{aligned}\forall LEMs'k',\ P_{lbl}[L, E, M]\ \wedge\ \texttt{find\_label}(L, (s_1; s_2), k) = \lfloor s', k' \rfloor \\ \Rightarrow \texttt{safe}_n\ \mathcal{S}(f, s', k', \rho, E, M)\end{aligned} \tag{4.78}$$

To begin, we focus on (4.77): we exploit the fact:

$$\mathcal{S}(f, (s_1; s_2), k, \rho, E, M) \to \mathcal{S}(f, s_1, (s_2; k), \rho, E, M) \tag{4.79}$$

so that it suffices to prove[8]:

$$\forall EM,\ P[E, M]\ \Rightarrow \texttt{safe}_{n-1}\ \mathcal{S}(f, s_1, (s_2; k), \rho, E, M) \tag{4.80}$$

This is one of the two conclusions of the induction hypotheses[9] on s_1, so it suffices to prove the continuation safety hypotheses $K_{n-1}[R, (s_2; k)]$, $K_{ret,n-1}[(s_2; k)]$, $K_{exit,n-1}[(s_2; k)]$ and $K_{goto,n-1}[(s_2; k)]$.

[7] It is worth noting that, because it is defined by coinduction, we do not have an induction scheme on proof trees.

[8] The case n=0 is trivial. We assume $n > 0$.

[9] We can use here either structural induction on the statements or induction on n. We prefer the induction hypothesis on n because it is slightly stronger.

Because $\texttt{safe}_n$ is monotonic on n, $K_{ret,n-1}[(s_2;k)]$ and $K_{goto,n-1}[(s_2;k)]$ follow immediately from $K_{ret,n}[k]$ and $K_{goto,n}[k]$, respectively.

Similarly, $K_{exit,n-1}[(s_2;k)]$ follows from $K_{exit,n}[k]$ using the same monotonicity argument and noticing that $\mathcal{S}(f, \texttt{exit}\ x, (s_2;k), \rho, E, M) \to \mathcal{S}(f, \texttt{exit}\ x, k, \rho, E, M)$.

To prove $K_{n-1}[R, (s_2;k)]$, we first notice that:

$$\mathcal{S}(f, \texttt{skip}\ x, (s_2;k), \rho, E, M) \to \mathcal{S}(f, s_2, k, \rho, E, M) \tag{4.81}$$

so that it suffices to prove:

$$\forall EM,\ R[E,M]\ \Rightarrow \texttt{safe}_{n-2}\ \mathcal{S}(f, s_2, k, \rho, E, M) \tag{4.82}$$

This is easily proved by the induction hypothesis on s_2, whose continuation safety hypotheses are exactly K_n, $K_{ret,n}$, $K_{exit,n}$ and $K_{goto,n}$, which we had in hypotheses.

To prove (4.78), we distinguish two cases. If the target label is in the second statement[10], the induction hypothesis[11] on s_2 is exactly the proposition we need. If the target label is in s_1, we use the induction hypothesis on s_1, and prove the continuation safety hypotheses as done for the other conclusion.

The proof for the sequence statement did not use in a primordial way the induction hypothesis on n, the step index. This is symptomatic of the fact that the sequence statement is not a recursive construct: in fact, it would have been possible to prove termination using this same technique. On the contrary, the proof for the `loop` statement definitely needs the step-index induction. We see the same phenomenon for function call and the `goto` statement, which both introduce non-termination[12]. Let's see in more details the proof for the `loop` case.

We start similarly to the sequence case: we assume that the loop rule (4.31) has been used to prove the judgment, with some invariant I. We also assume the continuation safety hypotheses K_n, $K_{ret,n}$, $K_{exit,n}$ and $K_{goto,n}$ and prove separately both conclusions of the semantic judgment. We focus only on the first one: the second one uses similar ideas.

To this end, we first use $P \Rightarrow I$ and the transition for `loop` of the operational semantics to reduce to proving:

$$\forall EM,\ I[E,M]\ \Rightarrow \texttt{safe}_{n-1}\ \mathcal{S}(f, s, (\texttt{loop}\ s;k), \rho, E, M) \tag{4.83}$$

This is proved using either induction hypothesis. All the continuation safety hypotheses are treated similarly to the sequence case, except for $K_{n-1}[I, (\texttt{loop}\ s;k)]$. We use the transition:

$$\mathcal{S}(f, \texttt{skip}, (\texttt{loop}\ s;k), \rho, E, M) \to \mathcal{S}(f, \texttt{loop}\ s, k, \rho, E, M) \tag{4.84}$$

so that it suffices to prove:

$$\forall EM,\ I[E,M]\ \Rightarrow \texttt{safe}_{n-2}\ \mathcal{S}(f, \texttt{loop}\ s, k, \rho, E, M) \tag{4.85}$$

Here, we can either restart again and again the reduction we have just done, decrementing n by 2 at each iteration. Instead, we can also use the step index induction hypothesis, whose continuation safety hypotheses can be proved directly by using the hypotheses. This concludes the proof.

[10] i.e., $\texttt{find\_label}(L, s_1, (s_2;k)) = \emptyset$

[11] We use here the structural induction, because we keep the same step index n.

[12] It is interesting to note that the use of step index induction could be replaced with induction on the proof tree if we used an inductive proof tree instead of a coinductive one. We recover here the fact that such an inductive system could not be used to prove recursive functions.

Looking at the proof structure, it appears that the proof of this theorem has interesting computational contents: it is, in fact, a dependently typed interpreter of C#minor written in continuation passing style, with a parameter n giving an upper bound on the number of steps to perform. An alternative would be, instead of generating partial traces bound by n, to generate potentially infinite, coinductive traces.

4.3. Design and Proof of the Abstract Interpreter

One way to describe the role of the abstract interpreter is to automatically build program proofs using elements of the abstract domains as logical assertions in order to infer interesting specifications. For this reason, the design of the axiomatic semantics mimics the design of the interpreter. That is, the abstract interpreter is programmed using three main mutually recursive functions, one for each of the three judgments of the axiomatic logic. Each of them takes as parameter the pre-conditions (as abstract states of the state abstract domain) and returns, in the `alarm_mon` monad, the tuple of the inferred post-conditions (still as abstract states).

4.3.1. Transfer Functions

By abuse of notations, the three transfer functions defining the abstract interpreter are denoted by T. The definition of transfer functions for statements and jump tables are given in Figure 4.3 and Figure 4.4. For the sake of simplicity of the description, we omitted the fact that they are all defined within the `alarm_mon` monad: the actual definitions use monadic operators such as `ret` and `bind`. They are defined by mutual structural induction. They are both parameterized by r, the optional temporary to be assigned to the return value of the current function in the calling frame (this is used for `return`, which modifies the calling frame). The transfer function for statements takes as arguments:

- A, the input abstract state,
- A_{lbl}, the abstract states corresponding to the incoming goto control flow, actually represented as a map from labels to abstract states,
- the statement to be analyzed.

The transfer function for jump tables calls the `assume` transfer function on the fly to specialize the abstract state on cases. It takes as parameters:

- e, the expression on which case analysis is done,
- τ, the type of the case analysis (i.e., either `int` or `int64`),
- the jump table involved,
- A, the abstract state before the case analysis,
- A_{def}, the abstract sate assuming no `case` item is taken, evaluated using the `assume_def` function,
- A_{lbl}, the abstract states for labels (as for statements),
- A_{ft} the abstract state corresponding to the control flow falling through the previous case.

$$T_r\ (\texttt{skip})\ (A, A_{lbl}) = (A, \bot, \bot, \bot)$$

$$T_r\ (x := e)\ (A, A_{lbl}) = (\texttt{assign}\ x\ e\ A, \bot, \bot, \bot)$$

$$T_r\ (x :=?_\tau)\ (A, A_{lbl}) = (\texttt{assign\_any}\ x\ \tau\ A, \bot, \bot, \bot)$$

$$T_r\ (\texttt{store}(\tau, e_1, e_2))\ (A, A_{lbl}) = (\texttt{store}\ \tau\ e_1\ e_2\ A, \bot, \bot, \bot)$$

$$T_r\ (t^? := e(\vec{e}) : sig)\ (A, A_{lbl}) = \left(\bigsqcup_{f\ \in\ \texttt{deref\_fun}\ e\ A} T^{sig}_{t^?}\ f\ \vec{e}\ A, \bot, \bot, \bot \right)$$

$$T_r\ (s_1; s_2)\ (A, A_{lbl}) = (A'', A_{ret} \sqcup A'_{ret}, A_{exit} \sqcup A_{exit}, A_{goto} \sqcup A'_{goto})$$
$$\text{where } \begin{cases} (A', A_{ret}, A_{exit}, A_{goto}) = T_r\ s_1\ (A, A_{lbl}) \\ (A'', A'_{ret}, A'_{exit}, A'_{goto}) = T_r\ s_2\ (A', A_{lbl}) \end{cases}$$

$$T_r\ (\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2)\ (A, A_{lbl}) = T_r\ s_1\ (A_t, A_{lbl})\ \sqcup\ T_r\ s_2\ (A_f, A_{lbl})$$
$$\text{where } (A_t, A_f) = \texttt{assume}\ e\ A$$

$$T_r\ (\texttt{switch}_\tau\ e\ s_{tbl})\ (A, A_{lbl}) = T_r^{e:\tau}\ s_{tbl}\ (A, (\texttt{assume\_def}^{e:\tau}\ s_{tbl}\ A), A_{lbl}, \bot)$$

$$T_r\ (\texttt{loop}\ s)\ (A, A_{lbl}) = (\bot, A_{ret}, A_{exit}, A_{goto})$$
$$\text{where } (\_, A_{ret}, A_{exit}, A_{goto}) = \texttt{pfp}\ (\lambda X.\ T_r\ s\ (X, A_{lbl}))\ A$$

$$T_r\ (\texttt{exit}\ n)\ (A, A_{lbl}) = (\bot, \bot, (\lambda n'.\ \texttt{if}\ n' = n\ \texttt{then}\ A\ \texttt{else}\ \bot), \bot)$$

$$T_r\ (\texttt{block}\ s)\ (A, A_{lbl}) = (A' \sqcup A_{exit}(0), A_{ret}, (\lambda n.\ A_{exit}(n+1)), A_g)$$
$$\text{where } (A', A_{ret}, A_{exit}, A_g) = T_r\ s\ (A, A_{lbl})$$

$$T_r\ (\texttt{goto}\ L)\ (A, A_{lbl}) = (\bot, \bot, \bot, (\lambda L'.\ \texttt{if}\ L' = L\ \texttt{then}\ A\ \texttt{else}\ \bot))$$

$$T_r\ (\texttt{return}\ e^?)\ (A, A_{lbl}) = (\bot, \texttt{pop\_frame}\ e\ r\ A, \bot, \bot)$$

$$T_r\ (L : s)\ (A, A_{lbl}) = T_r\ s\ (A \sqcup A_{lbl}(L), A_{lbl})$$

Figure 4.3: Abstract interpreter for statements

$$\texttt{assume\_def}^{e:\tau}\ ([])\ A = A$$

$$\texttt{assume\_def}^{e:\tau}\ (\texttt{case}\ n : s; s_{tbl})\ A = \texttt{assume\_def}^{e:\tau}\ s_{tbl}\ A' \quad \text{where } (\_, A') = \texttt{assume}\ (e =_\tau n)\ A$$

$$\texttt{assume\_def}^{e:\tau}\ (\texttt{default} : s; s_{tbl})\ A = \texttt{assume\_def}^{e:\tau}\ s_{tbl}\ A$$

$$T_r^{e:\tau}\ ([])\ (A, A_{def}, A_{lbl}, A_{ft}) = (A_{def} \sqcup A_{ft}, \bot, \bot, \bot)$$

$$T_r^{e:\tau}(\texttt{case}\ n : s; s_{tbl})\ (A, A_{def}, A_{lbl}, A_{ft}) = (A'', A_{ret} \sqcup A'_{ret}, A_{exit} \sqcup A_{exit}, A_{goto} \sqcup A'_{goto})$$
$$\text{where } \begin{cases} (A_{case}, \_) & = \texttt{assume}\ (e =_\tau n)\ A \\ (A', A_{ret}, A_{exit}, A_{goto}) & = T_r\ s\ (A_{case}, A_{lbl}) \\ (A'', A'_{ret}, A'_{exit}, A'_{goto}) & = T_{tbl}\ s_{tbl}\ (A, A_{def}, A_{lbl}, A') \end{cases}$$

$$T_r^{e:\tau}(\texttt{default} : s; s_{tbl})\ (A, A_{def}, A_{lbl}, A_{ft}) = (A'', A_{ret} \sqcup A'_{ret}, A_{exit} \sqcup A_{exit}, A_{goto} \sqcup A'_{goto})$$
$$\text{where } \begin{cases} (A', A_{ret}, A_{exit}, A_{goto}) & = T_r\ s\ (A_{def}, A_{lbl}) \\ (A'', A'_{ret}, A'_{exit}, A'_{goto}) & = T_r^{e:\tau}\ s_{tbl}\ (A, \bot, A_{lbl}, A') \end{cases}$$

Figure 4.4: Abstract interpreter for jump tables

Both transfer functions return a 4-tuple containing:

- A', the abstract state at the end of the statement,
- A_{ret}, the abstract state after one of the `return` statements has been executed,
- A_{exit}, the abstract states after the execution of `exit` statement, actually represented as a list of abstract states,
- A_{goto}, the abstract states after the execution of `goto` statements, represented like A_{lbl} as a map from labels to abstract states.

They call several primitives of the state abstract domain, whose interface is shown in Figure 3.2. For the sake of simplicity of the description, we omitted a few details: we make implicit the fact that all the computations occur both in the `alarm_mon` and `+⊥` monads; we omitted that, in the case of an empty `switch` statement, it is necessary to check that the expression does not generate an error; and we omitted the capability of the transfer function for loops to unroll a few initial iterations. Finally, note that the transfer function for statements depends on the `pfp` function, that computes a loop invariant. We describe `pfp` in Section 4.3.3.

Moreover, we do not give much details about the transfer function for functions. It takes as parameters the function to be analyzed, the signature expected at call sites, the temporary to be assigned by the return value, the list of parameters, as expressions, and the input abstract state. It returns the abstract state after the return of the function. Because we do not check a priori that the program is not recursive, this function is defined by induction on an additional fuel parameter in `nat`, limiting the depth of the call stack. The case where the limit is reached is handled by emitting an alarm[13]. Here is a summary of the contents of the transfer function:

[13]In practice, after the extraction, we give for the fuel parameter the OCaml value recursively defined by `let rec inf = S inf`. This is a standard practice to avoid proving the termination of a Coq function.

- it checks that the signature of the called function is as expected;
- it does a few syntactic checks on the function (for example, no temporaries or parameter identifier should appear twice);
- it calls the `push_frame` primitive of the state abstract domain;
- it computes a post-fixpoint of the transfer function for statements applied to the function body, for which $A_{goto} \sqsubseteq A_{lbl}$. The transfer function for statements itself uses a recursive call of the transfer function for functions, with the fuel parameter decremented[14];
- using this fixpoint, it checks that the returned A_{goto} and A_{exit} abstract states do not leak control flow (that is, A_{exit} should be the empty list, and A_{goto} should contain non-bottom abstract states only for defined labels);
- it emulates the presence of a `return` statement at the end of the body;
- it returns the outgoing abstract state.

The global iterator constructs an initial abstract state by calling the `init_mem` primitive of the state abstract domain, searches for the `main` function, and runs the function transfer function on it.

4.3.2. Soundness

The abstract interpreter main theorem states that if the interpreter does not return an alarm (i.e., the returned list of alarms is `nil`), then the program cannot go wrong (i.e., reach an erroneous state):

```
Theorem iter_program_sound:
  ∀ res, iter_program unroll prog = (res, nil) ->
  ∀ tr, program_behaves (semantics prog) (Goes_wrong tr) ->
  False.
```

Statements and jump tables

In order to prove this theorem, we need first to give a specification to the iterator for statements and prove it sound. As already sketched, this specification ensures, given that the transfer function returns no alarms, that there exists a Hoare tuple consisting in the concretizations of the parameters, the statement in question and the concretizations of the resulting abstract states. The situation is, however, a bit more complex than it may appear at first. Indeed, as we have seen in Section 3.2.1, the concretization of an abstract state is a concrete state, containing not only the current memory state and environments for temporary and local variables, but also the identifiers of calling functions and the other stacked environments.

For this reason, the whole proof of soundness for the transfer function for statements is parameterized by:

This is sound, because if the extracted program terminates, then it is guaranteed that it has explored only a finite part of this infinite value, so the behavior would have been the same if we had replaced this infinite value by a finite value sharing that finite part.

[14] This is implemented by using *open recursion*: the transfer function for statements takes as parameter (using a section variable) the tranfer function for functions, which is always the same term, except for the fuel parameter that depend on the recursion depth.

- the tail σ of the call stack, of type `list (ident * (temp_env * env))`;
- the identifier i of the current function;
- the local environment ρ;
- the transfer function for functions, with its soundness proof[15];
- and r, the optional temporary to be assigned at function return.

Using these parameters, we can translate the abstract states to Hoare pre- and post-conditions:

$$P(A)[E, M] \equiv ((i, (E, \rho)) :: \sigma, M) \in \gamma(A) \tag{4.86}$$

$$P_{lbl}(A_{lbl})[L, E, M] \equiv ((i, (E, \rho)) :: \sigma, M) \in \gamma(A_{lbl}(L)) \tag{4.87}$$

$$P^{e:\tau}_{case}(A_{case})[n, E, M] \equiv \rho, E, M \vdash e \Downarrow \tau\ n\ \wedge\ P(A_{case})[E, M] \tag{4.88}$$

$$Q_{ret}(A_{ret})[v, M] \equiv \begin{cases} ((i, (E\{r \to v\}, \rho)) :: \sigma', M) \in \gamma(A_{ret}) & \\ \qquad\qquad\qquad\qquad\qquad \text{if } \sigma = (i, (E, \rho)) :: \sigma' & \\ (\texttt{nil}, M) \in \gamma(A_{ret})\ \wedge\ \exists n, v = \texttt{int}\ n \qquad \text{if } \sigma = \texttt{nil} & \end{cases} \tag{4.89}$$

$$Q_{exit}(A_{exit})[n, E, M] \equiv ((i, (E, \rho)) :: \sigma, M) \in \gamma(A_{exit}(n)) \tag{4.90}$$

$$Q(A) \equiv P(A) \tag{4.91}$$

$$Q_{goto}(A_{goto}) \equiv P_{lbl}(A_{goto}) \tag{4.92}$$

The soundness of the transfer functions for statements and jump tables is stated as follows:

Lemma 4.3.1. *1. If, for some s, A, A_{lbl}, A', A_{ret}, A_{exit} and A_{goto}, we have, without raising alarms:*

$$T_r\ s\ (A, A_{lbl}) = (A', A_{ret}, A_{exit}, A_{goto})$$

Then, the following Hoare tuple is derivable:

$$\rho \vdash \left\{ \begin{array}{c} P(A) \\ \texttt{lbl} \to P_{lbl}(A_{lbl}) \end{array} \right\}\ s\ \left\{ \begin{array}{cc} Q(A') & \texttt{ret} \to Q_{ret}(A_{ret}) \\ \texttt{exit} \to Q_{exit}(A_{exit}) & \texttt{goto} \to Q_{goto}(A_{goto}) \end{array} \right\}$$

2. If, for some e, τ, s_{tbl}, A, A_{def}, A_{lbl}, A_{ft}, A', A_{ret}, A_{exit} and A_{goto}, we have, without raising alarms:

$$T^{e:\tau}_r\ s_{tbl}\ (A, A_{def}, A_{lbl}, A_{ft}) = (A', A_{ret}, A_{exit}, A_{goto})$$

Then, the following Hoare tuple is derivable:

$$\rho \vdash \left\{ \begin{array}{c} \texttt{case} \to P^{e:\tau}_{case}(A) \\ \texttt{def} \to P(A_{def}) \\ \texttt{lbl} \to P_{lbl}(A_{lbl}) \\ \texttt{ft} \to P(A_{ft}) \end{array} \right\}\ s_{tbl}\ \left\{ \begin{array}{c} Q(A') \\ \texttt{ret} \to Q_{ret}(A_{ret}) \\ \texttt{exit} \to Q_{exit}(A_{exit}) \\ \texttt{goto} \to Q_{goto}(A_{goto}) \end{array} \right\}$$

The proof is by structural induction on the statement, following the definition of the transfer functions.

The following lemma proves the soundness of the transfer function for functions:

[15] Like in the definitions of the transfer functions, we use open recursion.

```
Fixpoint pfp_widen (fuel:nat) (f:abstate+⊥ -> alarm_mon outcome)
         (x:abstate+⊥) (y:abstate_iter): alarm_mon outcome :=
  let fx := f x in
  match fuel with
  | S fuel =>
    if (fst fx).(onormal) ⊑ x then fx
    else let '(y, x) := y ▽ (fst fx).(onormal) in
         pfp_widen fuel f x y
  | O =>
    do _ <- alarm_am "Not enough fuel to reach a fixpoint";
    fx
  end.
```

Figure 4.5: Increasing iterations with widening

Lemma 4.3.2. *If, for some f, sig, $t^?$, $\vec{e}$, A and A', we have, without raising alarms:*

$$T^{sig}_{t^?}\ f\ \vec{e}\ (A) = A'$$

Then, for every temporary environment E and memory state M such that $P(A)[E, M]$ holds, there exists parameter values $\vec{v}$ such that:

1. *f has signature sig;*
2. *$\rho, E, M \vdash \vec{e} \Downarrow \vec{v}$;*
3. *the following Hoare tuple is derivable:*

$$\vdash \{\lambda \vec{v}' M'.\ \vec{v}' = \vec{v}\ \wedge\ M' = M\}\ f\ \{\lambda v' M'.\ Q(A')[E\{t^? \leftarrow v'\}, M']\}$$

4.3.3. Inferring Invariants Using Widening and Decreasing Steps

In the definition of the transfer functions, we did not explain how the `pfp` function computes invariants of loops. We neither explained how invariants for `goto`s and labels are inferred. Both fixpoint computations use the same methodology. For the sake of simplicity, we concentrate on the `pfp` function, dedicated to loops.

This function is actually decomposed into two steps: the first step does increasing iterations with widening, as sketched in Section 3.1.2. The second step consists of a few decreasing iterations in order to refine the invariant.

Increasing iterations

The increasing iteration scheme is implemented in the `pfp_widen` function shown in Figure 4.5. It basically implements the iteration scheme of (3.13), but taking into account the fact that the transfer function of the loop body returns several abstract values in a monad. It does the computation ignoring alarms and returned abstract values except the normal one (the role of `(fst fx).(onormal)` is to extract this abstract value from the returned monadic value). It returns the tuple of post-conditions associated with the loop body instead of the invariant: we are actually not interested in the invariant directly, and this avoids computing angain the post-conditions. Finally, this computation uses recursion on a fuel parameter and emits an alarm in the case the fuel is exhausted.

The specification of this function is the following lemma:

```
Lemma pfp_widen_ok:
  ∀ fuel f x y o,
    pfp_widen fuel f x y = (o, nil) ->
    ∃ x0, f x0 = (o, nil) /\
          γ o.(onormal) ⊆ γ x0 /\
          γ y ∩ γ x ⊆ γ x0.
```

That is, if the iteration finishes without raising an alarm, then the result has been produced by applying the transfer function on an abstract state that has two properties. First, it is a stable invariant, meaning that it contains the loop post-condition. Second, it is greater than the intersection of the initial conditions of the iteration. This second condition is important to make sure that the invariant contains the loop initial pre-condition: indeed, `pfp_widen` is initially called with `(x:=pre)` and `(y:=init_iter pre)`, where `pre` is the loop pre-condition. With the specification of `init_iter`, this is enough to prove `γ pre ⊆ γ x0`.

This lemma would let us prove directly the Hoare tuple for the loop, by using `γ x0` as invariant. However, we use a common technique to improve this invariant using decreasing iterations.

Decreasing iterations

Intuitively, decreasing iterations are a kind of loop unrolling, at the end of the considered trace. They potentially provide a more precise loop invariant than the one provided by the increasing iteration.

Algorithmically, it consists in applying the transfer function a few more times, and joining the input state at each iteration. There is no need to check again that the result is still an invariant (intuitively, it corresponds to a loop unrolling at the end of the considered trace). If the new iterations create new alarms, we abort the process and use the previous invariant. Conversely, if the new iteration no longer has alarms but the previous one had, we have to check the inferred invariant is indeed an invariant using `⊑`, because there is no guarantee the previous invariant was a valid one (it generated alarms).

We use one last improvement: sometimes, we are able to statically infer that the loop ends at its first iteration[16]. This can be detected when the normal post-condition of the loop is `⊥`. It is very unlikely that in that case we may find a better invariant by decreasing iterations, because we use the exact same pre-condition for the loop body. So, we abort immediately the computation in this case. The final decreasing iteration function is defined by:

```
Fixpoint pfp_narrow (f : abstate+⊥ -> alarm_mon outcome) (lb:abstate+⊥)
         (steps:nat) (cur:alarm_mon outcome) : alarm_mon outcome :=
  match steps with
  | S steps =>
    if is_bot (fst cur).(onormal) then
      (* The normal outcome of the body is ⊥: it is very unlikely that we
         find an improvement. *)
      cur
    else
      (* Computing the next iteration... *)
      let next := f (lb ▯ (fst cur).(onormal)) in
      (* Looking at alarms. *)
      match snd cur, snd next with
      | _ :: _, nil =>
        (* We had alarms before, not any more: we need to check we have
           an invariant. *)
        if (fst next).(onormal) ⊑ (fst cur).(onormal)
```

[16]This is typical when using initial loop unrolling or a `{do ... while(0)}` construct in a macro.

```
        then pfp_narrow f lb steps next
        else cur
    | nil, _ :: _ =>
        (* We had no alarms before, but now we have one. Just forget
           this step. *)
        cur
    | nil, nil | _ :: _, _ :: _ =>
        (* Otherwise: continue the iteration. *)
        pfp_narrow f lb steps next
    end
| 0 => cur
end.
```

Note that we do not use a narrowing operator to automatically infer the number of steps needed to reach a fixpoint at the decreasing phase. Instead, we call this function with a small number of steps: performing a single step seems to be a good compromise in practice.

Combining the increasing and decreasing phases, the `pfp` function is defined by:

```
Definition pfp (f : abstate+⊥ -> alarm_mon outcome) (lb:abstate+⊥) :=
  let fp := pfp_widen fuel f lb (init_iter lb) in
  pfp_narrow f lb narrowing_steps fp.
```

The proof of the decreasing phase is not trivial. Indeed, the transfer function of the loop body is not always increasing[17]. For this reason, the abstract state returned by `pfp_narrow` may very well not be a stable invariant, even if it holds at every iteration. Thus, we use the proof scheme described in Section 4.2.4: the disjunction of the loop precondition and of the abstract value returned by `pfp_widen` is an invariant, and, *a fortiori* an unstable invariant. It is then easy to prove by induction on the decreasing steps that `pfp_narrow` also returns an unstable invariant.

4.4. Related Work and Perspectives

4.4.1. Comparison with Related Work

The fact that operational semantics are not adapted for reasoning on the soundness of abstract interpreters is not new. This observation has already been made by Cousot and Cousot at the very beginning of the theory of abstract interpretation [CC77]. In this early study, Cousot and Cousot described a "static semantics" for their programs, which is nowadays usually referred as a *collecting semantics*. The idea behind collecting semantics is to describe the semantics of the program by giving, for each control point, the set of states in which the machine can be when reaching this control point. The result of the abstract interpretation of the program therefore associates each control point to an abstract state approximating the concrete states reached at this control point.

Our proof technique for our abstract interpreter has two major differences with the Cousots' initial approach. First, in our axiomatic semantic, instead of speaking of sets of concrete states, we manipulate logical predicates (i.e., pre- and post-conditions) that approximate these sets. Second, our source language does not provide a notion of control point to associate with concrete or abstract states, so we have defined our program logic by following the syntax of the language.

Bertot [Ber09] already used an axiomatic semantics based on logical predicates for proving the soundness of a static analyzer using abstract interpretation. In this work, Bertot defined a toy static analyzer for a very simple language featuring while loops and variable

[17] Neither with respect to $\sqsubseteq$ nor with respect to the inclusion of concretizations.

assignments with pure arithmetical expressions over unbounded integers and Booleans. This analyzer returns an annotated version of the input program. The annotations are assertions inferred by the static analyzer, that are guaranteed to hold during the execution of the program.

Other axiomatic semantics have already been defined for C-like languages with complex control flow. Our axiomatic semantics is greatly inspired from these works [Kre15, App14]. However, these program logics are based on separation logic in order to ease the direct proof of programs. We do not use such separation features in our program logic. This simplifies greatly our soundness proof, and lets us prove our conjunction rule, which is essential for proving the soundness of decreasing iterations. The conjunction rule is known to be unsound in separation logic when stated naively [O'H04, Section 7].

4.4.2. Possible Extensions to the Interpreter

Enriching the Result of the Analysis

A possible extension to the abstract interpreter of Verasco would be to include in the result of the analysis, more properties inferred by the analyzer on program states instead of just proving the absence of undefined behaviors. These properties could be used by other static analysis tools, by the compiler optimizer or even to help the user debug the imprecision of analyses.

One important problem is the specification of this new feature: it is really unclear how to specify assertions on intermediate steps of computation of the program, when these steps do not produce any interaction in the execution trace. A solution to this problem, inspired by Bertot [Ber09] would be to produce, as the result of the analysis, an annotated version of the program, with assertions that are guaranteed to hold during executions of the program. Another solution to the specification problem would be to let the user annotate the input program at strategic places where she would need information. These annotations would produce an event in the execution trace, and the final theorem of the analyzer could relate the result of the analyzer with these events. They could be either manually inserted by the user or generated by a preliminary pass.

Improving the Precision Through Trace Partitioning

Trace partitioning [RM07] is a technique used in static analyzers to improve their precision by partitioning the set of concrete states reached at some program point depending on the path that led to the given state. For example, an analyzer can choose to store two abstract environments at a program point following the merge point of an `if` statement: the first approximates the set of states reached by taking the first branch and the second approximates the set of states reached by taking the second branch.

The implementation of this technique in Verasco would bring more precision in the analysis. However, it would either require user annotations or carefully tuned heuristics in order to chose the partitioning strategy.

Caching Loop Invariants

When analyzing nested loops, the invariant inferred for the inner loop has to be recomputed for each iteration of the analysis of the outer loop. This can lead to exponential behavior of the analyzer in case of large nesting depth. To overcome this problem, Astrée caches the invariant inferred for the inner loops in order to reuse them during subsequent iterations of

outer loops [CCF+09, Section 5.1]. Implementing this strategy could improve the performance of Verasco but it would require to reimplement the iterator by using a state monad keeping track of the inferred invariants.

III

The Numerical Backend of Verasco

Chapter 5

Handling Machine Arithmetic

The root of the hierarchy of numerical abstract domains in Verasco handles machine arithmetic. Indeed, no arithmetic operation in C compute on ideal mathematical numbers. That is, integer arithmetic is performed modulo 2^{32} or 2^{64}, and real arithmetic is estimated using single or double precision floating-point numbers.

In Verasco, both machine integer and floating-point arithmetic are handled in a sound and relatively precise manner. However, this precision does not complicate much the numerical abstract domain, because all these arithmetic constructs are encoded into simpler constructs: ideal integer arithmetic and double precision floating-point arithmetic. This encoding is described by the machine arithmetic functor, which transforms an ideal numerical abstract domain (actually a combination of such domains) into a machine arithmetic abstract domain.

One difficulty in the design of this abstract domain is that C#minor does not distinguish signed integers and unsigned integers. That is, many operations (such as addition, subtraction, multiplication, many bitwise operations...) operate on bit vectors in a signedness-unaware fashion.

This problem has already been encountered in other analyzers. Several approaches are known: using wrapped intervals [NSSS12], or the reduced product of two intervals of $\mathbb{Z}$, tracking signed and unsigned interpretations respectively [BLMP13]. However, these approaches do not extend well to the relational domains used in Verasco. Instead, Verasco provides a mechanism able to handle machine arithmetic in a signedness agnostic way using any kind of potentially relational numerical abstract domain.

We have chosen not to use directly the arithmetic over reals at this level, for several reasons: first, the simplification would have been mitigated by the fact that floating-point values can be infinites or NaNs, which have no equivalent in $\mathbb{R}$, so a type like $\mathbb{R} \cup \{+\infty, -\infty, \texttt{NaN}\}$ would have been needed; second, the implementation of the interval abstract domain over reals (but using floating-point numbers) would be very similar (but slightly less precise, given that we would have to use rounding towards $+\infty$). Thus, instead, we simply merge the single precision arithmetic and the double precision arithmetic by using the innocuousness of double rounding in these cases [Fig95, Rou14].

Moreover, in order to simplify further the ideal numerical domain interface, our machine arithmetic functor also issues appropriate queries to the ideal numerical domain to check for arithmetic errors. These arithmetic errors are division errors (i.e., when dividing by 0 or when dividing `min_int` by -1), shift overflows, and overflows when converting from

floating-point types to integer types.

The mode of operation of this functor is the following: it converts each query using machine arithmetic to an equivalent query that uses ideal arithmetic. In order to check for errors and handle numerical overflow, it does additional queries on sub-expressions.

5.1. Relating Ideal Environments and Machine Environments

The first step to understanding our abstract domain functor is to see how ideal values and environments are related to machine ones. Our approach is basic: an N-bit machine integer value is related to every ideal integer value that shares its N low bits. Said differently, an ideal integer value is related to its representative modulo 2^{32} or 2^{64}. Double precision floating-point numbers are related to themselves, and single precision floating-point numbers are related to their double-precision expansion[1]. This is expressed using the following Coq inductive predicate:

```
Inductive related : mach_num -> ideal_num -> Prop :=
| rel_int : ∀ x, related (MNint (Int.repr x)) (INz x)
| rel_long : ∀ x, related (MNint64 (Int64.repr x)) (INz x)
| rel_float : ∀ x, related (MNfloat x) (INf x)
| rel_single : ∀ x, related (MNsingle x) (INf (Float.of_single x)).
```

which relates machine numerical values (Section 3.2.2) to ideal numerical values (Section 3.2.3). This relation naturally extends to environments by stating that values mapped to a variable are related.

Using this relation, we can easily extend a concretization function for an ideal numerical abstract domain to machine environments. Assume `abt` is a type for which we have a type class instance of `gamma_op abt (var -> mach_num)` (that is, a concretization function to ideal numerical environments). Then, we can define an instance for machine environments using the same abstract values[2]:

```
Instance gamma_mach : gamma_op abt (var -> mach_num) :=
  fun (v:abt) (ρ:var -> mach_num) =>
    ∃ ρ':var -> ideal_num,
      ρ' ∈ γ v /\ ∀ id, related (ρ id) (ρ' id).
```

This choice of relation between machine and ideal values has two interesting properties. First, it does not rely on signedness. That is, both signed and unsigned interpretations of a machine integer are related to this machine integer; we do not need to choose a signedness for expressions a priori. Second, many arithmetic operations are compatible with this relation. That is, for many operators $*_m$ of integer machine arithmetic, there exists a corresponding operator $*_i$ of ideal integer arithmetic such that if a_m and b_m are machine values related to a_i and b_i, respectively, then the result of $a_m *_m b_m$ (if it exists) is related to $a_i *_i b_i$. This is the case of addition, subtraction, negation, multiplication, bitwise and, bitwise or, logical left shift (but only for the left operand), etc. For this reason, there is no need to check

[1] Every single precision floating-point number has an exact representation as a double precision floating-point number.

[2] We actually use a slightly different definition in the Coq development: indeed, as we will see in Section 5.5, we implemented and disabled an optimization consisting in using intermediate variables for sub-expressions that are analyzed several times. This complicates some parts of the implementation, but does not change the general idea: we will omit this aspect in the current description. As an example, the type of variables used in ideal environments is different from the one used in machine environments: it can either be a program variable or an intermediate variable, but we ignore this fact.

for overflows for operands of such operators. In case of overflow, the correct wraparound behavior is modeled.

5.2. Translating Expressions

The conversion of machine expressions to ideal expressions is done in a function that follows the structure of the expression:

```
Fixpoint convert_expr (v:abt) (e:mexpr) : (iexpr * bool) :=
  match e with
  [...]
  end.
```

This function takes as parameter an abstract environment and a machine expression, and returns an ideal expression, with a Boolean indicating whether the given expression may trigger an arithmetic error. Its specification is decomposed into two parts. The first part concerns the evaluation of expressions:

```
Lemma convert_expr_eval:
  ∀ ab ρ ρ', (∀ id, related (ρ id) (ρ' id)) ->
             ρ' ∈ γ ab ->
  ∀ e e' nerr', convert_expr ab e = (e', nerr') ->
  ∀ x, eval_mexpr ρ e x ->
  ∃ x', eval_iexpr ρ' e' x' /\ related x x'.
```

That is, consider `ρ` and `ρ'` two related environment such as `ρ'` (the ideal environment) is in the concretization of a given abstract environment `ab`. If `convert_expr` returns an ideal expression `e'` when given a machine expression `e` and the abstract environment `ab`, then the values of `e` in `ρ` are related to values of `e'` in `ρ'`.

The second part of the specification ensures the soundness of the arithmetic error checks:

```
Lemma convert_expr_noerror:
  ∀ ab ρ' ρ , (∀ id, related (ρ id) (ρ' id)) ->
              ρ' ∈ γ ab ->
  ∀ e e', convert_expr ab e = (e', true) ->
          ¬error_mexpr ρ e.
```

That is, consider `ρ`, `ρ'` and `ab` as previously[3]. If `convert_expr` returns `true` on its second component when given an expression `e`, then `e` cannot have an arithmetic error when evaluated in `ρ`.

5.2.1. Leaves: Variables, Literals and Non-Deterministic Atoms

Variables are translated into the same variable, converting the type tag to its ideal counterpart. Doing so is always correct, since variables evaluate to related values when evaluated in related environments. Moreover, no error can be generated by variables.

```
Fixpoint convert_expr (v:abt) (e:mexpr) : (iexpr * bool) :=
  match e with
  | MEvar (MNTint | MNTint64) x => (IEvar INTz x, true)
  | MEvar (MNTfloat | MNTsingle) x => (IEvar INTf x, true)
  [...]
  end.
```

[3] Here, this is equivalent to say that `ρ` is in the concretization of `ab`, because `ρ'` does not appears in the rest of the statement of the lemma.

Literals of machine expressions are translated into literals of ideal expressions, by choosing the signed interpretation for integers:

```
Fixpoint convert_expr (v:abt) (e:mexpr) : (iexpr * bool) :=
  match e with
  [...]
  | MEconst (MNint i) => (IEconst (INz (Int.signed i)), true)
  | MEconst (MNint64 i) => (IEconst (INz (Int64.signed i)), true)
  | MEconst (MNfloat f) => (IEconst (INf f), true)
  | MEconst (MNsingle f) => (IEconst (INf (Float.of_single f)), true)
  [...]
  end.
```

The choice of the signed interpretation is heuristic: we could have chosen, for example, the unsigned interpretation or any other integer modulo 2^N. This is motivated by the fact that the signed interpretation is the most likely to be meant by the author of the analyzed code: indeed, if she means an unsigned literal, it is likely to be smaller than 2^{N-1}, so both interpretations coincide (large literal are rare in real code). In any case, if this is the wrong interpretation, the correct modeling of wraparound on overflow is likely to recover the loss of precision. If it does not, the analysis is sound but imprecise.

5.2.2. Operators Preserving Classes Modulo 2^N

We have already mentioned that many operators do not need any check for overflow, because the corresponding ideal operator behaves correctly whatever the related ideal integer that serves as operand. This includes negation, addition, subtraction, multiplication, bitwise negation, bitwise or, bitwise and, and conversion from 64-bit to 32-bit integers. None of them can generate errors. They are all translated directly to their equivalent in ideal arithmetic[4].

As an example, the translation of 32-bit addition is defined by:

```
Fixpoint convert_expr (v:abt) (e:mexpr) : (iexpr * bool) :=
  match e with
  [...]
  | MEbinop op e1 e2 =>
    let '(e1, nerr1) := convert_expr v e1 in
    let '(e2, nerr2) := convert_expr v e2 in
    match op with
    [...]
    | Oadd => (IEbinop IOadd e1 e2, (nerr1 && nerr2)%bool)
    [...]
    end
  [...]
  end
```

5.2.3. Other Integer Arithmetic Operators

For the other integer arithmetic operations (including comparisons, division, modulo, bitwise shift, in signed and unsigned versions), it is necessary to make sure their operands do not overflow. It appears that all these operators do specify a signedness for their operands. Thus, we have two dedicated functions, `normalize_expr_signedness` (for 32-bit integers) and `normalize_expr_signedness64` (for 64-bit integers) that query an interval for a given expression, and, if necessary, add a multiple of the modulus to the expression to make

[4]Bitwise operators in ideal integer arithmetic are defined on the infinite binary stream representation of integers, using 2's complement negation.

sure it fits the range of the given signedness. If that is not possible (e.g., when the size of the interval returned by the numerical abstract domain is larger than the modulus), then it returns the whole range of possible integers for this signedness and number of bits. In any case, these functions return an expression that is a safe approximation of the given expression, modulo 2^{32} or 2^{64}, and whose domain is the given signedness range. Here is the signature and the specification lemma of `normalize_expr_signedness`:

```
Definition normalize_expr_signedness (v:abt) (e:iexpr) (s:signedness):
   iexpr :=
  [...].

Lemma normalize_expr_signedness_correct:
  ∀ v e s e', normalize_expr_signedness v e s = e' ->
  ∀ ρ, ρ ∈ γ v ->
  ∀ x, eval_iexpr ρ e (INz x) ->
  ∃ x', x' ∈ γ (match s with Signed => signed_itv
                           | Unsigned => unsigned_itv end) /\
        Int.eqm x x' /\
        eval_iexpr ρ e' (INz x').
Proof. [...] Qed.
```

The translation of such an expression node consists in first converting the operands, then applying these normalization functions, then using the corresponding ideal operator. In the case of operators able to generate errors (division, modulo and bitwise shift), checks are performed. As an example, we give here the translation for unsigned 32-bit division:

```
Fixpoint convert_expr (v:abt) (e:mexpr) : (iexpr * bool) :=
  match e with
  [...]
  | MEbinop op e1 e2 =>
    let '(e1, nerr1) := convert_expr v e1 in
    let '(e2, nerr2) := convert_expr v e2 in
    match op with
    [...]
    | Odivu =>
      let e1 := normalize_expr_signedness v e1 Unsigned in
      let e2 := normalize_expr_signedness v e2 Unsigned in
      let nerr := nerr1 && nerr2 &&
        is_bot (idnc_assume (IEbinop (IOcmp Ceq) e2 (IEconst (INz 0)))
                            true v)
      in
      (IEbinop IOdiv e1 e2, nerr)
    [...]
    end
  end.
```

5.2.4. Floating-Point Arithmetic

Double precision floating-point arithmetic is available in the ideal numerical abstract domain, so no translation is needed. For single-precision floating-point arithmetic, we use a result well-known by floating-point arithmetic experts [Fig95]. An implication of this result is that many single-precision arithmetic operations can be emulated by first computing the corresponding double precision operation and then rounding the result to single precision. This is valid for negation, absolute value[5], addition, subtraction, multiplication and division. This result has recently been formally verified in the Flocq library [Rou14].

[5] No rounding happens for negation and absolute value, so this trick is not even necessary for these operators.

These theorems help us translate single precision floating-point arithmetic into double precision floating-point arithmetic, using a specific single precision to double precision rounding operator of ideal arithmetic, called `IOsingleoffloat`. As an example, here is the code dedicated to the translation of single precision addition:

```
Fixpoint convert_expr (v:abt) (e:mexpr) : (iexpr * bool) :=
  match e with
  [...]
  | MEbinop op e1 e2 =>
    let '(e1, nerr1) := convert_expr v e1 in
    let '(e2, nerr2) := convert_expr v e2 in
    match op with
    [...]
    | Oaddfs => (IEunop IOsingleoffloat (IEbinop IOaddf e1 e2),
                 (nerr1 && nerr2)%bool)
    [...]
    end
  end.
```

5.2.5. Conversion Operators

Expressions at the machine arithmetic level contain many conversion operators. Some of them can be translated into a no-op in ideal arithmetic: this is the case for 64-bit to 32-bit integer conversion, and for double to single precision conversion. The translation of some others is just the renormalization operation we already explained for the translation of divisions: this is the case for all conversions from larger integer formats to smaller integer formats: 64-bit to 32-bit, 32-bit to 8-bit, and 32-bit to 16-bit; all of those exist in both signed and unsigned versions.

Conversions from floating-point types to integer types are translated into a dedicated operator `IOzoffloat` of ideal expressions. It is necessary to check for overflows on the result of the conversion.

After translation and renormalization of operands, conversions from integer types to floating-point types are translated into two dedicated operators over ideal expressions: `IOfloatofz` and `IOsingleofz`. These computations cannot fail, so no check is performed. It is worth noting that, when converting from 64-bit integers to a single precision floating-point number, double rounding can be harmful, thus it is necessary to distinguish single precision and double precision.

5.3. Handling Ambiguity

Perhaps one disadvantage of this choice of relation between ideal and machine values is its ambiguity. Indeed, a machine integer is related to infinitely many ideal integer values, leading, in practice, to an imprecise behavior of the analyzer. To understand why this happens, let us consider the backward analysis of a comparison. More precisely, we assume the functor is parameterized by an interval domain, and explain the behavior of the analyzer on the following code fragment:

```
int x = verasco_any_int();
if(x > 12) {
  verasco_assert(x > 12);
}
```

Before the analysis of the condition, the interval domain has no information about `x`. Thus, the conversion of the expression `x > 12` will fail, because it will not be possible to renormalize the left operand of `>`. As a result, no information will be acquired from the analysis of the condition, and the assertion check will fail.

A solution would be to assign `x` to the interval `[min_int, max_int]`, at the point of the definition of `x`, but the analysis will still fail for an unsigned comparison, because the analyzer cannot guess the signedness of a variable at the point of its definition in the C#minor code.

Our solution is different: before translating an expression, we first do a heuristic analysis of the expression in order to guess the signedness of variables it contains, based on the context of their occurrences. If we guess some new signedness information for a variable, we ask the ideal abstract domain an interval for this variable. If the answered interval is $\top$, then the variable is assigned to the range of values of its signedness.

In our example, the analysis will guess that `x` is a signed variable. Given that the interval domain knows nothing about it, the machine arithmetic functor will assign `x` the interval `[min_int, max_int]` as a preliminary step of the analysis of the condition. This analysis will then succeed in inferring the interval `[13, max_int]` for `x`.

Unfortunately, this trick is sometimes insufficient. Indeed, consider the analysis of the following piece of code:

```
int a = verasco_any_int(), b, c = verasco_any_int();
if(a > 12) {
  b = (c > 10);
} else {
  b = 0;
}
if(b) verasco_assert(c > 10);
```

This pattern is very common: it is generated for example by the front-end of CompCert when encountering the logical `&&` operator. If we used our solution as-is, the symbolic equality abstract domain we describe in Chapter 7 would succeed in recovering the condition `c > 10` at the entry of the second test. However, at this point, the interval domain has no information on `c`, leading to a the interval $[11, +\infty]$ for `c` after the analysis of the recovered condition[6], which is useless as it has no upper bound.

To avoid this kind of imprecision, we use one additional ingredient: first, we propagate the guessed signedness information with abstract environments, hence a machine abstract environment is actually a pair of an ideal abstract environment and a map of guessed signednesses. Then, when computing a join, if, for some variable, one branch has guessed more signedness information than the other, and if the interval for this variable in the other branch is $\top$, then, in this other branch, we assign this variable to the interval corresponding to the inferred signedness in the first branch.

We give here an annotated version of the previous example: for each important control points, we give the relevant information stored in the abstract domains:

```
int a = verasco_any_int(), b, c = verasco_any_int();
if(a > 12) {
  b = c > 10;
  /* a : signed  c : signed
     a ∈ [13, max_int] c ∈ [min_int, max_int]
     b == (c > 10) */
```

[6] It is important to understand that this second analysis is done at the ideal level, where it is no longer possible to arbitrarily assign the interval `[min_int, max_int]` to `c`

```
} else {
  b = 0;
  /* a : signed
     a ∈ [min_int, 12]
     b == 0
  */
}
/* Join point: c is signed in the first branch, ⊤ in the second branch.
     So, in the second branch, we assign it to [min_int, max_int].
   a : signed c : signed
   a ∈ [min_int, max_int]   c ∈ [min_int, max_int]
   b == ((a > 12) ? (c > 10) : 0)
*/
if(b) {
  /* c ∈ [11, max_int] */
  verasco_assert(c > 10);
}
```

To summarize, the following adjustments are needed:

- In the abstract states, we store additional signedness information. This information does not create any constraint on concrete states. It just provides hints used when assigning a signedness range to a variable. However, when doing so, it is needed to know variable types (e.g., for a signed variable, should we use $[-2^{31}, 2^{31} - 1]$ or $[-2^{64}, 2^{64} - 1]$?) For that reason, the functor also embeds a type abstract domain, constraining the types of values of abstracted environments.

- Before translating expressions, we do a first analysis over them in order to extract signedness and type information. If a new signedness is guessed for a variable for which the numerical abstract domains return $\top$ when queried for an interval, this variable is assigned the interval corresponding to its signedness.

- At join points (i.e., when computing a join operation), we *transfer* the signedness information before joining numerical abstract values. That is, if some signedness information is available on the left branch but not on the right branch, but both branches agree on the type information, and the numerical abstract domains returns $\top$ when queried for an interval for this variable on the right branch, then, on the right branch, this variable is assigned the interval corresponding to its signedness. The converse operation is also carried from the right branch to the left branch.

5.4. Abstract Transfer Functions and Lattice Operations

Some primitives of the machine numerical abstract domain interface (Figure 3.4) have a simple implementation: `mach_top`, `mach_leb`, `mach_widen` and `forget` are wrappers around the corresponding primitive of the ideal arithmetic abstract domain.

Primitives taking an expression as parameter (e.g., `assign`, `assume`, `get_itv`, `noerror` and `concretize_int`) translate expressions using `convert_expr` before querying the ideal abstract domain. Before this translation, the analysis described in Section 5.3 is performed. After the translation, `assign`, `assume` and `get_itv` forward the call to the ideal abstract domain.

Two queries are performed by `concretize_int`: an interval query and a congruence query. The two results are used to enumerate, if possible, the values the given expression can take.

The implementation of `noerror` uses the Boolean returned by `convert_expr` to determine, if possible, that the expression will not fail.

The `mach_join` operation is performed by first transferring signedness information, as described in Section 5.3, and then joining abstract environments.

5.5. Weaknesses and Possible Improvements

This abstract domain functor does the job: it abstracts away all issues of overflow, making it easier to design numerical abstract domains. However, several improvements are possible, improving either the precision or the efficiency.

Using temporaries for sub-expressions

The translation of an expression involves issuing several queries to the undelying numerical abstract domains. These queries ask for intervals for sub-expressions (in order to renormalize them), or check that a denominator cannot be 0. However, this potentially leads to a quadratic behavior when analyzing large expressions, because the functor will issue queries on nested sub-expressions.

To reduce the number of queries, the functor could use *phantom variables* as aliases for subexpressions: before issuing a query, it would assign a fresh phantom variable the sub-expression, perform the query on this alias, and return the alias as a translation for the expression. The translation function modifies the ideal abstract environment by assigning phantom variables; then the abstract transfer function (such as `assign` or `assume`) uses this modified ideal abstract environment, and finally issues `forget` queries on phantom variables used. Hence, the whole translation function is written in a state monad in order to thread the abstract environment and manage a counter to generate fresh phantom variables.

This work has been implemented in the current version of Verasco. However, it is currently disabled, because we observed that it actually slowed down the analysis of our examples. Further investigation is needed to understand whether and when this optimization is useful.

Less *ad-hoc* mechanism for handling ambiguity

The mechanism described in Section 5.3 seems ad-hoc. It works in our tests, but it may be imprecise on unforeseen cases. We can see two potential problems that can arise when using this method.

First, if the inferred signedness for a variable appears to be wrong when analyzing an `assume` query, for example, then the interval for this variable may not be renormalizable, and the translation fails. However, we did not encounter a realistic code where the inferred signedness would be wrong and where the interval for a variable would not be renormalizable.

Second, when we assign a variable $[-2^{N-1}, 2^{N-1} - 1]$ or $[0, 2^N - 1]$, we may lose some information. We cannot lose interval information, because we check before assigning that the known interval for a variable is $\top$. However, we may lose congruence information or relational information that may be crucial. The same problem occurs when an expression cannot be renormalized: it is then replaced with an interval, and all the congruence and relational information is lost.

We are still looking for a less ad-hoc mechanism. Perhaps a promising idea is to replace those signedness range intervals with a modulo operator: that is, when an expression e

cannot be properly renormalized, we can replace it with $e \bmod 2^N$, so that at least the congruence information is kept.

Another improvement could come from modifications of the C#minor language, together with modifications of the CompCert front-end: indeed, if C#minor variables were annotated with signedness hints inferred from C source types, the analyzer would rely much less on ad-hoc signedness inference. Similarly, Verasco would benefit from integer literals being represented in C#minor as $\mathbb{Z}$ integers as in the C source instead of 32-bit integers that do not carry signedness information: as an example, Verasco could treat differently the 32-bit integer literals `-1` and `0xFFFFFFFF`, that are semantically equivalent but are typically used in different signedness contexts.

Translating and error checking as two different processes

In the current version, the translation mechanism and the error checking mechanism are done in one single recursive function. This means that when executing the `noerror` primitive of the machine arithmetic interface, the whole expression is translated, even if it cannot syntactically raise an error. Conversely, when translating an expression for an assignment, for example, the error checking is performed even if the result is being ignored.

Translating an expression and checking for error are potentially costly operations, because they imply calling primitives of the underlying numerical abstract domain. That is why it should be possible to only translate the relevant sub-expressions when checking for errors and to skip all the error checking when the expression only uses safe constructs.

Chapter 6

Non-Relational Numerical Abstract Domains

Non-relational numerical abstract domains in Verasco include intervals over integers and floating-point numbers and arithmetical congruences. They both play an essential role: the interval abstract domain is the only one handling all the arithmetic constructs of the language, and the arithmetic congruence abstract domain is able to prove the alignment properties for array accesses, that are required in the C#minor semantics via the CompCert memory model.

Both these domains share a common part, independent of the non-relational abstract domain: indeed, a non-negligible part of the implementation involves traversing expressions and manipulating maps of abstract values, which is not specific to a particular non-relational abstract domain. This part is implemented as a functor transforming an interface for non-relational abstract domains over ideal arithmetic to the interface of possibly relational abstract domains shown on Figure 3.9. We describe this implementation in Section 6.1.

In Section 6.2, we explain some details of our implementation of the interval abstract domain; in Section 6.3, we do so for arithmetical congruences.

6.1. Common Non-Relational to Relational Functor

Non-relational abstract domains share a common interface, specific to them. Using an instance of this interface, an abstract domain functor builds a non-relational abstract domain that meets the relational interface `ab_ideal_env` of Figure 3.9. This idea was already present in unverified static analyzers such as Astrée and in previous formalizations of abstract interpretation (by Pichardie [Pic05], for example), or in the early stages of the development of Verasco [BLMP13]. The version we present here is modernized in order to adapt to some features of Verasco. As an example, it is made more flexible with respect to communication channels.

The interface provides a type of abstract values `Val`, with a concretization to concrete values (as an instance of `gamma_op Val ideal_num`). The abstract environment is then defined using functional maps from variables to such abstract values:

```
Definition t := Tree.t Val.
```

Reading such a map returns an element of `Val+T`: if reading returns `All`, this represents the absence of a mapping, i.e., no information is available for the variable, and reading returns `Just x` when the abstract value `x` is known for the variable. This helps maintaining sparse maps, where we do not store information for unconstrained variables.

6.1.1. Pointwise Operations : ⊑, ⊔ and ▽

In the non-relational numerical abstract domain interface, a comparison operator, a join operator and a widening operator are required[1]:

```
as_leb:> leb_op t
as_join:> join_op t (t+T)
widen: t -> t -> t+T
```

These operators come with their specifications[2]:

```
as_leb_correct:> leb_op_correct t ideal_num
as_join_correct:> join_op_correct t (t+T) ideal_num
widen_incr: ∀ x y, γ x ⊆ γ (widen x y)
```

They are used to build the corresponding operators for abstract environments: they are applied in a pointwise manner, using a dedicated function of the map library. In these abstract domains and for these operations, we currently ignore message channels, query channels and the possibility of returning a different abstract value for $\triangledown_1$ and $\triangledown_2$ (see Section 3.1.2)[3].

For performances, it is of paramount importance that these operations be optimized to not iterate on shared portions of the environment, because they are most often used on very similar environments. The importance of this optimization has already been noticed in the design Astrée [BCC+02, CCF+09]. We will discuss how this optimization is implemented in Verasco in Section 9.1. For now, let us just precise that this requires more properties on the pointwise operators. Namely, ⊔ and `widen` have to be idempotent and ⊑ reflexive:

```
join_idem: ∀ x:t, join x x = Just x
widen_idem: ∀ x:t, widen x x = Just x
leb_refl: ∀ x:t, leb x x = true
```

6.1.2. Forward Analysis of Expressions

The `assign` primitive is implemented using forward abstract evaluation of expressions. That is, abstract values are propagated from expression leaves to roots, in a bottom-up manner. To this purpose, several primitives of non-relational abstract domains are needed for the different kinds of expression nodes.

Variables

Variables are handled by using the abstract value for the variable present in the abstract environment[4].

[1] For now, at this level, we do not use the type classes `widen_op` and `widen_op_correct` for widenings described in Section 3.1.2. There is no pointwise reduction after widening, so it does not seem necessary to make the non-relational widening return two abstract values for the moment.

[2] `widen_incr` is the Coq form of (3.7)

[3] However, note that this does not mean that both components of the result of a widening at the root of the hierarchy of numerical abstract domains will contain identical non-relational information, because the second component could have been reduced by a message coming from another domain.

[4] Actually, the implementation uses a second-order `forward_var` primitive implemented by the non-relational abstract domain, to allow, if needed, to use the query channel to refine this abstract values, and even

Constants and intervals

Constant and interval nodes are handled using primitives converting concrete values and intervals of $\mathbb{Z}$ into abstract values:

```
const: ideal_num -> Val+⊤
z_itv: zitv -> Val+⊤
const_correct: ∀ n, n ∈ γ (const n)
z_itv_correct: ∀ i, γ i ⊆ (γ (z_itv i) ∘ INz)
```

Operators

Unary and binary operators are handled using dedicated abstract transfer functions:

```
forward_unop: i_unary_operation -> Val+⊤ -> Val+⊤+⊥
forward_binop: i_binary_operation -> Val+⊤ -> Val+⊤ -> Val+⊤+⊥
```

They take as parameter the operator involved, and abstractions of the value of operands. They return an abstraction of the value of the result of the operation. Their specifications follow:

```
forward_unop_sound: ∀ op x x_ab,
  x ∈ γ x_ab ->
  eval_iunop op x ⊆ γ (forward_unop op x_ab)
forward_binop_sound: ∀ op x x_ab y y_ab,
  x ∈ γ x_ab -> y ∈ γ y_ab ->
  eval_ibinop op x y ⊆ γ (forward_binop op x_ab y_ab)
```

where `eval_iunop` and `eval_ibinop` are the concrete semantics of unary and binary operators, respectively.

Abstract evaluation of expressions

A straightforward implementation of the abstract evaluation of expressions would be defined by induction on the expression, returning the abstract value of the given expression and using at each node the corresponding primitive of the non-relational abstract domain interface.

However, as we will see, the backward analysis of expressions requires forward abstract values for many nodes[5], hence a form of memoization is preferable. Indeed, the time of the forward analysis of a sub-expression is linear over the size of the sub-expression, which leads to a total quadratic backward analysis time if no computation is shared. A solution would be to implement the SMART approach described in Section 9.4.1, but this requires hash-consing of ideal expressions, which is heavy and thus left as future work.

Instead, we decided that the forward analysis of expressions produces a *tagged expression*. Such a tagged expression shares the same structure as the corresponding expression, but embeds tags containing the abstract value of each node. The definition of the type of tagged expressions uses an inductive definition following the structure of ideal expressions, and encoding as a dependent type the constraint that it is equivalent to the untagged expression:

```
Inductive tagged_iexpr : iexpr -> Type :=
| IEvar_tag:
    Val+⊤+⊥ ->
    ∀ ty x, tagged_iexpr (IEvar ty x)
```

to analyze recursively an expression returned by the `get_eq_expr` query channel. This is provided for symmetry with the backward analysis, and is not used currently.

[5] This is necessary for children of binary operators, and it does not hurt for children of unary operators.

```
| IEconst_tag:
    Val+T+⊥ ->
    ∀ n, tagged_iexpr (IEconst n)
| IEZitv_tag:
    Val+T+⊥ ->
    ∀ i, tagged_iexpr (IEZitv i)
| IEunop_tag:
    Val+T+⊥ ->
    ∀ op {e}, tagged_iexpr e -> tagged_iexpr (IEunop op e)
| IEbinop_tag:
    Val+T+⊥ ->
    ∀ op {e1 e2}, tagged_iexpr e1 -> tagged_iexpr e2 ->
      tagged_iexpr (IEbinop op e1 e2).
```

The abstract evaluation of expressions consists in building a valid tagged expression for the given expression, by structural induction over the expression structure. The correctness of this evaluation function states that any node is tagged with an abstract value that is an abstraction of the set of values of the corresponding sub-expression.

6.1.3. Implementation of `assign`

The implementation of `assign` for non-relational abstract domains consists in first doing a forward evaluation of the right hand side and then using the root tag as the new abstract value associated with the left hand side in the abstract environment.

The second step consists in optionally computing a message to send via the message channel. (For example, the interval abstract domain being the only one handling all arithmetic constructs, it sends a message after each assignment to other domains to share the result of its computation.) Forging this message depends on the particular domain, hence a dedicated primitive of non-relational abstract domains is used for this purpose:

```
assign_msgs: var -> Val -> messages_chan
assign_msgs_sound: ∀ x v ρ,
  ρ x ∈ γ v -> ρ ∈ γ (assign_msgs x v)
```

6.1.4. Backward Analysis of expressions

When analyzing an `if` statement, we have already explained in Section 3.3.1 that a `Fact_msg` is sent to numerical abstract domains. Analyzing such a message is like executing an `assume` primitive in traditional implementations of abstract domains: as much information as possible needs to be extracted from the fact that the expression in question evaluates to the given value.

The way this is usually done is by *backward analysis* of the expression: abstract values are propagated from the root (where the information is known using the message) to the leaves (where the information can be stored in the abstract environment).

So, when analyzing backward a binary operator, an abstract value is known for the result of the operation, and we need to compute one for operands. However, for most binary operators, this process is not possible without an abstract value for the operands themselves. As an example, from the fact that $x + y \in [0, 10]$, it is not possible to infer an interval for x without any information on y.

This leads us to conclude that a forward analysis needs to be done before any backward analysis. The backward analysis itself is done by recursion over the tagged expression built by the forward analysis: additionally, it takes as parameter the abstract value known for the root of the expression, and the abstract environment being refined by the backward

analysis. It returns the refined abstract environment. All the computations are done in the +⊥ monad, to take into account the possibility of discovering a contradiction.

Backward analysis of leaves

When reaching a variable, we need to refine the abstract value present in the abstract environment using the abstract value guessed by the analysis. To this end, we require the non-relational abstract domain to have a □ operator, using the already defined type classes:

```
as_meet:> meet_op t (t+⊥)
as_meet_correct:> meet_op_correct t (t+⊥) ideal_num
```

This operator is also used during the traversal of any node, to make sure the backward pass has always at least as much information as the forward pass: when entering any node (even non-leaf), the current abstract value is refined using □ by the abstract value of the forward pass.

Apart from this refinement, that can reveal a contradiction, no operation is done for the backward analysis of constant and interval leaves.

Backward analysis of unary operators

The backward analysis of unary operators uses a dedicated primitive of the non-relational abstract domain, and then calls the backward analysis function recursively on the sub-expression. The primitive in question has the following type and specification:

```
backward_unop: i_unary_operation -> t+⊤ -> t+⊤ -> t+⊤+⊥
backward_unop_sound: ∀ op x x_ab res res_ab,
  x ∈ γ x_ab -> res ∈ γ res_ab ->
  eval_iunop op x res ->
  x ∈ γ (backward_unop op res_ab x_ab)
```

That is, it takes as parameters the considered operator, an abstract value for the result, an abstract value for the operand, and returns a possibly refined abstract value for the operand[6].

Backward analysis of binary operators

Similarly to unary operators, the backward analysis of binary operators uses a dedicated procedure of the underlying non-relational abstract domain:

```
backward_binop: i_binary_operation -> t+⊤ -> t+⊤ -> t+⊤ -> (t+⊤*t+⊤)+⊥
backward_binop_sound: ∀ op x x_ab y y_ab res res_ab,
  x ∈ γ x_ab -> y ∈ γ y_ab -> res ∈ γ res_ab ->
  eval_ibinop op x y res ->
  (x, y) ∈ γ (backward_binop op res_ab x_ab y_ab)
```

After calling this transfer function, the backward analysis does two recursive calls, one for each of the two operands. The abstract environment is threaded, so that the information collected on both branches will be available at the end of the analysis.

[6]The passed abstract value for the operand, is not necessarily taken into account, and is provided only in the case of need. Indeed, as we have already explained, the recursive call will anyway meet it with the result, to make sure no information is lost.

Customizing the processing of variables

When handling a variable during a backward analysis, we need to perform various domain-dependent actions. In particular, we may want to query various query channels in the hope of refining again the inferred abstract value. In particular, a non-relational abstract domain may decide to unfold a variable using `get_eq_expr` and analyze recursively the unfolded expression. To this end, the non-relational abstract domain has to provide a function that is called whenever encountering a variable. This function can call recursively the forward and backward analyzers, and refine the abstract value associated to the variable in question.

Sending messages after a backward analysis

After a backward analysis, the non-relational abstract domain may send messages based on the result. To this purpose, at the end of a backward analysis, the functor calls a dedicated primitive of the non-relational domain for every variable whose abstract value has been refined. This primitive can create messages containing the new information concerning the variable of the expression.

6.1.5. Receiving Messages

Some messages are treated directly by the non-relational functor: `Fact_msg` messages, that come from the analysis of `if` statements, trigger a backward analysis; the value of `Known_value_msg` messages is converted to an abstract value using the `const` primitive, and then used to refine the abstract environment.

Additionally, the non-relational abstract environment provides a `process_msg` primitive to process messages in a specialized way. This primitive is passed the forward and backward analysis functions to make it able to access the abstract environment.

6.2. Interval Abstract Domain

Intervals are one of the most commonly used abstract domains in literature. They have been introduced very early by Cousot and Cousot [CC76], and are a basis for designing an abstract domain on numerical properties of programs.

In our setting, we need an interval abstract domain able to give precise bounds to ideal integer arithmetic and IEEE 754 double precision floating-point arithmetic. The abstract domain should provide both forward and backward primitives for most of the operators of the arithmetic.

The type of intervals is defined in Coq using the following inductive types:

```
Inductive zitv :=
| ZITV: Z -> Z -> zitv.

Inductive fitv :=
| FITV: float -> float -> fitv.

Inductive iitv :=
| Az : zitv -> iitv
| Af : fitv -> iitv.
```

That is, an interval is either an integer interval or a floating-point interval. An integer interval is a pair of two elements of $\mathbb{Z}$, while a floating-point interval is a pair of two floating-point numbers. The concretization functions for the types `zitv`, `fitv` and `iitv` are as expected, taking bounds in the large sense:

```
Instance gamma_zitv : gamma_op zitv Z := fun i v =>
  let 'ZITV m M := i in
  m <= v <= M.

Instance gamma : gamma_op fitv float := fun i v =>
  let 'FITV m M := i in
  Fleb m v = true /\ Fleb v M = true.

Instance itv_num_gamma : gamma_op iitv ideal_num := fun v x =>
  match v, x with
  | Az v, INz x => x ∈ γ v
  | Af v, INf x => x ∈ γ v
  | _, _ => False
  end.
```

We informally maintain the invariant that an interval is never empty: that is, upper bounds are always larger than lower bounds, and floating-point bounds cannot be `NaN`. The floating-point bounds of a floating-point interval need not be finite: a floating-point interval can contain IEEE 754 infinities. Depending on the context, we can add top or bottom elements using the `itv+T`, `itv+⊥` and `itv+T+⊥` types.

Note that there is no interval of all values for integers nor for floating-point numbers: the bounds of integer intervals are necessarily finite, and `NaN` values are not in any floating-point interval. This is partly justified by the fact that types are always known from the context, so we do not lose precision by using a common $\top$ abstract value. It is also worth noting it is unnecessary to represent semi-bounded integer intervals, because they do not correspond to any meaningful property on machine integers.

The definition of the `leb`, `join` and `meet` functions are very standard, and need not be detailed here. As for the widening, we use the widening to $\pm\infty$ for floating-point intervals, and the widening with thresholds [CCF+09] for integer intervals. The list of thresholds is extendable, but, for now, we use the bounds of the CompCert C integer types. Those widening strategies can be improved, both for precision and performance: we leave as future work the experimentation needed to find a good compromise.

We have defined the abstract transfer functions of the different operators using the appropriate abstract types, and then defined the "untyped" abstract transfer functions by projecting `iitv` abstract values to `zitv+⊥` or `fitv+⊥`, calling the typed transfer function and then using either `Az` or `Af`.

6.2.1. Integer Arithmetic

Transfer functions for arithmetic operators on integer intervals are fairly standard and well-known [Min04, Pic05]. Without entering into the details, we give here a brief summary.

Concerning addition, subtraction, negation and multiplication, we have the following properties, assuming $x \in [x^-;\ x^+]$, $y \in [y^-;\ y^+]$ and $r \in [r^-;\ r^+]$:

$$x + y \in [x^- + y^-;\ x^+ + y^+] \tag{6.1}$$

$$-x \in [-x^+;\ -x^-] \tag{6.2}$$

$$x - y \in [x^- - y^+;\ x^+ - y^-] \tag{6.3}$$

$$xy \in \begin{array}{l}[\min\{x^-y^-, x^-y^+, x^+y^-, x^+y^+\};\\ \qquad\qquad \max\{x^-y^-, x^-y^+, x^+y^-, x^+y^+\}]\end{array} \tag{6.4}$$

$$r = x + y \ \Rightarrow\ x \in [r^- - y^+;\ r^+ - y^-] \tag{6.5}$$

$$r = x + y \ \Rightarrow\ y \in [r^- - x^+;\ r^+ - x^-] \tag{6.6}$$

$$r = -x \;\Rightarrow\; x \in [-r^+;\; -r^-] \tag{6.7}$$
$$r = x - y \;\Rightarrow\; x \in [r^- + y^-;\; r^+ + y^+] \tag{6.8}$$
$$r = x - y \;\Rightarrow\; y \in [x^- - r^+;\; x^+ - r^-] \tag{6.9}$$

We can easily deduce forward abstract transfer functions for addition, negation, subtraction and multiplication, together with backward transfer functions for addition, negation and subtraction.

Concerning the forward abstract transfer function for multiplication, a special case is handled when no information is known for one multiplicand but 0 is the only possibility for the other multiplicand, in which case the resulting interval is the singleton 0.

Integer division

We do not implement currently a backward transfer function for division[7]. For the forward transfer function[8], we first consider the case where $0 < y^- \leq y \leq y^+$. In this case we have:

$$x \div y \in \begin{bmatrix} x^- \div y^- \text{ if } x^- < 0 & ; & x^+ \div y^+ \text{ if } x^+ < 0 \\ x^- \div y^+ \text{ otherwise} & & x^+ \div y^- \text{ otherwise} \end{bmatrix} \tag{6.10}$$

The case where $y^- \leq y \leq y^+ < 0$ is treated symmetrically, by using the fact that $x \div (-y) = -(x \div y)$.

The general case is handled by splitting the interval for y into three parts: the negative part, the null part (whose only element necessarily leads to an error, so the computed abstract value is $\bot$), and the positive part. The intervals for the results of each of these parts are joined, leading to the global result.

Again, special cases are treated if either operand is known to be equal to 0, so that no precision is lost even if the other interval is $\top$.

Integer modulo

We did not implement a backward transfer function for the modulo operation. As for the forward transfer function, integer modulo[9] is treated by splitting not only on the interval for y, but also on the interval for x. So, we first assume that $0 < x^- \leq x \leq x^+$ and $0 < y^- \leq y \leq y^+$. We have:

$$x \bmod y \in \begin{bmatrix} x^- \text{ if } x^+ < y^- & ; & \min\{y^+ - 1, x^+\} \\ 0 \text{ otherwise} & & \end{bmatrix} \tag{6.11}$$

This interval is not always the best one: the best transfer function would involve rather complex computations and arithmetic proofs. However, we believe it captures important properties of the modulo operation: namely, it is always included in $[0, y^+ - 1]$, and, if it is known that $x < y$, then the interval for the result is equal to that of x.

[7] Their is no theoretical reason for this limitation: it could be implement without much effort, but this requires lengthy arithmetic proofs, and this rarely improves the precision in practice, because division rarely appears in tests.

[8] The version of division we use is the one used in the C99 standard [ISO99] and in CompCert. This is the division with rounding towards 0, implemented in Coq's `Z.quot` function. We note it $\div$.

[9] The variant of the modulo we use, as in the C99 standard [ISO99] is such that $a = b(a \div b) + a \bmod b$. It is implemented in Coq using the `Z.rem` function.

The general case is treated by splitting the intervals for x and y, depending on the signs of x and y. We use the two following properties of the modulo operator of the C language:

$$x \bmod (-y) = x \bmod y \tag{6.12}$$
$$(-x) \bmod y = -(x \bmod y) \tag{6.13}$$

Backward transfer function for multiplication in $\mathbb{Z}$

Again, the backward transfer function for multiplication is built by splitting the intervals using signs. Let us consider the case where we want to compute an interval for x, given an interval $[y^-;\ y^+]$ for y and an interval $[r^-;\ r^+]$ for r. By splitting the interval for y depending on its sign, we can assume without loss of generality that $0 < y^- \leq y \leq y^+$ (the case where $y = 0$ is trivial, and the negative case is symmetric).

The following interval is used as a partial backward transfer function for multiplication when $0 < y^- \leq y \leq y^+$:

$$x \in \left[\begin{array}{l} \left\lceil \dfrac{r^-}{y^-} \right\rceil \text{ if } r^- < 0 \\ \left\lceil \dfrac{r^-}{y^+} \right\rceil \text{ otherwise} \end{array} \right. ; \left. \begin{array}{l} \left\lfloor \dfrac{r^+}{y^+} \right\rfloor \text{ if } r^+ < 0 \\ \left\lfloor \dfrac{r^+}{y^-} \right\rfloor \text{ otherwise} \end{array} \right] \tag{6.14}$$

Integer comparisons

Integer comparisons at the ideal numerical level are binary operators that return either 0 or 1. They are either a strict inequality test, a loose inequality test, an equality test or a disequality test.

The backward analysis of comparisons first reduces to the case where the result interval is $[1;\ 1]$. This can be done easily, because the set of possible comparisons is stable by negation. So, let us consider the case where we know $x \in [x^-;\ x^+]$, $y \in [y^-;\ y^+]$ and $x \diamond y$ for $\diamond \in \{<, >, \leq, \geq, =, \neq\}$. The objective is to deduce a new interval on x and y. We consider only the case of finding an interval for x (y is handled by symmetry). Moreover, recall the backward analysis engine will meet this new interval with the old interval $[x^-;\ x^+]$, so that we do not fear losing information on x. The backward transfer function is directly deduced from the following properties:

$$x = y \ \Rightarrow\ x \in [y^-;\ y^+] \tag{6.15}$$
$$x \leq y \ \Rightarrow\ x \in \begin{cases} \bot & \text{if } y^+ < x^- \\ [x^-;\ y^+] & \text{otherwise} \end{cases} \tag{6.16}$$
$$x \geq y \ \Rightarrow\ x \in \begin{cases} \bot & \text{if } x^+ < y^- \\ [y^-;\ x^+] & \text{otherwise} \end{cases} \tag{6.17}$$
$$x < y \ \Rightarrow\ x \in \begin{cases} \bot & \text{if } y^+ - 1 < x^- \\ [x^-;\ y^+ - 1] & \text{otherwise} \end{cases} \tag{6.18}$$
$$x > y \ \Rightarrow\ x \in \begin{cases} \bot & \text{if } x^+ < y^- + 1 \\ [y^- + 1;\ x^+] & \text{otherwise} \end{cases} \tag{6.19}$$

$$x \neq y \;\Rightarrow\; x \in \begin{cases} \bot & \text{if } y^- = y^+ = x^- = x^+ \\ [x^- + 1;\ x^+] & \text{if } y^- = y^+ = x^- < x^+ \\ [x^-;\ x^+ - 1] & \text{if } y^- = y^+ = x^+ > x^+ \\ \top & \text{otherwise} \end{cases} \tag{6.20}$$

The forward analysis of comparison is built using the backward analysis: to compute the result of the forward analysis of a comparison, we do two backward analyses using $[0;0]$ and $[1;1]$ as result intervals. The result of the forward analysis is then computed by joining the result intervals whose backward analysis is non-contradictory.

6.2.2. Bitwise Operators

Bitwise operators at the ideal numerical level of Verasco operate on the binary representation of integers, completed by an infinite stream of 0 for non-negatives and using 2's complement negation for negatives.

In this section, we will use the syntax of the C language as notations for bitwise operations: $\sim$ for bitwise negation, & for bitwise and, | for bitwise or, ^ for bitwise exclusive or and $\gg$ and $\ll$ for right and left shifts, respectively.

Bitwise negation

Because of the definition of 2's complement negation, the bitwise negation is easily expressed in terms of the arithmetic negation:

$$\sim x = -x - 1 \tag{6.21}$$

This leads us to an easy, precise transfer function for bitwise negation. Moreover, bitwise negation is involutive, so the same transfer function can be used as a backward transfer function.

Left and right shifts

Left shift and right shift can be interpreted as the multiplication or division by the power of 2 of the right operand, thus a precise transfer function for left and right shift is built using similar techniques as for multiplication and division[10]. However, we did not implement a backward transfer function for shifts.

Bitwise and, or, and exclusive or

Other bitwise operations are more complex to handle. We did not implement the best possible abstraction even in the forward case. Instead, we preferred a simpler forward transfer function, that is still relatively precise: its precision is enough to propagate constants. We did not implement a backward transfer function for these operations.

We focus here on the approximation of the & operation applied to non-negative operands. The full transfer functions for & and | are deduced from this transfer function by splitting the intervals of the operands depending on their signs (like for division), treating the highest order bits separately (their value is fixed by the sign), and using the De Morgan's laws. The

[10] Special care must be taken, however, for the right shift operator, that do not use the same rounding mode.

exclusive or operation is (imprecisely) handled by the formula:

$$x \wedge y = (x \,\&\, {\sim} y) | ({\sim} x \,\&\, y) \tag{6.22}$$

The transfer function for & on positive inputs $x \in [x^-;\ x^+]$ and $y \in [y^-;\ y^+]$ proceeds as follows:

- Count the number of high-order bits common to x^- and x^+. These bits determine the high-order bits of x.
- Do similarly for y.
- Deduce the high-order bits of $x \,\&\, y$ corresponding to bits known for both x and y by the previous steps. These bits are used as the high-order bits of the lower and upper bounds for $x \,\&\, y$, so it suffices to compute an interval for the low-order bits of $x \,\&\, y$.
- A lower bound for the low-order bits of $x \,\&\, y$ is given by 0.
- The low-order bits of $x \,\&\, y$ have always fewer bits at 1 than those of x and y, so the higher bound on the low-order bits of x and y can both be taken as a high bound for $x \,\&\, y$: we take the smallest one.

6.2.3. Forward Transfer Functions for Floats

In CompCert, the semantics of floating-point computation is based on the Flocq library by Boldo and Melquiond [BM11, BJLM13, BJLM15]. This is a rigorous formalization, using the Coq proof assistant of IEEE 754 floating-point arithmetic. As a restriction to C99, the semantics of CompCert only supports the round-to-nearest mode for all the operations: we stick to this choice and assume the given program only uses round-to-nearest mode.

Arithmetic operators

In the following, we note double precision floating-point arithmetic operations with round-to-nearest mode, using the corresponding operator, with a circle around (e.g., floating-point addition is noted $\oplus$).

Even if the IEEE 754 arithmetic operations have complex and non-intuitive properties in general, they obey similar monotonicity properties as those of $\mathbb{R}$. That is, if we check that the input intervals guarantee that the result is not a `NaN` (in which case we have no other choice than returning $\top$), then we can use the intuitive transfer function that could be used in $\mathbb{R}$. That is, under the assumptions $x \in [x^-;\ x^+]$ and $y \in [y^-;\ y^+]$, we have:

$$\texttt{notNan}(x \oplus y) \;\Rightarrow\; x \oplus y \in [x^- \oplus y^-;\ x^+ \oplus y^+] \tag{6.23}$$

$$\ominus x \in [\ominus x^+;\ \ominus x^-] \tag{6.24}$$

$$\texttt{notNan}(x \ominus y) \;\Rightarrow\; x \ominus y \in [x^- \ominus y^+;\ x^+ \ominus y^-] \tag{6.25}$$

$$\texttt{notNan}(x \otimes y) \;\Rightarrow\; x \otimes y \in \begin{array}{l}[\min\{x^- \otimes y^-, x^- \otimes y^+, x^+ \otimes y^-, x^+ \otimes y^+\};\\ \quad \max\{x^- \otimes y^-, x^- \otimes y^+, x^+ \otimes y^-, x^+ \otimes y^+\}]\end{array} \tag{6.26}$$

$$\texttt{notNan}(x \oslash y) \;\Rightarrow\; x \oslash y \in \begin{cases} [-\infty;\, +\infty] & \text{if } 0 \in [y^-;\, y^+] \\ \left[\min \left\{ \begin{array}{l} x^- \oslash y^-,\, x^- \oslash y^+ \\ x^+ \oslash y^-,\, x^+ \oslash y^+ \end{array} \right\} ;\right. & \text{otherwise} \\ \quad \left. \max \left\{ \begin{array}{l} x^- \oslash y^-,\, x^- \oslash y^+ \\ x^+ \oslash y^-,\, x^+ \oslash y^+ \end{array} \right\} \right] & \end{cases} \tag{6.27}$$

These properties are used directly as forward transfer functions for floating-point operations. The `notNan` checks are performed by a careful case analysis on the different possible forms of x and y.

Conversions operators

In the ideal expressions of Verasco, there are still various conversion operators: from floating-point numbers to $\mathbb{Z}$, from $\mathbb{Z}$ to double precision, from $\mathbb{Z}$ to single precision, and from double precision to single precision[11].

Once ruled out the possibility of an error during the conversion[12], it is easy to give an interval for the result, because all these conversions are increasing functions, so it suffices to compute the conversion on each interval bound (using, obviously, the rounding mode used at runtime by the program: round-to-nearest when the result is a floating-point number, round-towards-zero when it is an integer).

Comparisons

Similarly to integers, we use the backward transfer function for floating-point comparison to build the forward transfer function.

6.2.4. Backward Transfer Functions for Floating-Point Arithmetic

We have implemented and proved correct floating-point backward transfer functions for addition, subtraction, comparisons, negation and all the conversion operators. Backward analysis for multiplication and division are left as future work.

Apart from negation, whose backward transfer function is identical to the forward one (because of its involutiveness), all the backward transfer functions rely on two functions, $\texttt{next\_ulp}_d$ and $\texttt{next\_ulp}_s$. The former returns the double precision floating-point number immediately following its argument. More precisely, when given a non-`NaN` double precision floating-point number different from $+\infty$, it returns one of the smallest[13] strictly bigger double precision floating-point numbers. $\texttt{next\_ulp}_s$ is its single precision version. The symmetric operations, computing the immediately preceding floating-point number, are obtained by pre- and post-composing with floating-point negation.

[11] Every single precision floating-point number can be exactly represented as a double precision floating-point number. In order to avoid dealing with two different types of floating-point numbers at the ideal numerical level of Verasco, operators returning a single precision floating-point numerical actually return its representation as a double precision floating-point number. See Chapter 5.

[12] This can occur only when trying to convert a non-finite floating-point number to $\mathbb{Z}$, which is easily checked by checking the finiteness of the input interval bounds.

[13] There may be several ones, because -0 and $+0$ are two different floating-point numbers.

Floating-point operations and conversions

Most floating-point operations are defined as an optional mathematical operation (that can be the identity) followed by a rounding. In the case of arithmetic operations (e.g., addition and subtraction), the double precision round-to-nearest rounding is used. In the case of conversions, depending on the case, either the single or double precision round-to-nearest rounding or the integer round-to-zero rounding is used.

Thus, our strategy for the backward analysis of floating-point operations proceeds in two steps: first, we do a backward analysis of the rounding, and then we do a backward analysis of the mathematical operation.

The interval used between the two steps is an abstraction of the theoretical result of the mathematical operation in $\mathbb{R} \cup \{-\infty; +\infty\}$. The first step does as follows, depending on the rounding involved (we assume the result is $r \in [r^-;\ r^+]$):

- $[-\texttt{next\_ulp}_d(-r^-);\ \texttt{next\_ulp}_d(r^+)]$ for double-precision round-to-nearest;
- $[-\texttt{next\_ulp}_s(-r^-);\ \texttt{next\_ulp}_s(r^+)]$ for single-precision round-to-nearest;
- $\begin{bmatrix} r^- \text{ if } r^- > 0 & r^+ \text{ if } r^+ < 0 \\ r^- - 1 \text{ otherwise} & r^+ + 1 \text{ otherwise} \end{bmatrix}$ for integer round-to-0.

The second step consists in doing a backward analysis of the mathematical operation. For addition and subtraction, this is done similarly to integer intervals. For conversions, the mathematical operation is simply the identity, whose backward analysis is trivial. Note, however, that we know that the operands are floating-point numbers (or an integer, for the conversions from $\mathbb{Z}$), thus the computed bounds can be rounded towards the *interior* of the interval.

Comparisons

The backward analysis of floating-point comparisons is done in a very similar way as for integers, therefore we will not give much detail here. However, it is worth noting two differences. First, it is incorrect to say that the set of floating-point comparisons is stable by negation, because of `NaN`s, which do not compare with any floating-point number, even themselves[14]. Therefore, it is necessary to consider separately each pair of a comparison mode and a Boolean result. Second, for strict comparisons and disequality, the incrementing and decrementing of the bounds are replaced by proper uses of $\texttt{next\_ulp}_d$.

6.2.5. Cooperation with Other Domains

The interval abstract domain cooperates using message and query channels with the rest of the hierarchy. Those cooperations are of several kinds: the abstract domain is able to send information to other domains, but also to use some information given by other domains.

Providing information to other domains

After a backward analysis or after processing an assignment, the interval abstract domain sends a `Itv_msg` message for each variable whose abstract value has changed. There is a special case when the new interval is a singleton, in which case, instead, it broadcasts a `Known_value_msg` message. The interval abstract domain is the only one being able to handle

[14] Regarding `NaN`s, floating-point numbers have strange properties: for example, we do not have the property $\forall x, x =_f x$, where $=_f$ is the floating-point equality test. This property is wrong if x is a `NaN`.

all arithmetic constructs: these pieces of information are useful to, e.g., the congruence abstract domain and the octagon abstract domains, when the use of unsupported constructs leads to imprecision.

The interval abstract domain is also able to answer `get_itv` queries: they are processed by a forward analysis of the given expression. This query channel has several uses: it is used, for example, by the analyzer front-end to enumerate the values an expression can take, by the machine arithmetic functor to check for overflows, and by the expression linearization to give bounds to factors of a multiplication.

Refining intervals using the congruence information

The interval abstract domains cooperates with the congruence abstract domain to build a weak reduced product: each time a new interval is learned for a variable by a backward analysis, the interval abstract domain asks the `get_congr` channel and uses the following property to refine this interval:

$$\begin{cases} x \in [a, b] \\ x \bmod m = r \end{cases} \quad \Rightarrow \quad c \in [a + (r - a) \bmod m;\ b - (b - r) \bmod m] \tag{6.28}$$

Additionally, this property is used when receiving a `Congr_msg` message.

Unfolding variables

When doing a backward analysis, some more information may be recovered by unfolding variables using the `get_eq_expr` channel (answered by the domain of symbolic equalities). When reaching a variable during the backward analysis of an expression, the interval abstract domain tries to unfold this variable and do a backward analysis of the unfolded expression. This process is continued until a predefined depth.

Receiving `Itv_msg` messages

Other domains, like the octagon abstract domain, can send `Itv_msg` messages. These messages can contain information that the interval abstract domain does not have, because of its non-relational nature, for example. These messages are handled by the interval abstract domain, and help refine the abstract values it contains.

6.3. Arithmetical Congruences

The other non-relational abstract domain formally verified in Verasco is the domain of congruences. The idea of a congruence abstract domain is not new: it has been first proposed by Granger in 1989 [Gra89]. This domain is essential in Verasco, because memory accesses in the CompCert memory model need to be aligned, so the memory abstract domain has to check alignment constraints. In Verasco, this domain has been mainly contributed by Vincent Laporte: for the sake of completeness, we give here a rough description.

The principle is as follows: in an abstract value, we store two integers $M \in \mathbb{N}$ and $R \in \mathbb{Z}$ such that if $M > 0$, then $0 \leq R < M$. The set of integer values x represented by such a pair of integers is such that $x \equiv R \bmod M$, or, equivalently:

$$\exists k,\ x = R + kM \tag{6.29}$$

The case $M = 0$ corresponds to a singleton concretization ($x = R$), and the case $M = 1$ correspond to $\top$[15].

In Coq, a congruence abstract value is described by the following record type:

```
Record zcongr := ZC {
  zc_rem : Z;
  zc_mod : N;
  zc_rem_itv : (zc_mod > 0)%N -> 0 <= zc_rem < zc_mod
}.
```

We can easily give it a concretization function in $\mathbb{Z}$:

```
Instance zc_gamma : gamma_op zcongr Z := fun ab z =>
  ∃ q, z = ab.(zc_rem) + q * ab.(zc_mod)
```

It can be easily extended to any ideal number, by stating that such an abstract value never concretizes to floating-point numbers:

```
Instance zc_ideal_gamma : gamma_op zcongr ideal_num := fun ab x =>
  match x with
  | INz z => z ∈ γ ab
  | INf _ => False
  end.
```

That is, in the abstract environment (that contains abstract values of type `zcongr+⊤`) floating-point variables are always associated to `All` (i.e., the $\top$ element of `zcongr+⊤`).

We will not enter much more into the details of the implementation of this abstract domain, and refer the interested reader to the implementation. It contains the required lattice operations: `join`, `meet` and `leb`. The domain does not contain infinite increasing chains, therefore we can use `join` as a widening. It also contains abstract transfer functions for most of the basic arithmetic operators.

Finally, this domain is able to cooperate with other domains: additionally to the cooperation features provided by the non-relational abstract domain functor, it answers `get_congr` queries, and sends `Congr_msg` messages when a new piece of congruence information is learned on a variable (either by assignment or after a backward analysis).

[15] In this case, we have necessarily $R = 0$, so we have a unique top abstract value.

Chapter 7

Symbolic Equalities

Verasco contains an abstract domain of symbolic equalities. This domain improves the precision of many other abstract domains: this technique is a well-known technique to improve the precision of static analyses. It has already been implemented in the context of Astrée [Min04, Min06b], in conjunction with linearization techniques similar to those presented in Section 8.1. The static analyzer Clousot, used to analyze .Net code, also uses some form of symbolic equalities to recover the loss of precision resulting from the analysis of pre-compiled code [LF08a].

In the context of Verasco a pre-compilation phase, done by CompCert's front-end pulls side effects out of expressions. Even if this is desirable for Verasco, this program transformation can hurt precision. An important case is the analysis of Boolean operators: because these operators are lazy in C[1], they carry a form of side effect, which is not allowed in C#minor expression. As a result, the compiler front-end will remove uses of such operators, using `if` statements and temporary variables. As an example, consider the following C code fragment:

```
if(a > 0 && b > 0) { [...] }
```

The CompCert front-end will transform it into a C#minor equivalent of the following:

```
if(a > 0) tmp = b > 0;
else tmp = 0;
if(tmp) { [...] }
```

which is difficult to analyze precisely as-is: without unfolding variables, the backward analysis of intervals presented in Section 6.1.4 will not be able to deduce anything more precise than $\texttt{tmp} \in [1; 1]$ at the entry of the body of the `if` statement (which is useless, because `tmp` is not used anywhere else). Instead, at this point, we expect the analyzer to refine the abstract values of `a` and `b`, to reflect the fact that $\texttt{a} > 0$ and $\texttt{b} > 0$. This is, in fact, very similar to the situation of Clousot [LF08a]: this analyzer uses .Net bytecode as its input language. There is no structured expression in .Net bytecode, therefore the contents of tests have to be decompiled by Clousot.

[1] That is, the evaluation of `a && b` will not trigger the evaluation of `b` if `a` evaluates to `0`; and the evaluation of `a || b` will not do so for `b` if `a` evaluates to a non-zero integer.

Even if the code does not use Boolean operators, an abstract domain of symbolic equalities can still be useful: a common case is when the programmer uses temporary variables. As an example, consider the following piece of C code:

```
int b = verasco_any_int();
verasco_assume(0 < b && b < 1000);
int a = b*2;
if(a < 10) {
  verasco_assert(b < 5);
}
```

Here, some information learned on variable `a` (`a` < 10) leads to some new information on `b` (`b` < 10). However, the backward analysis of the test only does not bring any new information on `b`. To do so, some kind of relational information between `a` and `b` is needed. A weakly relational abstract domain, like octagons (see Section 8.3) is of no help here, because the linear relationship between `a` and `b` is not necessarily in the form the weakly relational supports. A solution would be to use a polyhedron abstract domain, but it has a high computational cost. Instead, an abstract domain of symbolic equalities stores the equality `a` $= 2$`b`, which is unfolded when analyzing the condition.

In the context of Astrée, Miné [Min06b] explains that some kind of symbolic equalities can be used when linearizing equations in order to improve precision. This feature is not currently implemented in Verasco, yet we do not foresee any serious difficulty.

7.1. Abstract Environments and their Concretization

Our abstract domain of symbolic equalities stores two kinds of information. First, it stores equalities between a variable and an expression. These equalities are unfolded by the backward analysis of expressions to improve precision. In order to get good precision for the analysis of Boolean operators, the type `ideal_expr` of ideal expressions is not enough, because it does not include any Boolean construct. We use a different type, `ite_iexpr`, adding if-then-else selectors in prenex position:

```
Inductive ite_iexpr : Type :=
| Leaf : iexpr -> ite_iexpr
| Ite : iexpr -> ite_iexpr -> ite_iexpr -> ite_iexpr.
```

Their semantics is defined in a big-step style, by the `eval_ite_iexpr` inductive predicate, similarly to `iexpr` expressions: `Leaf` nodes evaluate as their child, and `Ite` nodes evaluate like their second or third child, depending on whether their first child evaluates to `INZ 0` or `INZ 1`.

This new expression type makes us able to associate decision trees to variables, and, in particular, to represent Boolean operators. Note, however, that the decision trees we store correspond to a reality in the program, so that in practice, we avoid the exponential blowup that typically goes with decision trees.

The second kind of information stored in this abstract domains is *symbolic facts*. As we will see, they are used as conditions to build if-then-else selectors when reaching a join point in the program. They associate an expression to a Boolean, and they represent the fact that the expression evaluates to either `INZ 0` or `INZ 1`.

7.1.1. Data Structures

In order to have a fast lookup mechanism, the set of symbolic facts is indexed by expressions. This efficient map is built by computing a hash of every inserted expression, and using hashes as indexes of an efficient map implementation over integers (we use CompCert's tries).

When performing `assign` and `forget` operations on an abstract environment, we need to find all the equalities and facts depending on the modified variable. In order to do that without traversing the whole data structures, we maintain other structures able to compute quickly this information.

In the end, the abstract domain uses several types of efficient tree-like data structures:

- `MVar.t τ` is the type of maps indexed by variables and containing values of type `τ`;
- `SVar.t` is the type of sets of variables;
- `MExp.t τ` is the type of maps indexed by expressions and containing values of type `τ`;
- `SExp.t` is the type of sets of expressions.

These basic data structures are the basic blocks needed to define the type of raw abstract environments of the symbolic domain:

```
Record t0 := {
  eqs : MVar.t ite_iexpr;
  eqs_free : MVar.t SVar.t;
  facts : MExp.t bool;
  facts_free : MVar.t SExp.t
}.
```

It has four fields: `eqs` contains the equalities between a variable and an expression, and `facts` contains the expressions whose value is known to be `INZ 0` (i.e., the associated Boolean value is `false`) or `INZ 1` (i.e., the associated Boolean value is `true`). The two other fields, `eqs_free` and `facts_free` associate each variable to the set of keys of entries of `eqs` and `facts` where the variable appears.

These data structures have an invariant, stating that the `eqs_free` and `facts_free` maps are valid. The invariant has two parts: first, the stored sets contain enough keys (we say they are *correct*), second, they do not contain useless keys (we say they are *complete*). The completeness property is not useful for the soundness of the abstract domain, so we only maintain it informally, while the correctness property is formalized using the following Coq predicate, stated using a record type:

```
Record t0_inv (ab:t0) : Prop := {
  eqs_free_correct :
    ∀ x e, MVar.get x ab.(eqs) = Some e ->
    ∀ y, SVar.mem y (ite_iexpr_free_vars e) ->
    ∃ s, MVar.get y ab.(eqs_free) = Some s
      /\ SVar.mem x s;
  facts_free_correct :
    ∀ e b, MExp.get e ab.(facts) = Some b ->
    ∀ y, SVar.mem y (iexpr_free_vars e) ->
    ∃ s, MVar.get y ab.(facts_free) = Some s
      /\ SExp.mem e s
}.
```

That is, if a stored equality or fact has a free variable, then there is a corresponding entry in `eqs_free` or `facts_free`. This invariant uses two functions `iexpr_free_vars` and `ite_iexpr_free_vars`, computing the set of free variables of an expression. It also uses

the `*.mem` and `*.get` primitives of the sets and maps libraries, allowing to read a map and stating membership of a set, respectively.

We then define the type of abstract environment using a subset type, containing the data structure and the proof this data structure meets the invariant:

```
Definition t := {ab: t0 | t0_inv ab}.
```

This is the type of abstract environments that is exported to the rest of the abstract domain hierarchy, via the `ab_ideal_env` type class (see Figure 3.9).

7.1.2. Concretization

The concretization of these abstract environments states that the equalities and the facts are indeed valid. That is, the expression associated to a variable indeed evaluates to the value of the variable, and the expressions of the facts indeed evaluate as `INZ 0` of `INZ 1`. As usually, this is defined as instances of the `gamma_op` type class, one for `t0` (the type of abstract environments not necessarily verifying the invariant) and one for `t` (the subset type of `t0`, with a proof of the invariant)[2]:

```
Record gamm0 (ab:t0) (ρ:var -> ideal_num) : Prop := {
  eqs_gamm :
    ∀ x e, MVar.get x ab.(eqs) = Some e -> eval_ite_iexpr ρ e (ρ x);
  facts_gamm :
    ∀ e b, MExp.get e ab.(facts) = Some b ->
           eval_iexpr ρ e (INz (if b then 1 else 0)) }.
Instance gamm0' : gamma_op t0 (var -> ideal_num) := gamm0.

Instance gamm : gamma_op t (var -> ideal_num) :=
  fun ab => gamm0 (proj1_sig ab).
```

7.2. Lattice Operations

Top

The top element is given by a record containing empty maps: this trivially verifies the invariant, and imposes no constraint on the concretization.

Comparisons

Comparisons between abstract environment simply check that the constraints of the right hand side is a subset of the constraints of the left hand side: it there is a mismatch in an expression associated to a variable, no attempt is made to prove that the expressions are equivalent.

Join

A naive implementation of the join operation would keep only the equations and facts present in the left hand side and the right hand side. However, this would make us unable to properly analyze pre-compiled Boolean operators, which is one of the motivation of this abstract domain.

Instead, we try to create new if-then-else selectors. To this end, we first search, in the current sets of facts, for an expression that is known to evaluate to a different Boolean on

[2] `proj1_sig: ∀ A (P:A -> Prop), {x | P x} -> A` is the function of the Coq standard library that returns the unconstrained value contained in an element of a subset type.

the right hand side and on the left hand side. Because facts are indexed by expressions, this can be easily done using the `shcombine_diff` function described in Section 9.1 (a variant of the `combine` primitive that iterates simultaneously over two maps, but exploiting the sharing). If such an expression is found, it is used as a pivot for creating if-then-else selectors.

If such a pivot is found, an if-then-else selector is inserted for every variable having different equalities on each branches[3]. Otherwise, we discard equalities that are not common to both branches. When an equality is available only in one branch, it is discarded, and when the same equality exists in both branches, it is left untouched.

Facts are discarded as soon as they are not common to both branches. This happens either when the associated Boolean is different, or when an expression is not associated to a Boolean in either branch.

Updating the `eqs_free` and `facts_free` fields adds a non-negligible level of complexity, that we will not detail here.

Widening

In this domain, we do not use the possibility of returning a different abstract environment for $\triangledown_1$ and $\triangledown_2$ (see Section 3.1.2). The same value is returned, and the same type is used for both components.

The implementation of the widening shares most of the code of the join operation, except it does not try to insert if-then-else selectors, and discards facts and equations that are not common to both operands of the widening.

This is a weak variant of the join, so (3.7) and (3.8) are clearly true. The step towards an informal proof of (3.10) is really immediate. Moreover, the set of equations and facts can only get smaller, so that the termination condition (3.9) is trivial.

7.3. Abstract Transfer Functions and Channels

The `forget` operation.

The implementation of `forget` needs not only remove the equation associated to the given variable, but also all the equations and facts mentioning it. This is the purpose of `eqs_free` and `facts_free`: finding easily the affected entries when modifying a variable.

Nonetheless, we must not omit to update `eqs_free` and `facts_free` accordingly, even for the variables different from the one we consider: because we are removing equations and facts, these removed entries have to be removed from sets corresponding to their other free variables.

In order to make the domain more robust to the use of variables for intermediate computations, the abstract domain tries to substitute occurrences of the forgotten variable with the expression it is equal to (when an expression with no if-then-else selectors is available, and when this expression does not contain itself the variable). If the substitution leads to an expression that is larger than a specified threshold, the equation or fact is discarded anyway.

[3] We discard the equalities if the resulting expression would exceed a pre-defined threshold.

The `assign` operation.

The `assign` transfer function[4] first calls the `forget` transfer function on the assigned variable, to adjust the equations and the facts mentioning it (either by discarding them or by substituting the variable by another expression). Moreover, if the assigned variable appears in the right hand side of the assignment, we try to substitute it as `forget` would. If this substitution step fails, which can happen when the resulting expression is too big or when the variable had no expression associated to before the assignment, the abstract environment is left as-is. That is, in this case, `assign` does the same thing as `forget`.

If the substitution step succeeds or if it is not needed, the right hand side of the assignment is used to build a new equation for the variable. Heuristically, this step is only performed if the expression is deterministic (i.e., it does not contain interval nodes): it is very unlikely that a non-deterministic expression still contains useful information for this domain.

Receiving `Fact_msg` messages.

The only messages treated by this abstract domain are `Fact_msg` messages. Messages of this kind are sent when analyzing the condition of an `if` statement. Similarly to the case of `assign`, we ignore such messages if they contain non-deterministic expressions. Otherwise, we store the expression as a fact[5] (or generate a contradiction if the negated fact is already present). Additionally, if the message carries the equality of a variable with an expression, and if this variable is associated with no existing equation, then this equation is inserted.

The `get_eq_expr` query channel.

This abstract domain answers `get_eq_expr` queries by performing a lookup in the map of equations.

7.4. Examples

In order to better understand the operation of this abstract domain, we present here a few examples of program analyses that we find representative of its behavior. In each of them, we briefly describe the purpose of the example, and then give a chunk of C code, commented with the relevant information stored in the symbolic abstract domain.

Unfolding during backward analysis

Unfolding variables in conditions may improve the precision of backward analyses. In the following, we present an example (already seen above) where this is indeed the case:

```
int b = verasco_any_int();
verasco_assume(0 < b && b < 1000); /* We bound b to avoid overflows. Any pair of
                                      bounds avoiding overflows would work.*/
int a = b*2;
/* Equation: a -> b*2 */
if(a < 10) {
  verasco_assert(b < 5);
```

[4] We omit the very special case where a variable is assigned to itself, in which case the abstract environment is returned unchanged.

[5] Actually, some preliminar Boolean simplifications are performed on the expression, to get rid of Boolean negation, for example.

```
}
```

Analysis of Boolean operators

One of the motivation of the design of this abstract domain was the precise analysis of lazy Boolean operators of C, such as `&&` and `||`, or even `?:`, the C ternary operator. To illustrate, let us examine the analysis of the following example, which is equivalent to the analysis of the condition `0 < b && b < 2`:

```
int b = verasco_any_int();
int tmp;

if(0 < b) {
  /* Fact: 0 < b -> true */
  tmp = b < 2;
  /* Equation: tmp -> b<2       Fact: 0<b -> true */
} else {
  /* Fact: 0<b -> false */
  tmp = 0;
  /* Equation: tmp -> 0         Fact: 0<b -> false */
}
/* Equation: tmp -> if 0<b then b<2 else 0 */
if(tmp) {
  /* After unfolding and analysis of tmp, we know that b==1: */
  verasco_assert(b == 1);
}
```

Substituting in equations

For better robustness against program transformations, e.g., the use of intermediate variables, the abstract domain is able to perform substitutions in the facts and equations. We illustrate that by the following example:

```
int b = verasco_any_int();
verasco_assume(0 < b && b < 1000); /* We bound b to avoid overflows. */

int a = b*2;
/* Equation: a -> b*2 */
a++;
/* Equation: a -> b*2+1 */
if(a < 11) {
  verasco_assert(b < 5);
}
```

Chapter 8

Octagons and Expression Linearization

Many interesting program properties cannot be proved using only non-relational domains, or even simple symbolic abstract domains. Instead, for many of these properties, a semantic rather than syntactic form of relational information is needed. As a motivating example, consider the following piece of C code:

```
int src[10] = { ... };
int dst[10];

int i_src = 0, i_dst = 9;
while(i_dst >= 0) {
  dst[i_dst] = src[i_src];
  i_dst--; i_src++;
}
verasco_assert(i_src == 10);
```

In this piece of code, we are copying an array into another array, reversing the order of its contents. It uses a common pattern, where the loop is indexed using two variables, `i_src` and `i_dst`. The former is incremented at each iteration, and the latter is decremented at each iteration. Because both arrays have the same size, it is unnecessary to check in the loop condition that both indices are in array bounds: only one check is necessary. Actually when a programmer writes this loops, she implicitly uses the invariant `i_src + i_dst = 9`, which the static analyzer needs to infer in order to prove correct the source array access and the final assertion. This invariant is relational: it involves several program variables. Moreover, it does not appear syntactically in the program, hence a symbolic abstract domain will not be able to infer it. At the numerical level, a popular class of relational abstract domains that would be able to infer this invariant is the class of linear abstract domains.

In general, a linear numerical abstract domain approximates a set of concrete numerical environments, by storing a finite set of linear inequalities over the variables $(X_v)_{v\in\mathcal{V}}$. These constraints are of the form:

$$\alpha_0 + \sum_{v\in\mathcal{V}} \alpha_v^{(j)} X_v \leq 0 \qquad (1 \leq j \leq J) \tag{8.1}$$

Thus, the general form of the concretization of an abstract environment, the set of values $(x_v)_{v\in\mathcal{V}}$ satisfying all the constraints, is a convex polyhedron. That is why the most general

linear relational abstract domain is called the *Polyhedron abstract domain*. In order to implement its transfer functions, this abstract domain uses algorithms taken from the field of general linear optimization, such as the Fourier-Motzkin elimination procedure and the Simplex algorithm.

A formally verified implementation of the Polyhedron abstract domain, for integer variables only, using the technique of a posteriori validation, has been integrated into Verasco by Fouilhé et al. as part of the *Verasco Polyhedral Library* [FMP13, FB14].

However, the Polyhedron abstract domain has serious drawbacks when the number of variables increases: the number of inequalities representing the polyhedron is not bounded, and, in practice, it is exponential in the number of variables[1][Min04]. Moreover, the implementation of a Polyhedron abstract domain over floating-point numbers is challenging, and, even when used on integer variables, an efficient library of infinite precision rational numbers is needed.

For this reason, we decided to implement an alternative linear abstract domain. There exists many relaxations of the Polyhedron abstract domain, that try to reduce the computational cost of the Polyhedron abstract domain, while maintaining sufficient precision. Generically, they are called *weakly relational* numerical abstract domain. Among others, the available alternatives include subpolyhedra [LL09], two variables per inequalities [SK10], zonotopes [PG06, GLGP12], octagons [Min04, Min06a], zones [Min04] and pentagons [LF08b].

In particular, the Octagon abstract domain is a nice trade-off between precision, ease of implementation and performance: it accurately represents many of the linear variable relationships appearing in a program, while using simple, well-known algorithms (based on the famous Floyd-Warshall shortest-path algorithm), and still remaining reasonably fast (all the operations have quadratic or cubic complexity on the number of variables). As shown by Miné [Min04], it can naturally use floating-point numbers in its concretization, but also in its implementation. It is very popular in the static analysis community, which explains why it regularly gains from algorithmic improvements [CRK14, BHZ09, SPV15].

We implemented the Octagon abstract domain in Verasco. As reported by the designers of Astrée [CCF+09], the quadratic or cubic complexity of most of its algorithms still make it unusable as-is for a reasonable number of variables. A common solution is the use of *variable packing* [Min04, Section 8.4.2], where the Octagon abstract domain is only used on small packs of variables. By lack of time to experiment with highly tuned packing heuristics, we decided not to use this strategy. Instead, we developed original algorithms for the octagons abstract domain, where unrelated pairs of variables do not count towards the algorithm cost: the total complexity of the cost of the analysis of two independent sets of variables is very close to the sum of the costs of the analyses of the two sets of variables, taken independently. We describe the theoretical setting behind the abstract domain of octagons, together with these new algorithms in Section 8.2. The implementation details are described in Section 8.3.

A linear abstract domain is unable to understand directly the expressions appearing in a program, because the programming language typically contains non-linear constructs. Thus, an intermediate step, where arbitrary expressions are transformed into (quasi-)linear ones, is needed. In Section 8.1, we describe our linearization mechanism, which is greatly inspired from that of Miné [Min04].

[1] Another representation of polyhedra is possible, the *frame representation*, where the domain uses a set of vertices, whose polyhedron is the convex hull. However, the number of such needed vertices in practice is also exponential in the number of variables.

8.1. Expression Linearization

The role of the linearization abstract domain is to transform most of the expressions produced by the front-end of Verasco into quasi-linear expressions. It interacts with the rest of the hierarchy using two different mechanisms: first, it answers `linearize_expr` queries, by returning a quasi-linear expression corresponding to the query (or the value `None` if the conversion fails). Second, when receiving a `Fact_msg` message, coming from the front-end of the analyzer entering an `if` statement, it tries to convert it into a quasi-linear constraint of the form $e = 0$ for some quasi-linear expression e. Such a quasi-linear constraint is then embedded in a `Linear_zero_msg` message, which can be analyzed by the octagon abstract domain.

This architecture, where the linearization system is independent from the abstract domain, make us able to reuse the linearization system for other future linear abstract domains. Another possibility, followed by the Verasco Polyhedral Library, is to use smarter linearization techniques, that are required to be integrated into the abstract domain itself.

8.1.1. Quasi-Linear Expressions

A *quasi-linear expression* over variables $(X_v)_{v\in\mathcal{V}}$ is of the following form:

$$[\alpha_0^-;\alpha_0^+] + \sum_{v\in\mathcal{V}} [\alpha_v^-;\alpha_v^+] \times X_v \tag{8.2}$$

with $\forall v \in \mathcal{V}\cup\{0\},\ \alpha_v^- \in \mathbb{R}\cup\{-\infty\} \wedge \alpha_v^+ \in \mathbb{R}\cup\{+\infty\} \wedge \alpha_v^- \leq \alpha_v^+$. That is, it is an affine expression with interval coefficients.

Quasi-linear expressions have to be interpreted in ideal *real* arithmetic: given the values of the variables $(x_v)_{v\in\mathcal{V}}$, the *values of the quasi-linear expression* are real numbers that can be written as follows:

$$\alpha_0 + \sum_{v\in\mathcal{V}} \alpha_v x_v \qquad \text{with} \qquad \forall v \in \mathcal{V}\cup\{0\},\ \alpha_v \in [\alpha_v^-;\alpha_v^+] \tag{8.3}$$

In the Coq implementation, quasi-linear expressions are represented using a pair of an interval (for the constant term), and a map from variable identifiers to intervals[2] (for variable coefficients). The intervals bounds are represented by floating-point numbers. The semantics of quasi-linear expressions is defined in a very similar manner as in (8.3), except that, because the variables can be either integer or floating-point and that (8.3) has to be evaluated in $\mathbb{R}$, we use two conversion operators (from $\mathbb{Z}$ to $\mathbb{R}$, and from floating-point numbers[3] to $\mathbb{R}$).

Intervalization of quasi-linear expressions

Given a quasi-linear expression and intervals $([x_v^-;x_v^+])_{v\in\mathcal{V}}$ for the values of the variables it contains (i.e., $\forall v \in \mathcal{V},\ x_v \in [x_v^-;x_v^+]$), it is easy to compute an interval for the values of the quasi-linear expression, by evaluating the expression using interval arithmetic. This gives

[2] In fact, each variable is also associated with a pre-computed interval for its value, in order to avoid repeating `get_itv` queries on the same variable.

[3] In the Coq development, if one of the variables appearing in the expression has a non-finite floating-point value (i.e., $\pm\infty$ or `NaN`), then the evaluation of the quasi-linear expression is stuck. This introduces a subtle difference between an expression where a variable appears with the coefficient $[0;0]$ and the same expression with the variable occurrence removed.

the following interval for the values of the quasi-linear expression:

$$\left[\begin{array}{l} \alpha_0^- + \displaystyle\sum_{v\in\mathcal{V}} \min\{x_v^-\alpha_v^-;\ x_v^+\alpha_v^-;\ x_v^-\alpha_v^+;\ x_v^+\alpha^+\} \quad ; \\ \qquad\qquad \alpha_0^+ + \displaystyle\sum_{v\in\mathcal{V}} \max\{x_v^-\alpha_v^-;\ x_v^+\alpha_v^-;\ x_v^-\alpha_v^+;\ x_v^+\alpha^+\} \end{array}\right] \tag{8.4}$$

In Verasco, we do not compute exactly this interval: instead, we compute a sound approximation by rounding the bounds towards the exterior of the interval.

8.1.2. Converting Expressions into Quasi-Linear Expressions

Converting numerical expressions into quasi-linear ones consists in finding a quasi-linear expression whose set of values is a superset of the set of values[4] of the given expression. Because not all expressions appearing in the program are linear, this process is an approximation: in some cases, the returned expression will evaluate to more values than the given expression. This process will fail when no suitable approximation is found or when it could not be proved that the values of the given expression are finite.

We follow a naive but effective bottom-up strategy similar to the one described by Miné [Min04]: our humble aim is to get a precise quasi-linear approximation for expressions that are already close to linear in the source program, and, for now, we do not aim at implementing more precise techniques such as the ones proposed by Maréchal and Périn [MP14], which are able to build precise approximations for arbitrary multivariate polynomials.

Thus, we compute quasi-linear versions of expressions recursively, by considering expression nodes one after the other. We explain the algorithm as if we had arbitrary precision real arithmetic operators. In our implementation, we only use floating-point bounds for intervals: approximations of the real intervals are computed by rounding bounds towards the exterior of the interval.

Expression leaves

Except for the ones that can evaluate to non-finite values[5], whose linearization fail, the linearization of expression leaves is always trivial. Indeed, intervals and constants are translated to an only constant term, and variables are translated to a quasi-linear expression containing only the variable with coefficient $[1;1]$.

Linear operations

Addition, subtraction, negation and multiplication by a constant or an interval are inherently quasi-linear operators, so they can be applied pointwisely to each term of the operands. More precisely, consider the two quasi-linear expressions A and B defined by:

$$A = [\alpha_0^-;\alpha_0^+] + \sum_{v\in\mathcal{V}} [\alpha_v^-;\alpha_v^+] \times X_v \tag{8.5}$$

$$B = [\beta_0^-;\beta_0^+] + \sum_{v\in\mathcal{V}} [\beta_v^-;\beta_v^+] \times X_v \tag{8.6}$$

[4] Or, more precisely, the interpretation in $\mathbb{R}$ of these values.
[5] E.g., non-finite floating-point constants and variables whose interval contains non-finite values.

Their sum, their difference, the negation of the first, and multiplication by $[u^-; u^+]$ of the first are defined by, respectively:

$$A + B = [\alpha_0^- + \beta_0^-; \alpha_0^+ + \beta_0^+] + \sum_{v \in \mathcal{V}} [\alpha_v^- + \beta_v^-; \alpha_v^+ + \beta_v^+] \times X_v \tag{8.7}$$

$$A - B = [\alpha_0^- - \beta_0^+; \alpha_0^+ - \beta_0^-] + \sum_{v \in \mathcal{V}} [\alpha_v^- - \beta_v^+; \alpha_v^+ - \beta_v^-] \times X_v \tag{8.8}$$

$$-A = [-\alpha_0^+; -\alpha_0^-] + \sum_{v \in \mathcal{V}} [-\alpha_v^+; -\alpha_v^-] \times X_v \tag{8.9}$$

$$[u^-; u^+] \times A = [u^-; u^+] \boxtimes [\alpha_0^-; \alpha_0^+] + \sum_{v \in \mathcal{V}} [u^-; u^+] \boxtimes [\alpha_v^-; \alpha_v^+] \times X_v \tag{8.10}$$

Where $[u^-; u^+] \boxtimes [\alpha^-; \alpha^+]$ is a notation for the interval:

$$\left[\min\{u^-\alpha^-;\ u^+\alpha^-;\ u^-\alpha^+;\ u^+\alpha^+\}\ ;\ \max\{u^-\alpha^-;\ u^+\alpha^-;\ u^-\alpha^+;\ u^+\alpha^+\}\right] \tag{8.11}$$

These operations are used directly for integer arithmetic. For floating-point arithmetic, they are used in conjunction with the techniques for modeling rounding described in the next section.

Handling rounding

All the operations involving rounding, such as floating-point arithmetic or conversions operators, are decomposed into two steps: the first step is the actual operation in real arithmetic, while the second step is the rounding. In this section, we explain how standard rounding modes can be precisely accounted when using quasi-linear expressions.

The rounding mode used when converting from floating-point numbers to integers is round-to-zero[6]. This operation is not linear: it cannot be exactly represented as a linear expression. However, it can be noted that, when the input x is non-negative, then its integer rounding is in $]x - 1; x]$, and, when $x \leq 0$, then its integer rounding is in $[x; x + 1[$. For the sake of simplicity, We do not distinguish these two cases and model integer rounding towards zero by the addition of the interval $[-1; 1]$.

Single or double precision floating-point rounding is more complex. In CompCert and Verasco, we support only the most commonly used round-to-nearest mode. Let us consider the case where we have to model the rounding of an arbitrary real number x, approximated by the quasi-linear expression e, to a single or double precision floating-point number. We have to distinguish three cases:

- If $|x|$ is larger than or equal to a known threshold M, the rounding of x overflows, and the result is $\pm\infty$, so no quasi-linear expression suits. The linearization should fail. Thus, when approximating the rounding of a quasi-linear expression, we first intervalize the input, and check that it cannot overflow.

- If $|x|$ is strictly smaller than a known threshold m, then x is in the denormal range: the exponent of the result is fixed[7] to its minimal value, and the *absolute* rounding error is bounded in absolute value by the half of a denormal ulp. This kind of rounding can

[6] That is, if given a non-negative value, it returns the largest smaller integer, and, if it is non-positive, the smallest larger integer.

[7] Except in the corner case where x is very close to m, in which case it rounds to m, which has a different exponent. The bound on the absolute rounding error is still correct, though.

be modeled by the addition to e of a known interval $[-\eta; \eta]$, for a very small known real number η.

- Otherwise, x is in the normalized range. In this case, the *relative* rounding error is in a known small interval. This kind of rounding can be modeled by the multiplication of e with a known small interval $[1 - \varepsilon;\ 1 + \varepsilon]$ around 1.

For the sake of simplicity, we do not check whether our expression lies in the second or third case: to model floating-point rounding, we return the join of $[1-\varepsilon;\ 1+\varepsilon] \times e$ and $e + [-\eta; \eta]$.

We summarize in the table below the values of M, m, ε and η in single and double precision. Note that not all of these values are exactly representable as double precision floating-point numbers, so that we use sound approximations in practice.

	M	m	η	ε
Single precision	$2^{128} - 2^{103}$	2^{-126}	2^{-150}	$\frac{1}{1+2^{24}}$
Double precision	$2^{1024} - 2^{970}$	2^{-1022}	2^{-1075}	$\frac{1}{1+2^{53}}$

Other operators

Integer or floating-point division and general multiplication are handled similarly: a `get_itv` query is used to compute an interval for one of the operands, effectively transforming the multiplication or division into a multiplication by an interval, optionally followed by a rounding. These two operations are handled as above.

In the case of multiplication, we always choose the first operand to be intervalized. This is a very naive choice: the reader may refer to Miné's work for better heuristics [Min04].

No other operator is supported. Of course, a few other operators such as shifts could have reasonable quasi-linear approximations, but we leave it as future work. When encountering an unsupported operator, we do a `get_itv` query to get an interval for the expression and return a quasi-linear expression consisting in the computed interval only.

8.1.3. Linearization Abstract Domain

The abstract environments of the linearization abstract domain do not contain any information: there is only one abstract environment, of type `unit`. This makes all the abstract operations trivial to implement. Miné [Min04] proposes several improvements of the linearization process, that would make it possible to store in an abstract environment quasi-linear expressions for variables, and inline them afterwards. We can imagine going into this direction as future work, if the gain in precision is worth it.

As already sketched in the introduction of Section 8.1, the linearization abstract domain interacts with others in two ways: it answers `linearize_expr` queries by converting an expression passed as parameter into a quasi-linear approximation, and sends `Linear_zero` messages when receiving `Fact_msg` messages.

More precisely, it detects `Fact_msg` messages that correspond to a numerical comparison in the source code, linearizes both operands of the comparison, and computes the difference of the two linearized forms. Then, depending on the kind of the comparison, it extrapolates the bounds of the constant term of the computed expression, in order to build a quasi-linear expression that evaluates to the real 0 whenever the analyzed condition is true. For

example, if the condition implies that the quasi-linear expression e evaluates to a non-negative number, then the lower bound of the constant term is changed to $-\infty$ so that the expression evaluates to 0.

8.2. Theoretical Setting for Octagons

The abstract domain of octagons was introduced by Miné [Min04, Min06a]. Since then, many improvements in algorithms and in the proofs of completeness and soundness have been developed [CRK14, BHZ09, SPV15]. We give here a summary of the results in the literature, together with an improvement of the already existing algorithms that we use in Verasco.

This improvement makes us able to use a sparse representation of octagons, where we do not need to store any data relating variables when this data can be recovered from the available non-relational information. The idea of such a sparse representation has already been studied recently [SPV15], but the authors of this work did not explain how the relational information need not be stored when it is implied by the non-relational one, and how this has non-trivial implications for the join operator. In particular, their algorithm will switch to the fully dense representation as soon as some relational information is known for every variable [SPV15, Section 5.3].

By contrast, our method guarantees that the conjunct analysis of two independent sets of variables uses only the sum of the resources of the two independent analyses, both in terms of memory and computation time. This goal is achieved by removing the need for the *strengthening* operation, that usually breaks sparsity. Thus, instead of maintaining the invariant that the representation of octagons is fully saturated, we maintain the invariant that it is *weakly closed*. We prove that no precision is lost by using weakly closed abstract values, except for the join and widening operators, for which we give modified algorithms.

In this section, we study octagons in a simplified setting: we assume that all the computations can be made in arbitrary precision, we omit data structure details, and the only operations we support are the following:

- comparisons (see Section 8.2.3);
- the `forget` operation, that removes all the information known on a variable (see Section 8.2.4);
- the $\sqcup$ operation, that computes the least upper bound of two abstract environments (see Section 8.2.5);
- a weak version of the `assume` primitive, that makes us able to insert a constraint of the form $\pm x \pm y \leq C$ for two non-necessarily different variables x and y and $C \in \mathbb{R}$ (see Section 8.2.6);
- the widening operator (see Section 8.2.7).

These operators summarize all the algorithmic issues of octagons[8] when considering environments with real (or rational) values. The case of integer environments is discussed in Section 8.2.8. The implementation details in Verasco will be addressed later in Section 8.3.

[8] One important primitive missing from that list is the `assign` primitive. The assignment $x := e$ can be emulated by assuming the equality between a fresh variable y and e, then `forgeting` x, assuming $x = y$ and finally `forgeting` y. In Verasco, `assign` is not actually implemented using this scheme, but with an algorithm returning the same result.

8.2.1. Difference Bound Matrices

Let $\mathbb{V}_+$ be a finite set of variables. We call a *regular environment* a function from $\mathbb{V}_+$ to $\mathbb{R}$. The role of the Octagon abstract domain is to approximate sets of regular environments ρ. To that end, the abstract domain of octagons stores a set of inequalities of the following form:

$$\pm\rho(u) \pm \rho(v) \leq M_{uv} \qquad u, v \in \mathbb{V}_+ \tag{8.12}$$

This corresponds to giving bounds to sums and differences of values of ρ. Moreover, if we use twice the same variable with the same sign, we see that, using such constraints, we can express interval constraints over values of an environment[9].

In order to handle all the different combinations of signs in these constraints in a unified way, we introduce the set $\mathbb{V}_\pm$ of *signed variables*. Signed variables are of two kinds: they are either usual variables from $\mathbb{V}_+$, called *positive variables* in the context of signed variables, or their opposites form, *negative variables*. We equip $\mathbb{V}_\pm$ with an involutive operator, associating to each signed variable v its opposite $\overline{v}$, such that v is positive if and only if $\overline{v}$ is negative.

Regular environments are canonically extended to signed variables by taking $\rho(\overline{u}) = -\rho(u)$. More generally, we define *irregular environments* as functions from $\mathbb{V}_\pm$ to $\mathbb{R}$, and consider the set of regular environments as a subset of irregular environments. Regular environments are exactly irregular environments ρ that satisfy the property $\forall v, \rho(\overline{v}) = -\rho(v)$.

Using this new formalism, octagonal constrains of the form (8.12) can be seen as upper bounds on differences of values of ρ, a regular environment:

$$\rho(u) - \rho(v) \leq N_{uv} \qquad u, v \in \mathbb{V}_\pm \tag{8.13}$$

This has two benefits: first, all the different kinds of constraints allowed by (8.12) get factored out into one simpler form. Second, we can see these constraints as constraints on irregular environments, and further constrain them as being regular: we see that the study of the Octagon abstract domain will start by the study of a simpler abstract domain, where only differences of variables are bounded. The set of constraints, called *potential constraints*, of such abstract domain is well studied in the linear optimization literature, because it corresponds to the well-known shortest path problem in a weighted directed graph. The study of the Octagon abstract domain is an extension of the study of potential constraints.

Such a set of constraints is represented as difference bound matrices: a *difference bound matrix*, or *DBM*, is a matrix $(B_{uv})_{(u,v)\in\mathbb{V}_\pm^2}$ of elements of $\mathbb{R} \cup \{+\infty\}$. The meaning of these constraints is given by two concretization functions γ_{pot} and γ_{oct}, that associate to a DBM the set of irregular or regular environments, respectively, satisfying all the constraints:

$$\gamma_{pot}(B_{uv}) = \{\rho : \mathbb{V}_\pm \to \mathbb{R} \mid \forall uv \in \mathbb{V}_\pm,\ \rho(u) - \rho(v) \leq B_{uv}\} \tag{8.14}$$

$$\gamma_{oct}(B_{uv}) = \{\rho \in \gamma_{pot}(B_{uv}) \mid \forall u \in \mathbb{V}_\pm,\ \rho(\overline{u}) = -\rho(u)\} \tag{8.15}$$

We denote as $\overset{\sharp}{\leq}$ the natural order relation over DBMs, defined as follows:

$$A \overset{\sharp}{\leq} B \Leftrightarrow \forall uv \in \mathbb{V}_\pm,\ A_{uv} \leq B_{uv} \tag{8.16}$$

[9] For this reason, one may think that the Octagon abstract domain subsumes the interval abstract domain. This is wrong, because the octagon abstract domain only support a subset of the arithmetic operators, and it has a less precise handling of some others (with respect to floating-point rounding, for example).

The following easy lemma states that this order relation makes γ_{pot} and γ_{oct} increasing, which makes $\overset{\sharp}{\leq}$ a good candidate for a comparison operator of the Octagon abstract domain[10]:

Lemma 8.2.1. *Let A and B be two DBMs such that $A \overset{\sharp}{\leq} B$. Then, we have:*

$$\gamma_{pot}(A) \subseteq \gamma_{pot}(B) \qquad\qquad \gamma_{oct}(A) \subseteq \gamma_{oct}(B)$$

For any non-empty set S of irregular environments, there exists a minimal (in the sense of $\overset{\sharp}{\leq}$) DBM that approximates it. That is, there exists a minimal DBM $\alpha(S)$ such that $S \subseteq \gamma_{pot}(\alpha(S))$. This property follows immediately from the definition of α:

$$\alpha(S)_{uv} = \sup_{\rho \in S} \{\rho(u) - \rho(v)\} \tag{8.17}$$

This function α is called the concretization function. We can easily see that α is an increasing function[11]. Moreover, α does not only return best abstractions for γ_{pot}, but also for γ_{oct}: if the set S contains only regular environments, we can see that $\alpha(S)$ is also the minimal DBM such that $S \subseteq \gamma_{oct}(\alpha(S))$.

8.2.2. Closures

The DBM data structure contains many redundancies: that is, many DBMs have the same concretization. This is a problem, because the abstract environments that we manipulate are therefore not necessarily the most precise ones, and this can lead to imprecision. Thus, usually, an implementation of the Octagon abstract domain maintains the invariant that it only manipulates "canonical" forms of DBMs, such that $B = \alpha(\gamma_{oct}(B))$ (or, respectively, $B = \alpha(\gamma_{pot}(B))$). Such "canonical" DBMs are always the best possible representative over all the DBMs with the same concretization.

We can see that, for DBMs with non-empty concretization, canonical DBMs always exist:

Lemma 8.2.2. *Let B be a DBM with non-empty concretization. Then $\alpha(\gamma_{pot}(B))$ (respectively, $\alpha(\gamma_{oct}(B))$) has the same concretization as B:*

$$\gamma_{pot}(\alpha(\gamma_{pot}(B))) = \gamma_{pot}(B) \qquad \text{(respectively } \gamma_{oct}(\alpha(\gamma_{oct}(B))) = \gamma_{oct}(B)\text{)}$$

It follows that $\alpha(\gamma_{pot}(B))$ (respectively, $\alpha(\gamma_{oct}(B))$) is a best abstraction. That is,

$$\alpha(\gamma_{pot}(\alpha(\gamma_{pot}(B)))) = \alpha(\gamma_{pot}(B)) \qquad \text{(respectively } \alpha(\gamma_{oct}(\alpha(\gamma_{oct}(B)))) = \alpha(\gamma_{oct}(B))\text{)}$$

Proof. This is a classical property of Galois connections. We detail the proof only for γ_{pot}: as we have seen, $\alpha(\gamma_{pot}(B))$ is the smallest DBM whose concretization is larger than that of B, so $\gamma_{pot}(B) \subseteq \gamma_{pot}(\alpha(\gamma_{pot}(B)))$.

Because B is a candidate for such a DBM, we also have $\alpha(\gamma_{pot}(B)) \overset{\sharp}{\leq} B$, by minimality. It follows $\gamma_{pot}(\alpha(\gamma_{pot}(B))) \subseteq \gamma_{pot}(B)$. □

An important fact is that we can characterize best abstractions using the values they contain, and that we have algorithms to compute them. We will expose these characterizations, together with these algorithms. Moreover, in the case of octagons, we will give

[10] As we will see in Section 8.2.3, this is *not* the order relation we use as a comparison operator in our implementation of the Octagon abstract domain.

[11] Abstract interpretation or category theory experts will easily recognize here that $\overset{\sharp}{\leq}$ defines a complete lattice over DBMs extended with a bottom element, and that the pairs (γ_{pot}, α) and (γ_{oct}, α) form Galois connections.

weaker conditions over DBMs, that do not ensure canonicity, but that can be computed more quickly without losing any precision.

So, let us search for properties verified by canonical DBMs. We will not consider DBMs with empty concretization, since they do not have a canonical representative, thus a special element $\bot$ is used as their canonical form. The following lemma states that canonicity for γ_{oct} implies canonicity for γ_{pot}:

Lemma 8.2.3. *Let B be a DBM such that $\gamma_{oct}(B) \neq \emptyset$ and $B = \alpha(\gamma_{oct}(B))$. Then $B = \alpha(\gamma_{pot}(B))$.*

Proof. We have: $\gamma_{oct}(B) \subseteq \gamma_{pot}(B)$, thus $B = \alpha(\gamma_{oct}(B)) \stackrel{\sharp}{\leq} \alpha(\gamma_{pot}(B))$. Moreover, $\alpha(\gamma_{pot}(B)) \stackrel{\sharp}{\leq} B$ by minimality of $\alpha(\gamma_{pot}(B))$. □

As a result, we will first characterize canonicity for γ_{pot} and then refine this property for γ_{oct}.

Best abstractions for γ_{pot}

A first step is to remark that canonical DBMs always have null diagonal values. Moreover, they should always verify the triangular inequality. We call such DBMs *closed* DBMs:

Definition 8.2.1 (Closed DBM)**.** *A* closed *DBM is a DBM B verifying the two following properties:*

- $\forall v \in \mathbb{V}_{\pm},\ B_{vv} = 0$
- $\forall uvw \in \mathbb{V}_{\pm},\ B_{uw} \leq B_{uv} + B_{vw}$

It appears that closed DBMs are exactly the best abstractions for γ_{pot}:

Theorem 8.2.4. *Let B be a DBM. The two following properties are equivalent:*

(i) B is closed

(ii) $\gamma_{pot}(B) \neq \emptyset$ and $B = \alpha(\gamma_{pot}(B))$

Proof. See, e.g., [Min04, Theorem 3.3.6]. □

An interesting consequence of this theorem is that closed DBMs always have non-empty concretizations.

The typical algorithm used to detect the emptiness of the concretization of a DBM and to compute closures is the Floyd-Warshall algorithm. Our implementation does not explicitly compute closed DBMs, so we do not use this algorithm as-is. For this reason, we will not detail it here. Instead, we refer the interested reader to previous works [Min04, BHZ09].

Best abstractions for γ_{oct}

We now refine the notion of closure to canonical forms for γ_{oct}. We know that if ρ is a regular environment, then $\rho(u) - \rho(v) = \rho(\overline{v}) - \rho(\overline{u})$, so for any non-empty set S of regular environments, we see that, $\alpha(S)_{uv} = \alpha(S)_{\overline{v}\,\overline{u}}$. Thus, canonical DBMs for γ_{oct} will verify the *coherence* property:

Definition 8.2.2 (Coherent DBM)**.** *A DBM B is* coherent *when:*

$$\forall uv \in \mathbb{V}_{\pm},\ B_{uv} = B_{\overline{v}\,\overline{u}}$$

Moreover, we can see that the matrix elements of the form $B_{u\overline{u}}$ (for $u \in \mathbb{V}_{\pm}$) impose interval constraints on values of ρ. These interval constraints can be themselves combined to entail constraints on any difference of values of ρ. For this reason, we guess that canonical forms for γ_{oct} will verify the following strong closedness property:

Definition 8.2.3 (Strongly closed DBM)**.** *A DBM B is* strongly closed *when it is closed and coherent and:*

$$\forall uv \in \mathbb{V}_{\pm},\ B_{uv} \leq \frac{B_{u\overline{u}} + B_{\overline{v}v}}{2}$$

This condition is necessary and sufficient: strong closedness characterizes canonical DBMs for γ_{oct}.

Theorem 8.2.5. *Let B be a DBM. The two following properties are equivalent:*

(i) B is strongly closed

(ii) $\gamma_{oct}(B) \neq \emptyset$ and $B = \alpha(\gamma_{oct}(B))$

Proof. See, e.g., [Min04, Theorems 4.3.2 and 4.3.3]. □

The usual method [BHZ09] for computing strong closures consists in first ensuring that the given matrix is coherent, then computing a closure (i.e., a canonical representative in the sense of $\gamma_{pot}(B)$), and, finally, performing a so-called *strengthening step*[12]:

Definition 8.2.4 (Strengthening)**.** *Let B be a DBM. The* strengthening *of B, noted $^\sharp\mathcal{S}(B)$ is defined by:*

$$^\sharp\mathcal{S}(B)_{uv} = \min\left\{\frac{B_{u\overline{u}} + B_{\overline{v}v}}{2}; B_{uv}\right\}$$

The following easy lemma states the soundness of strengthening:

Lemma 8.2.6. *Le B be a DBM. Then $\gamma_{oct}(B) = \gamma_{oct}(^\sharp\mathcal{S}(B))$*

Proof. $^\sharp\mathcal{S}(B) \stackrel{\sharp}{\leq} B$, so $\gamma_{oct}(^\sharp\mathcal{S}(B)) \subseteq \gamma_{oct}(B)$.

Conversely, let $\rho \in \gamma_{oct}(B)$. ρ is a regular environment. It verifies, for $u, v \in \mathbb{V}_{\pm}$:

$$\rho(u) - \rho(v) \leq \frac{B_{u\overline{u}} + B_{\overline{v}v}}{2} \qquad\qquad \rho(u) - \rho(v) \leq B_{uv}$$

Thus: $\rho(u) - \rho(v) \leq {}^\sharp\mathcal{S}(B)_{uv}$, and $\rho \in \gamma_{oct}(^\sharp\mathcal{S}(B))$. □

In order to prove that the strengthening operation does indeed compute the strong closure, another ingredient is needed: we need to prove that it computes strong closures when given coherent closed DBMs. Instead of proving this result directly, we introduce yet another different notion of closure, that we will reuse later to explain our sparse algorithm:

Definition 8.2.5 (Weakly closed DBM)**.** *Let B be a DBM. We say that B is* weakly closed *when any of the two following* equivalent *statements hold:*

(i) B has a null diagonal and $^\sharp\mathcal{S}(B)$ is strongly closed;

(ii) B has a null diagonal, $^\sharp\mathcal{S}(B)$ is coherent, and:

$$\forall uvw,\ {}^\sharp\mathcal{S}(B)_{uw} \leq B_{uv} + B_{vw} \tag{8.18}$$

[12]This is actually an improvement of the method described initially by Miné [Min04].

Proof. We prove here that the two properties are equivalent. The proof of this result is technical. As an additional guarantee of its correctness, we mechanically verified it in Coq. As this is a precision result neither needed nor correct in the actual implementation because of the use of floating-point numbers, the Coq proof is not present in the main development.

(i) ⇒ (ii) We first assume that B has a null diagonal and that $\sharp\mathcal{S}(B)$ is strongly closed. In this case, $\sharp\mathcal{S}(B)$ is coherent, and:

$$\forall uvw,\ \sharp\mathcal{S}(B)_{uw} \leq \sharp\mathcal{S}(B)_{uv} + \sharp\mathcal{S}(B)_{vw} \leq B_{uv} + B_{vw}$$

(ii) ⇒ (i) Conversely, we assume that B has a null diagonal, that $\sharp\mathcal{S}(B)$ is coherent and that:

$$\forall uvw,\ \sharp\mathcal{S}(B)_{uw} \leq B_{uv} + B_{vw}$$

We need to prove that $\sharp\mathcal{S}(B)$ is strongly closed. It remains to prove the following three properties:

$$\forall uv,\ \sharp\mathcal{S}(B)_{uv} \leq \frac{\sharp\mathcal{S}(B)_{u\overline{u}} + \sharp\mathcal{S}(B)_{\overline{v}v}}{2} \qquad\qquad \forall u,\ \sharp\mathcal{S}(B)_{uu} = 0$$

$$\forall uvw,\ \sharp\mathcal{S}(B)_{uw} \leq \sharp\mathcal{S}(B)_{uv} + \sharp\mathcal{S}(B)_{vw}$$

The first one is easy:

$$\forall uv,\ \sharp\mathcal{S}(B)_{uv} \leq \frac{B_{u\overline{u}} + B_{\overline{v}v}}{2} = \frac{\sharp\mathcal{S}(B)_{u\overline{u}} + \sharp\mathcal{S}(B)_{\overline{v}v}}{2}$$

Concerning the remaining two, we first remark that:

$$\forall u,\ \sharp\mathcal{S}(B)_{uu} \leq B_{u\overline{u}} + B_{\overline{u}u}$$

By reasoning on the two different possibles expressions for $\sharp\mathcal{S}(B)_{uu}$, we deduce $\forall u,\ 0 \leq B_{u\overline{u}} + B_{\overline{u}u}$. It follows that $\forall u,\ \sharp\mathcal{S}(B)_{uu} = 0$.

Concerning the last property (the triangular inequality), let $u, v, x \in \mathbb{V}_{\pm}$. We distinguish four cases:

- If $\sharp\mathcal{S}(B)_{uv} = B_{uv}$ and $\sharp\mathcal{S}(B)_{vw} = B_{vw}$, we use (8.18) directly.

- If $\sharp\mathcal{S}(B)_{vv} = \dfrac{B_{u\overline{u}} + B_{\overline{v}v}}{2}$ and $\sharp\mathcal{S}(B)_{vw} = \dfrac{B_{v\overline{v}} + B_{\overline{w}w}}{2}$, we use the previously proved property $0 \leq B_{v\overline{v}} + B_{\overline{v}v}$, and deduce:

$$\sharp\mathcal{S}(B)_{uw} \leq \frac{B_{u\overline{u}} + B_{\overline{w}w}}{2} \leq \sharp\mathcal{S}(B)_{uv} + \sharp\mathcal{S}(B)_{vw}$$

- The two other cases are symmetric. We assume, for example, that $\sharp\mathcal{S}(B)_{uv} = B_{uv} \neq \dfrac{B_{u\overline{u}} + B_{\overline{v}v}}{2}$ and $\sharp\mathcal{S}(B)_{vw} = \dfrac{B_{v\overline{v}} + B_{\overline{w}w}}{2}$.

  We have $\sharp\mathcal{S}(B)_{uv} = \sharp\mathcal{S}(B)_{\overline{v}\,\overline{u}}$ by coherence, so $\sharp\mathcal{S}(B)_{\overline{v}\,\overline{u}} \neq \dfrac{B_{u\overline{u}} + B_{\overline{v}v}}{2}$. So:

$$\sharp\mathcal{S}(B)_{uv} = \sharp\mathcal{S}(B)_{\overline{v}\,\overline{u}} = B_{\overline{v}\,\overline{u}} = B_{uv}$$

Now, the assumption (8.18) gives:

$$\sharp\mathcal{S}(B)_{v\overline{u}} \leq B_{v\overline{v}} + B_{\overline{v}\,\overline{u}}$$

We have again two cases:

- Either $B_{v\overline{u}} \leq B_{v\overline{v}} + B_{\overline{v}\,\overline{u}}$. Then:

$$B_{u\overline{u}} = \sharp\mathcal{S}(B)_{u\overline{u}} \leq B_{uv} + B_{v\overline{u}} \leq B_{uv} + B_{v\overline{v}} + B_{\overline{v}\,\overline{u}}$$

  It follows:

$$\begin{aligned} \frac{B_{u\overline{u}}}{2} &\leq B_{uv} + \frac{B_{v\overline{v}}}{2} \\ \frac{B_{u\overline{u}} + B_{\overline{w}w}}{2} &\leq B_{uv} + \frac{B_{v\overline{v}} + B_{\overline{w}w}}{2} \\ \sharp\mathcal{S}(B)_{uw} &\leq \sharp\mathcal{S}(B)_{uv} + \sharp\mathcal{S}(B)_{vw} \end{aligned}$$

- Either $\dfrac{B_{v\overline{v}} + B_{u\overline{u}}}{2} \leq B_{v\overline{v}} + B_{\overline{v}\,\overline{u}}$. If $B_{v\overline{v}} = +\infty$, the inequality is immediate. Otherwise, we have:

$$\frac{B_{u\overline{u}}}{2} \leq \frac{B_{v\overline{v}}}{2} + B_{uv}$$

  We conclude as in the previous case. □

We shall emphasize that, contrarily to the other notions of closedness, weak closedness is not a notion of canonicity. In particular, there are many different weakly closed DBMs with the same concretization.

Using the theorem-definition of weakly closed DBMs, we can prove the following theorem, that states the correctness of the strong closure algorithm sketched above.

Theorem 8.2.7. *Let B be a coherent DBM with $\gamma_{oct}(B) \neq \emptyset$. Then:*

$$\alpha(\gamma_{oct}(B)) = \sharp\mathcal{S}(\alpha(\gamma_{pot}(B)))$$

In particular, if B is coherent and closed, then $\sharp\mathcal{S}(B)$ is strongly closed.

Proof. For the first statement, it suffices to prove that $\sharp\mathcal{S}(\alpha(\gamma_{pot}(B)))$ is strongly closed. Indeed, in this case, we have:

$$\begin{aligned} \sharp\mathcal{S}(\alpha(\gamma_{pot}(B))) &= \alpha(\gamma_{oct}(\sharp\mathcal{S}(\alpha(\gamma_{pot}(B))))) && \text{by Theorem 8.2.5} \\ &= \alpha(\gamma_{oct}(\alpha(\gamma_{pot}(B)))) && \text{by Lemma 8.2.6} \\ &= \alpha(\{\rho \text{ regular} \mid \rho \in \gamma_{pot}(\alpha(\gamma_{pot}(B)))\}) \\ &= \alpha(\{\rho \text{ regular} \mid \rho \in \gamma_{pot}(B)\}) && \text{by Lemma 8.2.2} \\ &= \alpha(\gamma_{oct}(B)) \end{aligned}$$

We now prove that $\sharp\mathcal{S}(\alpha(\gamma_{pot}(B)))$ is strongly closed. It suffices to prove that $\alpha(\gamma_{pot}(B))$ is weakly closed. Being closed, it is clear that it has a null diagonal. Moreover, it is very easy to show that it is coherent, because B is coherent. It remains to prove:

$$\forall uvw,\ \sharp\mathcal{S}(\alpha(\gamma_{pot}(B)))_{uw} \leq \alpha(\gamma_{pot}(B))_{uv} + \alpha(\gamma_{pot}(B))_{vw}$$

But $\alpha(\gamma_{pot}(B))$ is closed. Thus, it verifies the triangular inequality, that in turn implies the desired inequality.

At this stage, the second statement is a straightforward application of Theorem 8.2.4, Lemma 8.2.6 and Theorem 8.2.5. □

Usually, the implementations of the Octagon abstract domain maintain all used DBM strongly closed, so that maximal information is known when entering an abstract operation. However, this breaks the sparsity of the matrix. Indeed, the matrix elements of the form $B_{u\overline{u}}$ are non-relational interval bounds on the program variables: as we expect many variables to be bounded, the strengthening step gives finite bounds for many DBM cells, and a strengthened DBM loses most of the sparsity. In general, a DBM has a quadratic size in the number of variables, and therefore this loss of sparsity is very costly.

We propose to avoid the strengthening step: instead of maintaining the invariant that all the manipulated DBMs are strongly closed, we maintain the invariant that they are *weakly* closed. Some abstract operators need to be adapted: in order to make sure we do not lose precision, we will prove for each of those operators that it computes abstract values with the same concretization as with the usual algorithms. Equivalently, we prove that the strengthening of the abstract value computed by our operator is equal to the abstract value computed by the usual operator on the strengthened parameters.

This notion of weak closedness has been introduced by Bagnara et al. [BHZ09, Appendix A] as an intermediate notion for proving the correctness of the tight closure algorithm (see Section 8.2.8). To the best of our knowledge, the use of the notion of weak closedness as an invariant for manipulating sparse DBMs is an original result of our work.

8.2.3. Comparing Difference Bound Matrices

In order to use octagons in our analyzer, we need to define a comparison operator, taking two DBMs and returning a Boolean. If this Boolean is `true`, then we have the guarantee that the concretization of the first operand is included in that of the second operand.

A good candidate is, of course, $\overset{\sharp}{\leq}$, the natural order relation between DBMs. Its soundness is guaranteed by the monotonicity of γ_{oct} and γ_{pot}. In usual implementations of the Octagon abstract domain, DBMs are kept strongly closed, so that this operator is actually as precise as possible: it returns `true` if and only if the concretizations are included:

Theorem 8.2.8. *Let A be a closed (respectively strongly closed) DBM, and B be any DBM. The two following statements are equivalent:*

(i) $\gamma_{pot}(A) \subseteq \gamma_{pot}(B)$ *(respectively* $\gamma_{oct}(A) \subseteq \gamma_{oct}(B)$*)*

(ii) $A \overset{\sharp}{\leq} B$

Proof. We give the proof only for γ_{pot}: the proof for γ_{oct} is analogous.

(ii) ⇒ (i) cf. Lemma 8.2.1.

(i) ⇒ (ii) We assume $\gamma_{pot}(A) \subseteq \gamma_{pot}(B)$.

By monotonicity of α we have $\alpha(\gamma_{pot}(A)) \overset{\sharp}{\leq} \alpha(\gamma_{pot}(B))$, and, by minimality of $\alpha(\gamma_{pot}(B))$, we have $\alpha(\gamma_{pot}(B)) \overset{\sharp}{\leq} B$. We conclude $\alpha(\gamma_{pot}(A)) \overset{\sharp}{\leq} B$.

Moreover, by Theorem 8.2.4, $A = \alpha(\gamma_{pot}(A))$, so $A \overset{\sharp}{\leq} B$. □

In the setting of weakly closed DBMs, this theorem does not hold, however. In order not to lose precision while still using sparse DBMs, we need another comparison operator that strengthens the bounds of the left operand when they do not entail the right operand:

Definition 8.2.6 (Weakly closed comparison)**.** *Let A be a weakly closed DBM and B be any DBM. The weakly closed comparison of A and B, noted $A \,{}^\sharp{\leq}_{weak}\, B$ is defined by:*

$$A \,{}^\sharp{\leq}_{weak}\, B \equiv \bigwedge_{\substack{u,v\in\mathbb{V}_\pm \\ B_{uv}<+\infty}} A_{uv} \leq B_{uv} \;\vee\; \frac{A_{u\overline{u}} + A_{\overline{v}v}}{2} \leq B_{uv}$$

That is, for every finite bound on B, we first check whether it is directly entailed by the corresponding bound in A, and then try to entail it using non-relational bounds. It exploits sparsity, since only finite bounds in B are considered. The following theorem states that it implements the comparison on concretizations, hence we can use it in a sparse context without losing precision:

Theorem 8.2.9. *Let A be a weakly closed DBM and B any DBM. The two following statements are equivalent:*

(i) $\gamma_{oct}(A) \subseteq \gamma_{oct}(B)$

(ii) $A \,{}^\sharp{\leq}_{weak}\, B$

Proof. Because A is weakly closed, ${}^\sharp\mathcal{S}(A)$ is strongly closed. Then:

$$\begin{aligned}
A \,{}^\sharp{\leq}_{weak}\, B &\equiv \bigwedge_{\substack{u,v\in\mathbb{V}_\pm \\ B_{uv}<+\infty}} A_{uv} \leq B_{uv} \;\vee\; \frac{A_{u\overline{u}} + A_{\overline{v}v}}{2} \leq B_{uv} \\
&\iff {}^\sharp\mathcal{S}(A) \,{}^\sharp{\leq}\, B \\
&\iff \gamma_{oct}({}^\sharp\mathcal{S}(A)) \subseteq \gamma_{oct}(B) && \text{by Theorem 8.2.8} \\
&\iff \gamma_{oct}(A) \subseteq \gamma_{oct}(B) && \text{by Lemma 8.2.1}
\end{aligned}$$

□

8.2.4. Forgetting Variables

An important operation provided by abstract domains is "forget a variable". When given a DBM and a variable v, it returns another DBM where all the information on v has been forgotten. It exists in two versions: when considering regular environments and when considering irregular environments. Before defining the abstract operations themselves, let us define formally the concrete operations they approximate:

Definition 8.2.7 (Concrete forgetting)**.** *1. Let $x \in \mathbb{V}_\pm$ and S be a set of irregular environments. We define:*

$$\mathcal{F}^x_{pot}(S) = \{\rho + [x \Rightarrow r] \mid \rho \in S, r \in \mathbb{R}\}$$

2. Let $x \in \mathbb{V}_+$ and S be a set of regular environments. We define:

$$\mathcal{F}^x_{oct}(S) = \{\rho + [x \Rightarrow r; \overline{x} \Rightarrow -r] \mid \rho \in S, r \in \mathbb{R}\}$$

Their abstract counterpart are implemented by:

Definition 8.2.8 (Abstract forgetting). *1. Let $x \in \mathbb{V}_\pm$ and B be a DBM. We define ${}^\sharp\mathcal{F}^x_{pot}(B)$ the DBM such that:*

$$ {}^\sharp\mathcal{F}^x_{pot}(B)_{uv} = \begin{cases} 0 & \text{if } u = v = x \\ +\infty & \text{otherwise if } u = x \text{ or } v = x \\ B_{uv} & \text{otherwise} \end{cases} $$

2. Let $x \in \mathbb{V}_+$ and B be a DBM. We define:

$$ {}^\sharp\mathcal{F}^x_{oct}(B) = {}^\sharp\mathcal{F}^x_{pot}({}^\sharp\mathcal{F}^{\overline{x}}_{pot}(B)) $$

The following theorem guarantees the soundness of ${}^\sharp\mathcal{F}^x_{pot}$ and ${}^\sharp\mathcal{F}^x_{oct}$. Moreover, it shows that, when applied to closed and strongly closed DBMs, these abstract operators are exact (i.e., they do not approximate the concrete operator, but return exactly the result of the concrete operation; the only approximation resides in the input DBM), and preserve closure and strong closure:

Theorem 8.2.10. *Let B be a DBM and $x \in \mathbb{V}_\pm$. We have:*

1. $\mathcal{F}^x_{pot}(\gamma_{pot}(B)) \subseteq \gamma_{pot}({}^\sharp\mathcal{F}^x_{pot}(B))$
2. *If B is closed, then* $\mathcal{F}^x_{pot}(\gamma_{pot}(B)) = \gamma_{pot}({}^\sharp\mathcal{F}^x_{pot}(B))$.
3. *If B is closed, so is* ${}^\sharp\mathcal{F}^x_{pot}(B)$.

Moreover, if $x \in \mathbb{V}_+$, we have:

4. $\mathcal{F}^x_{oct}(\gamma_{oct}(B)) \subseteq \gamma_{oct}({}^\sharp\mathcal{F}^x_{oct}(B))$
5. *If B is strongly closed, then* $\mathcal{F}^x_{oct}(\gamma_{oct}(B)) = \gamma_{oct}({}^\sharp\mathcal{F}^x_{oct}(B))$.
6. *If B is strongly closed, so is* ${}^\sharp\mathcal{F}^x_{oct}(B)$.

Proof. See, e.g., [Min04, Theorems 3.6.1 and 4.4.2]. □

To these properties, we add similar properties for weak closedness, that let us use it as-is for weakly closed DBMs without loss of precision:

Theorem 8.2.11. *Let B be a weakly closed DBM and $x \in \mathbb{V}_+$. We have:*

1. ${}^\sharp\mathcal{S}({}^\sharp\mathcal{F}^x_{oct}(B)) = {}^\sharp\mathcal{F}^x_{oct}({}^\sharp\mathcal{S}(B))$
2. $\mathcal{F}^x_{oct}(\gamma_{oct}(B)) = \gamma_{oct}({}^\sharp\mathcal{F}^x_{oct}(B))$
3. ${}^\sharp\mathcal{F}^x_{oct}(B)$ *is weakly closed*

Proof. 1. Let $u, v \in \mathbb{V}_\pm$. We distinguish the following cases:

- If $u, v \notin \{x, \overline{x}\}$, we clearly have ${}^\sharp\mathcal{S}({}^\sharp\mathcal{F}^x_{oct}(B))_{uv} = {}^\sharp\mathcal{F}^x_{oct}({}^\sharp\mathcal{S}(B))_{uv}$.
- If $u \in \{x, \overline{x}\}$ and $v \notin \{x, \overline{x}\}$ (or symmetrically), or $u = \overline{v} \in \{x, \overline{x}\}$:

$$ {}^\sharp\mathcal{S}({}^\sharp\mathcal{F}^x_{oct}(B))_{uv} = +\infty = {}^\sharp\mathcal{F}^x_{oct}({}^\sharp\mathcal{S}(B))_{uv} $$

- If $u = v \in \{x, \overline{x}\}$:

$$ {}^\sharp\mathcal{S}({}^\sharp\mathcal{F}^x_{oct}(B))_{uv} = 0 = {}^\sharp\mathcal{F}^x_{oct}({}^\sharp\mathcal{S}(B))_{uv} $$

2. We have:

$$\begin{aligned}
\mathcal{F}^x_{oct}(\gamma_{oct}(B)) &= \mathcal{F}^x_{oct}(\gamma_{oct}(\overset{\sharp}{\mathcal{S}}(B))) && \text{by Lemma 8.2.6}\\
&= \gamma_{oct}(\overset{\sharp}{\mathcal{F}}{}^x_{oct}(\overset{\sharp}{\mathcal{S}}(B))) && \text{by Theorem 8.2.10 } (B \text{ is weakly closed})\\
&= \gamma_{oct}(\overset{\sharp}{\mathcal{S}}(\overset{\sharp}{\mathcal{F}}{}^x_{oct}(B))) && \text{using 1.}\\
&= \gamma_{oct}(\overset{\sharp}{\mathcal{F}}{}^x_{oct}(B)) && \text{by Lemma 8.2.6}
\end{aligned}$$

3. $\overset{\sharp}{\mathcal{F}}{}^x_{oct}(\overset{\sharp}{\mathcal{S}}(B))$ is strongly closed by Theorem 8.2.10, thus so $\overset{\sharp}{\mathcal{S}}(\overset{\sharp}{\mathcal{F}}{}^x_{oct}(B))$ is (using 1.). Moreover, $\overset{\sharp}{\mathcal{F}}{}^x_{oct}(B)$ has a null diagonal. As a consequence, $\overset{\sharp}{\mathcal{F}}{}^x_{oct}(B)$ is weakly closed. □

8.2.5. Join

The usual join operator on DBMs is defined by the usual least upper bound operator for $\overset{\sharp}{\leq}$:

Definition 8.2.9 (DBM least upper bound)**.** *Let A and B be two DBMs. The least upper bound $\overset{\sharp}{\cup}$ on DBMs is defined by:*

$$\forall uv,\ (A \overset{\sharp}{\cup} B)_{uv} = \max\{A_{uv}\ ;\ B_{uv}\}$$

The order relation $\overset{\sharp}{\leq}$ and the operator $\overset{\sharp}{\cup}$ clearly form an upper semi-lattice, thus usual properties on Galois connections hold, providing nice results on the soundness and precision of this operator:

Theorem 8.2.12. *Let A and B be two DBMs. We have:*

1. $\gamma_{pot}(A) \cup \gamma_{pot}(B) \subseteq \gamma_{pot}(A \overset{\sharp}{\cup} B)$
2. *If A and B are closed, then* $A \overset{\sharp}{\cup} B = \alpha(\gamma_{pot}(A) \cup \gamma_{pot}(B))$
3. *If A and B are closed, so is* $A \overset{\sharp}{\cup} B$
4. $\gamma_{oct}(A) \cup \gamma_{oct}(B) \subseteq \gamma_{oct}(A \overset{\sharp}{\cup} B)$
5. *If A and B are strongly closed, then* $A \overset{\sharp}{\cup} B = \alpha(\gamma_{oct}(A) \cup \gamma_{oct}(B))$
6. *If A and B are strongly closed, so is* $A \overset{\sharp}{\cup} B$

For weakly closed DBMs, even though the fourth point would guarantee the soundness of $\overset{\sharp}{\cup}$, this operator would lose precision when applied to non-strongly closed DBMs. As an example, consider the weakly closed DBM A representing the two following inequalities on positive variables x and y:

$$x + x \leq 1 \qquad\qquad y + y \leq 0$$

The weakly closed DBM B, in turn, represents the two following inequalities:

$$x + x \leq 0 \qquad\qquad y + y \leq 1$$

The inequality $x + y \leq 1/2$ is not present in A nor in B, even though it exists in $\overset{\sharp}{\mathcal{S}}(A)$ and in $\overset{\sharp}{\mathcal{S}}(B)$. As a result, $A \overset{\sharp}{\cup} B$ contains the inequalities $x + x \leq 1$ and $y + y \leq 1$, but would not entail $x + y \leq 1/2$, which is entailed however by $\overset{\sharp}{\mathcal{S}}(A) \overset{\sharp}{\cup} \overset{\sharp}{\mathcal{S}}(B)$.

The rationale behind this example is that a join can create some amount of relationality that was not present in one or both operand. Our operator has to reflect this fact. Care should be taken, however, not to break the sparsity of the operands by introducing spurious finite values in the matrix. Our join for weakly closed DBMs is defined as follows:

Definition 8.2.10 (Weakly closed join for octagons)**.** *Let A and B be two weakly closed DBMs. We take $B^{1/2}_{uv} = \frac{B_{u\overline{u}}+B_{\overline{v}v}}{2}$ and $A^{1/2}_{uv} = \frac{A_{u\overline{u}}+A_{\overline{v}v}}{2}$. The weakly closed join $^\sharp\cup_{weak}$ is defined in two steps:*

1. *We first define $A\,{}^\sharp\cup^0_{weak}\,B$. Let $u, v \in \mathbb{V}_\pm$. We define:*

$$(A\,{}^\sharp\cup^0_{weak}\,B)_{uv} = \begin{cases} A_{uv} & \text{if } A_{uv} = B_{uv} \\ B_{uv} & \text{if } A_{uv} < B_{uv} \leq B^{1/2}_{uv} \\ \max\{A_{uv}\ ;\ B^{1/2}_{uv}\} & \text{if } A_{uv} < B_{uv} \wedge B^{1/2}_{uv} < B_{uv} \\ (B\,{}^\sharp\cup^0_{weak}\,A)_{uv} & \text{if } A_{uv} > B_{uv} \end{cases}$$

2. *Let $u, v \in \mathbb{V}_\pm$. We define:*

$$(A\,{}^\sharp\cup_{weak}\,B)_{uv} = \begin{cases} \min\left\{\begin{array}{c}(A\,{}^\sharp\cup^0_{weak}\,B)_{uv} \\ \max\left\{A^{1/2}_{uv}\ ;\ B^{1/2}_{uv}\right\}\end{array}\right\} & \text{if } A_{u\overline{u}} < B_{u\overline{u}} \wedge A_{\overline{v}v} > B_{\overline{v}v} \\ & \text{or } A_{u\overline{u}} > B_{u\overline{u}} \wedge A_{\overline{v}v} < B_{\overline{v}v} \\ (A\,{}^\sharp\cup^0_{weak}\,B)_{uv} & \text{otherwise} \end{cases}$$

The first step can be computed by iterating over all the matrix elements that are different in A and B. This first step thus preserves the sparsity, and consumes computing time only for variables that are different in both branches. The second step can be computed efficiently by first collecting in a list all the variables u for which $A_{u\overline{u}} < B_{u\overline{u}}$ and, in another list, all those for which $B_{u\overline{u}} < A_{u\overline{u}}$. By iterating over the two lists, we can efficiently modify only the cells meeting the given condition. It should be noted that we break in the second step only the sparsity that needs to be broken, as the modified cells correspond to the cases where the join create new relational information (as in the example above).

The following theorem states that this modified join operator can be used on weakly closed DBMs without loss of precision nor of soundness:

Theorem 8.2.13. *Let A and B be two weakly closed DBMs. We have:*

1. $^\sharp\mathcal{S}(A\,{}^\sharp\cup_{weak}\,B) = {}^\sharp\mathcal{S}(A)\,{}^\sharp\cup\,{}^\sharp\mathcal{S}(B)$
2. $\gamma_{oct}(A\,{}^\sharp\cup_{weak}\,B) = \gamma_{oct}(\alpha(\gamma_{oct}(A) \cup \gamma_{oct}(B)))$
3. *$A\,{}^\sharp\cup_{weak}\,B$ is weakly closed*

Proof. 1. The proof of the theorem is technical. For more confidence in its correctness, we mechanically verified it in Coq. The theorems present in the Verasco main Coq development do not imply this result, which is neither needed nor correct in the actual implementation because of the use of floating-point numbers. Instead, a weaker soundness result is proved in Verasco, and we formally proved this result in a separate development.

We proceed by antisymmetry of $^\sharp\leq$.

- We first prove ${}^\sharp\mathcal{S}(A) \mathbin{{}^\sharp\cup} {}^\sharp\mathcal{S}(B) \mathbin{{}^\sharp\leq} {}^\sharp\mathcal{S}(A \mathbin{{}^\sharp\cup_{weak}} B)$. We have, by Theorem 8.2.12:

$$ {}^\sharp\mathcal{S}(A) \mathbin{{}^\sharp\cup} {}^\sharp\mathcal{S}(B) = \alpha(\gamma_{oct}({}^\sharp\mathcal{S}(A)) \cup \gamma_{oct}({}^\sharp\mathcal{S}(B))) $$

Thus it suffices to prove, by minimality of α:

$$ \gamma_{oct}({}^\sharp\mathcal{S}(A)) \cup \gamma_{oct}({}^\sharp\mathcal{S}(B)) \subseteq \gamma_{oct}({}^\sharp\mathcal{S}(A \mathbin{{}^\sharp\cup_{weak}} B)) $$

By Lemma 8.2.6, this is equivalent to the following soundness statement:

$$ \gamma_{oct}(A) \cup \gamma_{oct}(B) \subseteq \gamma_{oct}(A \mathbin{{}^\sharp\cup_{weak}} B) $$

So, let $\rho \in \gamma_{oct}(A) \cup \gamma_{oct}(B)$. We first prove that $\rho \in \gamma_{oct}(A \mathbin{{}^\sharp\cup^0_{weak}} B)$. Let $u, v \in \mathbb{V}_\pm$. Without loss of generality, we can assume $A_{uv} \leq B_{uv}$, so $\rho(u) - \rho(v) \leq B_{uv}$. We distinguish several cases:

  - If $A_{uv} = B_{uv}$, then $\rho(u) - \rho(v) \leq B_{uv} = A_{uv} = (A \mathbin{{}^\sharp\cup^0_{weak}} B)_{uv}$
  - If $A_{uv} < B_{uv} \leq B^{1/2}_{uv}$, then $\rho(u) - \rho(v) \leq B_{uv} = (A \mathbin{{}^\sharp\cup^0_{weak}} B)_{uv}$
  - If $A_{uv} < B_{uv} \wedge B^{1/2}_{uv} < B_{uv}$ and $\rho \in \gamma_{oct}(A)$, we have:

$$ \rho(u) - \rho(v) \leq A_{uv} \leq \max\{A_{uv} \ ; \ B^{1/2}_{uv}\} = (A \mathbin{{}^\sharp\cup^0_{weak}} B)_{uv} $$

  - If $A_{uv} < B_{uv} \wedge B^{1/2}_{uv} < B_{uv}$ and $\rho \in \gamma_{oct}(B)$, we see that, since ρ is regular:

$$ \rho(u) - \rho(v) \leq B^{1/2}_{uv} \leq \max\{A_{uv} \ ; \ B^{1/2}_{uv}\} = (A \mathbin{{}^\sharp\cup^0_{weak}} B)_{uv} $$

We now continue by proving $\rho \in \gamma_{oct}(A \mathbin{{}^\sharp\cup_{weak}} B)$. Let $u, v \in \mathbb{V}_\pm$. Given that $\rho \in \gamma_{oct}(A \mathbin{{}^\sharp\cup^0_{weak}} B)$, it suffices to prove that $\rho(u) - \rho(v) \leq \max\left\{A^{1/2}_{uv} \ ; \ B^{1/2}_{uv}\right\}$. Without loss of generality, we assume $\rho \in \gamma_{oct}(A)$. We have $\rho(u) - \rho(v) \leq A^{1/2}_{uv} \leq \max\left\{A^{1/2}_{uv} \ ; \ B^{1/2}_{uv}\right\}$.

- Now, we prove ${}^\sharp\mathcal{S}(A \mathbin{{}^\sharp\cup_{weak}} B) \mathbin{{}^\sharp\leq} \mathcal{S}(A) \mathbin{{}^\sharp\cup} {}^\sharp\mathcal{S}(B)$. Let $u, v \in \mathcal{V}_\pm$. Without loss of generality, we assume $A_{uv} \leq B_{uv}$, and we distinguish several cases:
  - If $B_{uv} \leq B^{1/2}_{uv}$, then:

$$ {}^\sharp\mathcal{S}(A \mathbin{{}^\sharp\cup_{weak}} B)_{uv} \leq (A \mathbin{{}^\sharp\cup_{weak}} B)_{uv} \leq (A \mathbin{{}^\sharp\cup^0_{weak}} B)_{uv} = B_{uv} $$
$$ B_{uv} \leq \max\{{}^\sharp\mathcal{S}(A)_{uv} \ ; \ B_{uv}\} = ({}^\sharp\mathcal{S}(A) \mathbin{{}^\sharp\cup} {}^\sharp\mathcal{S}(B))_{uv} $$

  - If $A_{uv} \leq A^{1/2}_{uv}$ and $B_{uv} > B^{1/2}_{uv}$, then we remark that, independently of whether $A_{uv} = B_{uv}$ or not, $(A \mathbin{{}^\sharp\cup^0_{weak}} B)_{uv} = \max\{A_{uv} \ ; \ B^{1/2}_{uv}\}$. Therefore:

$$ ({}^\sharp\mathcal{S}(A \mathbin{{}^\sharp\cup_{weak}} B))_{uv} \leq (A \mathbin{{}^\sharp\cup^0_{weak}} B)_{uv} = \max\{A_{uv} \ ; \ B^{1/2}_{uv}\} $$
$$ = ({}^\sharp\mathcal{S}(A) \mathbin{{}^\sharp\cup} {}^\sharp\mathcal{S}(B))_{uv} $$

  - If $A_{uv} > A^{1/2}_{uv}$ and $B_{uv} > B^{1/2}_{uv}$, we distinguish again two cases:
    * Either $\max\{A^{1/2}_{uv} \ ; \ B^{1/2}_{uv}\} = \frac{\max\{A_{u\overline{u}}; B_{u\overline{u}}\} + \max\{A_{\overline{v}v}; B_{\overline{v}v}\}}{2}$, then, we remark that, whatever the case, $(A \mathbin{{}^\sharp\cup_{weak}} B)_{u\overline{u}} = \max\{A_{u\overline{u}}; B_{u\overline{u}}\}$ (and similarly

for v). We deduce:

$$(\sharp\mathcal{S}(A \,\sharp\cup_{weak}\, B))_{uv} \leq \frac{(A \,\sharp\cup_{weak}\, B)_{u\overline{u}} + (A \,\sharp\cup_{weak}\, B)_{\overline{v}v}}{2}$$
$$= \max\{A_{uv}^{1/2}\ ;\ B_{uv}^{1/2}\} = (\sharp\mathcal{S}(A) \,\sharp\cup\, \sharp\mathcal{S}(B))_{uv}$$

* Otherwise, a bit of reasoning leads to $A_{u\overline{u}} < B_{u\overline{u}} \wedge A_{\overline{v}v} > B_{\overline{v}v}$ or $A_{u\overline{u}} > B_{u\overline{u}} \wedge A_{\overline{v}v} < B_{\overline{v}v}$. This is the case where:

$$(\sharp\mathcal{S}(A \,\sharp\cup_{weak}\, B))_{uv} \leq \max\{A_{uv}^{1/2}\ ;\ B_{uv}^{1/2}\} = (\sharp\mathcal{S}(A) \,\sharp\cup\, \sharp\mathcal{S}(B))_{uv}$$

2. We have :

$$\begin{aligned}
\gamma_{oct}(A \,\sharp\cup_{weak}\, B) &= \gamma_{oct}(\sharp\mathcal{S}(A \,\sharp\cup_{weak}\, B)) && \text{by Lemma 8.2.6}\\
&= \gamma_{oct}(\sharp\mathcal{S}(A) \,\sharp\cup\, \sharp\mathcal{S}(B)) && \text{using 1.}\\
&= \gamma_{oct}(\alpha(\gamma_{oct}(A) \cup \gamma_{oct}(B))) && \text{by Theorem 8.2.12}
\end{aligned}$$

3. Using 1. and Theorem 8.2.12, we have that $\sharp\mathcal{S}(A \,\sharp\cup_{weak}\, B) = \sharp\mathcal{S}(A) \,\sharp\cup\, \sharp\mathcal{S}(B)$ is strongly closed. Moreover, because A is weakly closed, we have $\forall u, 0 \leq A_{u\overline{u}} + A_{\overline{u}u}$, and similarly for B. From that, it is easy to derive that $A \,\sharp\cup_{weak}\, B$ has a null diagonal, and therefore it is weakly closed.

□

8.2.6. Assuming Constraints

An important operation for abstract domains is the `assume` primitive. In Verasco, it is implemented using `Fact_msg` messages, that refine abstract environments using the fact that a given expression evaluates to a given value. In this section, we only consider the cases where this operation is exact, i.e., it does not lead to any approximation. These cases amount to assuming that $\rho(x) - \rho(y) \leq C$, for $C \in \mathbb{R}$ and x and y two variables. Again, to simplify the notations, we give it in two versions: one adapted to γ_{pot}, and one adapted to γ_{oct}.

Similarly to forgetting, we first give the concrete semantics of this operation, which is the same for irregular and regular environments:

Definition 8.2.11 (Assuming constraints in the concrete)**.** *Let $C \in \mathbb{R}$, $x, y \in \mathbb{V}_{\pm}$ and S be a set of irregular environments. We define:*

$$\mathcal{A}^{x-y\leq C}(S) = \{\rho \in S \mid \rho(x) - \rho(y) \leq C\}$$

It is easy to see that we can reflect exactly this operation in DBMs. Indeed, it suffices to change the cell corresponding to the new constraint in the DBM, if the old value is larger than the new one. However, this does not maintain any kind of closedness, whether it be the normal closure, the strong closure or the weak closedness. As a result, it is necessary to run a closure algorithm just after inserting the new constraint. Usually these algorithms are costly (i.e., cubic complexity), and do not leverage the fact that the input matrix is already almost closed. For this reason, *incremental closure algorithms* have been developed, with quadratic worst case complexity. We give here a slightly different presentation of these algorithms to the one originally given by Miné [Min04]:

Definition 8.2.12 (Assuming constraints in the abstract)**.** *Let $C \in \mathbb{R}$, B be a DBM and $x, y \in \mathbb{V}_{\pm}$.*

1. *We define $\sharp\mathcal{A}_{pot}^{x-y\leq C}(B)$ the DBM such that, for $u, v \in \mathbb{V}_{\pm}$:*

$$\sharp\mathcal{A}_{pot}^{x-y\leq C}(B)_{uv} = \min\{B_{uv} \ ; \ B_{ux} + C + B_{yv}\}$$

2. *If $x, y \in \mathbb{V}_{+}$, we define $\sharp\mathcal{A}_{weak}^{x-y\leq C}(B)$ and $\sharp\mathcal{A}_{oct}^{x-y\leq C}(B)$ as:*

$$\sharp\mathcal{A}_{weak}^{x-y\leq C}(B) = \sharp\mathcal{A}_{pot}^{\overline{y}-\overline{x}\leq C}(\sharp\mathcal{A}_{pot}^{x-y\leq C}(B))$$
$$\sharp\mathcal{A}_{oct}^{x-y\leq C}(B) = \sharp\mathcal{S}(\sharp\mathcal{A}_{weak}^{x-y\leq C}(B))$$

The following theorem states their soundness and exactness in the context of closed and strongly closed DBMs. It also provides an easy criterion for determining whether the new constraint is contradictory:

Theorem 8.2.14. *Let $C \in \mathbb{R}$, B be a DBM and $x, y \in \mathbb{V}_{\pm}$. We have:*

1. $\mathcal{A}^{x-y\leq C}(\gamma_{pot}(B)) \subseteq \gamma_{pot}(\sharp\mathcal{A}_{pot}^{x-y\leq C}(B))$
2. *If B has a null diagonal (or, a fortiori, is closed), then:*

$$\gamma_{pot}(\sharp\mathcal{A}_{pot}^{x-y\leq C}(B)) = \mathcal{A}^{x-y\leq C}(\gamma_{pot}(B))$$

3. *If B is closed, the following statements are equivalent:*
   (i) $\mathcal{A}^{x-y\leq C}(\gamma_{pot}(B)) \neq \emptyset$
   (ii) $0 \leq \sharp\mathcal{A}_{pot}^{x-y\leq C}(B)_{xx}$
   (iii) $0 \leq C + B_{yx}$
   (iv) $\sharp\mathcal{A}_{pot}^{x-y\leq C}(B)$ *is closed.*

Moreover, if $x, y \in \mathbb{V}_{+}$, we have:

4. $\mathcal{A}^{x-y\leq C}(\gamma_{oct}(B)) \subseteq \gamma_{oct}(\sharp\mathcal{A}_{oct}^{x-y\leq C}(B))$
5. *If B has a null diagonal (or, a fortiori, is strongly closed), then:*

$$\gamma_{oct}(\sharp\mathcal{A}_{oct}^{x-y\leq C}(B)) = \mathcal{A}^{x-y\leq C}(\gamma_{oct}(B))$$

6. *If B is strongly closed, the following statements are equivalent:*
   (i) $\mathcal{A}^{x-y\leq C}(\gamma_{oct}(B)) \neq \emptyset$
   (ii) $0 \leq \sharp\mathcal{A}_{oct}^{x-y\leq C}(B)_{xx}$
   (iii) $0 \leq C + B_{yx}$
   (iv) $\sharp\mathcal{A}_{oct}^{x-y\leq C}(B)$ *is strongly closed.*

Proof. We gave a slightly different presentation of the abstract transfer function as the one present in the literature. To convince the reader this presentation is equivalent to others, we give here the full proof of these results.

1. Let $\rho \in \mathcal{A}^{x-y\leq C}(\gamma_{pot}(B))$ and $u, v \in \mathbb{V}_{\pm}$. We have:

$$\begin{aligned}\rho(u) - \rho(v) &= \rho(u) - \rho(x) + \rho(x) - \rho(y) + \rho(y) - \rho(v) \leq B_{ux} + C + B_{yv}\\ \rho(u) - \rho(v) &\leq B_{uv}\end{aligned}$$

   So, $\rho(u) - \rho(v) \leq \sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{uv}$, and $\rho \in \gamma_{pot}(\sharp\mathcal{A}^{x-y\leq C}_{pot}(B))$.

2. Using 1., it suffices to prove $\gamma_{pot}(\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)) \subseteq \mathcal{A}^{x-y\leq C}(\gamma_{pot}(B))$. To that end, let $\rho \in \gamma_{pot}(\sharp\mathcal{A}^{x-y\leq C}_{pot}(B))$. Since $\sharp\mathcal{A}^{x-y\leq C}_{pot}(B) \stackrel{\sharp}{\leq} B$, we have $\rho \in \gamma_{pot}(B)$. It remains to prove $\rho(x) - \rho(y) \leq C$.

   We have $B_{xx} = B_{yy} = 0$, so $\rho(x) - \rho(y) \leq \sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{xy} \leq 0 + C + 0 = C$.

3. (i) $\Rightarrow$ (ii) follows directly from 2. and the definitions.

   (ii) $\Rightarrow$ (iii) We have:

$$0 \leq \sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{xx} \leq B_{xx} + C + B_{yx}$$

   With $B_{xx} = 0$ because B is closed.

   (iii) $\Rightarrow$ (iv) We first prove that $\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)$ has a null diagonal: let $u \in \mathbb{V}_{\pm}$. We have:

$$\begin{aligned}\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{uu} &= \min\{B_{uu}\ ;\ B_{ux} + C + B_{yu}\}\\ B_{uu} = 0 \qquad & 0 \leq C + B_{yx} \leq B_{ux} + C + B_{yu}\end{aligned}$$

   Thus $\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{uu} = 0$.

   We now prove that $\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)$ verifies the triangular equality. Let $u, v, w \in \mathbb{V}_{\pm}$, we consider several cases:

   - If $\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{uv} = B_{uv}$ and $\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{vw} = B_{vw}$, we have:

$$\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{uw} \leq B_{uw} \leq B_{uv} + B_{vw} = \sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{uv} + \sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{vw}$$

   - If $\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{uv} = B_{uv}$ and $\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{vw} = B_{vx} + C + B_{yw}$, we have:

$$\begin{aligned}\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{uw} \leq B_{ux} + C + B_{yw} &\leq B_{uv} + B_{vx} + C + B_{yw}\\ &= \sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{uv} + \sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{vw}\end{aligned}$$

   - The case $\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{uv} = B_{ux} + C + B_{yv}$ and $\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{vw} = B_{vw}$ is treated similarly to the previous one.
   - If $\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{uv} = B_{ux} + C + B_{yv}$ and $\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{vw} = B_{vx} + C + B_{yw}$, then:

$$\begin{aligned}\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{uw} &\leq B_{ux} + C + B_{yw} \leq B_{ux} + C + B_{yx} + C + B_{yw}\\ &\leq B_{ux} + C + B_{yv} + B_{vx} + C + B_{yw}\\ &= \sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{uv} + \sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{vw}\end{aligned}$$

   (iv) $\Rightarrow$ (i) follows from Theorem 8.2.4 and 2.

4. Elements of $\rho \in A^{x-y\leq C}(\gamma_{oct}(B))$ are all regular, so they verify $\rho(\overline{y}) - \rho(\overline{x}) = \rho(x) - \rho(y) \leq C$. We deduce:

$$\begin{aligned}\mathcal{A}^{x-y\leq C}(\gamma_{oct}(B)) &\subseteq \mathcal{A}^{\overline{y}-\overline{x}\leq C}(\mathcal{A}^{x-y\leq C}(\gamma_{pot}(B))) \\ &\subseteq \gamma_{pot}(\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)) \qquad \text{by 1.}\end{aligned}$$

   Moreover, elements of $\mathcal{A}^{x-y\leq C}(\gamma_{oct}(B))$ are regular, therefore:

$$\mathcal{A}^{x-y\leq C}(\gamma_{oct}(B)) \subseteq \gamma_{oct}(\sharp\mathcal{A}^{x-y\leq C}_{weak}(B))$$

   We conclude by using Lemma 8.2.6.

5. By using 2. and Lemma 8.2.6, we get:

$$\gamma_{oct}(\sharp\mathcal{A}^{x-y\leq C}_{oct}(B)) \subseteq \mathcal{A}^{\overline{y}-\overline{x}\leq C}(\mathcal{A}^{x-y\leq C}(\gamma_{pot}(B))) \subseteq \mathcal{A}^{x-y\leq C}(\gamma_{pot}(B))$$

   All the elements of $\gamma_{oct}(\sharp\mathcal{A}^{x-y\leq C}_{oct}(B))$ are regular, so we also have:

$$\gamma_{oct}(\sharp\mathcal{A}^{x-y\leq C}_{oct}(B)) \subseteq \mathcal{A}^{x-y\leq C}(\gamma_{oct}(B))$$

   We conclude by using 4.

6. (i) $\Rightarrow$ (ii), (ii) $\Rightarrow$ (iii) and (iv) $\Rightarrow$ (i) are proved similarly as in 3. It remains to prove (iii) $\Rightarrow$ (iv).

   Thus, we assume $0 \leq C + B_{yx}$ and seek to prove that $\sharp\mathcal{A}^{x-y\leq C}_{oct}(B)$ is strongly closed. By applying 3., we deduce that $\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)$ is closed. Moreover, because B is strongly closed:

$$B_{\overline{x}x} + C + B_{y\overline{y}} \geq C + 2B_{yx} \qquad\qquad B_{\overline{x}\,\overline{y}} = B_{yx}$$

   Combining these identities and the definition of $\sharp\mathcal{A}^{x-y\leq C}_{pot}(B)$, we deduce $0 \leq C + \sharp\mathcal{A}^{x-y\leq C}_{pot}(B)_{\overline{x}\,\overline{y}}$, so we can apply 3. again: $\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)$ is closed. We now prove that it is also coherent. By definition, for $u, v \in \mathbb{V}_{\pm}$, we have:

$$\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)_{uv} = \min\left\{\begin{array}{c} B_{uv} \;;\; B_{ux} + C + B_{yv} \;;\; B_{u\overline{y}} + C + B_{\overline{x}v} \\ B_{u\overline{y}} + C + B_{\overline{x}x} + C + B_{yv} \\ B_{ux} + C + B_{y\overline{y}} + C + B_{\overline{x}v} \\ B_{u\overline{y}} + C + B_{\overline{x}x} + C + B_{y\overline{y}} + C + B_{\overline{x}v} \end{array}\right\}$$

   But $0 \leq 2C + B_{\overline{x}x} + B_{y\overline{y}}$, because B is strongly closed and $0 \leq C + B_{yx}$. So, the last parameter of the minimum is useless. The rest of the expression is symmetric, so the coherence of $\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)$ follows from that of B. Theorem 8.2.7 finishes the proof.

□

The theorem above implies in particular, that when applied to weakly closed DBMs, $\sharp\mathcal{A}^{x-y\leq C}_{oct}$ is sound and exact, since weakly closed DBMs have null diagonals. However, because this operator uses $\sharp\mathcal{S}$, it breaks sparsity. The advantage of using weakly closed DBMs is that, in the setting of weakly closed DBMs, $\sharp\mathcal{S}$ is no longer needed: $\sharp\mathcal{A}^{x-y\leq C}_{weak}$ can be used as-is without strengthening. The following theorem summarizes this result, and justifies the use of this transfer function in the context of sparse DBMs without loss of precision:

Theorem 8.2.15. *Let $C \in \mathbb{R}$, B be a weakly closed DBM and $x, y \in \mathbb{V}_+$. We have:*

1. *If $0 \leq 2C + B_{y\overline{y}} + B_{\overline{x}x}$, then ${}^\sharp\mathcal{S}({}^\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)) = {}^\sharp\mathcal{A}^{x-y\leq C}_{oct}({}^\sharp\mathcal{S}(B))$*
2. *$\gamma_{oct}({}^\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)) = \mathcal{A}^{x-y\leq C}(\gamma_{oct}(B))$*
3. *If B is weakly closed, the following statements are equivalent:*
   - *(i) $\mathcal{A}^{x-y\leq C}(\gamma_{oct}(B)) \neq \emptyset$*
   - *(ii) $0 \leq {}^\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)_{xx}$*
   - *(iii) $0 \leq C + B_{yx}$ and $0 \leq 2C + B_{y\overline{y}} + B_{\overline{x}x}$*
   - *(iv) ${}^\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)$ is weakly closed.*

Proof. 1. Similarly to Definition 8.2.5 and Theorem 8.2.13, this technical result has been formally verified in Coq outside of the main Verasco development.

Because ${}^\sharp\mathcal{A}^{x-y\leq C}_{oct}$ is monotonic and ${}^\sharp\mathcal{S}(B) \mathrel{{}^\sharp\leq} B$, it follows that:

$$ {}^\sharp\mathcal{A}^{x-y\leq C}_{oct}({}^\sharp\mathcal{S}(B)) \mathrel{{}^\sharp\leq} {}^\sharp\mathcal{A}^{x-y\leq C}_{oct}(B) = {}^\sharp\mathcal{S}({}^\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)) $$

It remains to prove ${}^\sharp\mathcal{S}({}^\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)) \mathrel{{}^\sharp\leq} {}^\sharp\mathcal{A}^{x-y\leq C}_{oct}({}^\sharp\mathcal{S}(B))$.

We first reduce to the case where B is coherent: let $\tilde{B}_{uv} = \max\{B_{uv} \ ; \ B_{\overline{v}\,\overline{u}}\}$. $\tilde{B}$ is clearly coherent. We prove that $\tilde{B}$ verifies the same hypotheses than B (i.e., it is weakly closed and $0 \leq 2C + \tilde{B}_{y\overline{y}} + \tilde{B}_{\overline{x}x}$), and that if the inequality holds fold $\tilde{B}$, then it holds for B.

We prove here that $\tilde{B}$ is weakly closed. By using the distributivity of min over max, we have that $\forall u, v \in \mathbb{V}_\pm$, ${}^\sharp\mathcal{S}(\tilde{B})_{uv} = \max\{{}^\sharp\mathcal{S}(B)_{uv} \ ; \ {}^\sharp\mathcal{S}(B)_{\overline{v}\,\overline{u}}\}$. Because ${}^\sharp\mathcal{S}(B)_{uv}$ is coherent, we get ${}^\sharp\mathcal{S}(\tilde{B}) = {}^\sharp\mathcal{S}(B)$. So, ${}^\sharp\mathcal{S}(\tilde{B})$ is strongly closed. Moreover, it is clear that $\tilde{B}$ has a null diagonal, so it is weakly closed.

We have $B_{\overline{x}x} = \tilde{B}_{\overline{x}x}$ and $B_{y\overline{y}} = \tilde{B}_{y\overline{y}}$, so $0 \leq 2C + \tilde{B}_{y\overline{y}} + \tilde{B}_{\overline{x}x}$.

We assume ${}^\sharp\mathcal{S}({}^\sharp\mathcal{A}^{x-y\leq C}_{weak}(\tilde{B})) \mathrel{{}^\sharp\leq} {}^\sharp\mathcal{A}^{x-y\leq C}_{oct}({}^\sharp\mathcal{S}(\tilde{B}))$, and seek to prove the same inequality on B. We get:

$$ {}^\sharp\mathcal{S}({}^\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)) \mathrel{{}^\sharp\leq} {}^\sharp\mathcal{S}({}^\sharp\mathcal{A}^{x-y\leq C}_{weak}(\tilde{B})) \mathrel{{}^\sharp\leq} {}^\sharp\mathcal{A}^{x-y\leq C}_{oct}({}^\sharp\mathcal{S}(\tilde{B})) = {}^\sharp\mathcal{A}^{x-y\leq C}_{oct}({}^\sharp\mathcal{S}(B)) $$

So, the same equality on B holds.

Thus, we can assume that B is coherent.

We now prove that it suffices to prove:

$$ {}^\sharp\mathcal{S}({}^\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)) \mathrel{{}^\sharp\leq} {}^\sharp\mathcal{A}^{x-y\leq C}_{weak}({}^\sharp\mathcal{S}(B)) $$

to show the desired equality ${}^\sharp\mathcal{S}({}^\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)) \mathrel{{}^\sharp\leq} {}^\sharp\mathcal{A}^{x-y\leq C}_{oct}({}^\sharp\mathcal{S}(B))$ holds. We remark that ${}^\sharp\mathcal{S}$ is an idempotent operator, so:

$$ \begin{aligned} {}^\sharp\mathcal{S}({}^\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)) &= {}^\sharp\mathcal{S}({}^\sharp\mathcal{S}({}^\sharp\mathcal{A}^{x-y\leq C}_{weak}(B))) && \text{by idempotence of } {}^\sharp\mathcal{S} \\ &\mathrel{{}^\sharp\leq} {}^\sharp\mathcal{S}({}^\sharp\mathcal{A}^{x-y\leq C}_{weak}({}^\sharp\mathcal{S}(B))) && \text{by using our hypothesis} \\ &= {}^\sharp\mathcal{A}^{x-y\leq C}_{oct}({}^\sharp\mathcal{S}(B)) \end{aligned} $$

Thus, we now seek to prove $\sharp\mathcal{S}(\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)) \sharp\leq \sharp\mathcal{A}^{x-y\leq C}_{weak}(\sharp\mathcal{S}(B))$. Let $u, v \in \mathbb{V}_{\pm}$. We have, by unfolding the definitions and doing some arithmetic simplifications:

$$\sharp\mathcal{A}^{x-y\leq C}_{weak}(\sharp\mathcal{S}(B))_{uv} = \min\left\{\begin{array}{c} \sharp\mathcal{S}(B)_{uv} \\ \sharp\mathcal{S}(B)_{ux} + C + \sharp\mathcal{S}(B)_{yv} \ ; \ \sharp\mathcal{S}(B)_{u\overline{y}} + C + \sharp\mathcal{S}(B)_{\overline{x}v} \\ \sharp\mathcal{S}(B)_{u\overline{y}} + C + B_{\overline{x}x} + C + \sharp\mathcal{S}(B)_{yv} \\ \sharp\mathcal{S}(B)_{ux} + C + B_{y\overline{y}} + C + \sharp\mathcal{S}(B)_{\overline{x}v} \\ \sharp\mathcal{S}(B)_{u\overline{y}} + C + B_{\overline{x}x} + C + B_{y\overline{y}} + C + \sharp\mathcal{S}(B)_{\overline{x}v} \end{array}\right\}$$

By using $0 \leq 2C + B_{y\overline{y}} + B_{\overline{x}x}$, we can further simplify this expression into:

$$\sharp\mathcal{A}^{x-y\leq C}_{weak}(\sharp\mathcal{S}(B))_{uv} = \min\left\{\begin{array}{c} B_{uv} \ ; \ \frac{B_{u\overline{u}}+B_{\overline{v}v}}{2} \\ B_{ux} + C + B_{yv} \ ; \ B_{u\overline{y}} + C + B_{\overline{x}v} \\ 2C + B_{u\overline{y}} + B_{\overline{x}x} + B_{yv} \ ; \ 2C + B_{ux} + B_{y\overline{y}} + B_{\overline{x}v} \\ \frac{B_{u\overline{u}}+B_{\overline{x}x}}{2} + C + B_{yv} \ ; \ B_{ux} + C + \frac{B_{y\overline{y}}+B_{\overline{v}v}}{2} \\ \frac{B_{u\overline{u}}+B_{y\overline{y}}}{2} + C + B_{\overline{x}v} \ ; \ B_{u\overline{y}} + C + \frac{B_{\overline{x}x}+B_{\overline{v}v}}{2} \end{array}\right\}$$

We take each of the parameters of the minimum and prove it is larger or equal to $\sharp\mathcal{S}(\sharp\mathcal{A}^{x-y\leq C}_{weak}(B))_{uv}$. The only non-trivial ones are the four last ones. As they are symmetric, we only consider the following (recall that B is coherent):

$$\begin{aligned} \frac{B_{u\overline{u}} + B_{\overline{x}x}}{2} + C + B_{yv} &= \frac{B_{u\overline{u}}}{2} + \frac{B_{\overline{v}\,\overline{y}} + C + B_{\overline{x}x} + C + B_{yv}}{2} \\ &\geq \frac{\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)_{u\overline{u}}}{2} + \frac{\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)_{\overline{v}v}}{2} \\ &\geq \sharp\mathcal{S}(\sharp\mathcal{A}^{x-y\leq C}_{weak}(B))_{uv} \end{aligned}$$

2. B is weakly closed, so it has a null diagonal. We conclude by using statement 5. of Theorem 8.2.14 and Lemma 8.2.6.

3. (i) ⇒ (ii), (ii) ⇒ (iii) and (iv) ⇒ (i) are proved similarly to Theorem 8.2.14.

   We prove (iii) ⇒ (iv). We assume $0 \leq C + B_{yx}$ and $0 \leq 2C + B_{y\overline{y}} + B_{\overline{x}x}$ and prove $\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)$ is weakly closed.

   We have $0 \leq C + \sharp\mathcal{S}(B)_{yx}$, and $\sharp\mathcal{S}(B)$ is strongly closed, because B is weakly closed. So, by Theorem 8.2.14, $\sharp\mathcal{A}^{x-y\leq C}_{oct}(\sharp\mathcal{S}(B))$ is strongly closed. By 1., $\sharp\mathcal{S}(\sharp\mathcal{A}^{x-y\leq C}_{weak}(B))$ is strongly closed, and therefore, $\sharp\mathcal{A}^{x-y\leq C}_{weak}(B)$ is weakly closed.

□

8.2.7. Widening

Widening on octagons has several variants in the literature. In our work, we use its simplest form, where bounds are extrapolated to $+\infty$. Miné [Min04] describes a strategy of widening by thresholds for octagons. In this simple form, widening for DBMs in closed or strongly closed form is given by the following operator:

Definition 8.2.13. *Let A and B be two DBMs. We define $A \triangledown B$ as the DBM such that, for $u, v \in \mathbb{V}_{\pm}$:*

$$(A \triangledown B)_{uv} = \begin{cases} A_{uv} & \text{if } A_{uv} \leq B_{uv} \\ +\infty & \text{otherwise} \end{cases}$$

It is straightforward to check that this operator verifies the usual properties of widenings: (3.7), (3.8), (3.9) and even (3.10). It is worth noting that, even if A and B are closed or strongly closed DBMs, there is no reason for $A \triangledown B$ to be closed or strongly closed. Therefore, state-of-the-art implementations of the Octagon abstract domain might run a closure algorithm just after each widening. There is a catch, however: closure after a widening can break the termination of fixpoint iteration [Min04, Section 3.7.2]. Our solution to this particular problem lies in the use of a different widening and a different fixpoint iteration scheme (see Section 3.1.2).

In the weakly closed setting that we use for representing sparse DBMs, the use of $\triangledown$ would not compute abstract environments equivalent to those computed with the strongly closed setting. We aim at designing algorithms that can be used transparently instead of the regular ones. Thus, we need an adaptation of the widening for weakly closed DBMs.

The adapted widening uses the possibility offered by our widening formalism of returning two values for the widening, that we have exposed in Section 3.1.2. Recall that in our formalism, the widening returns two abstract environments: the first, *the iteration abstract state*, is used as the first argument of the next widening of the iteration, and the second is used to analyze e.g., the body of the considered loop. We have explained that there is no requirement that these two values have the same type.

In particular, for octagons in the weakly closed setting, we store additional information in the iteration abstract state. Namely, additionally to the current DBM stored in its sparse form, we store a matrix of Booleans indicating, for each cell of the DBM containing $+\infty$, whether it has already been extrapolated or not. This pair of matrices constitute an efficient representation of the DBM that would have been the iteration abstract state if we had used the usual, dense setting of strongly closed DBMs. It always verifies a specific invariant, stating that the DBM is maximally sparse and that the Boolean is never `true` when the corresponding cell in the DBM is finite.

Definition 8.2.14. *1. In the setting of sparse DBMs, a iteration abstract state is a pair (B, W) of a DBM B and a matrix of Booleans W such that, for all $u, v \in \mathbb{V}_{\pm}$ with $B_{uv} < +\infty$, we have:*

- $W_{uv} =$ `false`
- *If $u \neq \overline{v}$, then $B_{uv} < \frac{B_{u\overline{u}} + B_{\overline{v}v}}{2}$*

2. We associate to an iteration abstract state (B, W) of the sparse setting a DBM ${}^{W}\mathcal{S}(B, W)$, corresponding to the dense setting, defined such that, for $u, v \in \mathbb{V}_{\pm}$:

$$ {}^{W}\mathcal{S}(B, W)_{uv} = \begin{cases} +\infty & \text{if } W_{uv} = \texttt{true} \\ {}^{\sharp}\mathcal{S}(B)_{uv} & \text{otherwise} \end{cases} $$

We can now define our adaptation of the widening: the definition contains two operators: the actual widening operator, returning a pair of an iteration abstract state and a DBM, and `init_iter`, the operator that returns the initial iteration abstract state from the initial DBM of the iteration. The widening operator is defined in four steps, similarly to the join operator that used two steps.

Definition 8.2.15 (Sparse widening)**.** *Let B be a DBM, and (A, W) an iteration abstract state. We state $A^{1/2}_{uv} = \frac{A_{u\overline{u}} + A_{\overline{v}v}}{2}$ and $B^{1/2}_{uv} = \frac{B_{u\overline{u}} + B_{\overline{v}v}}{2}$.*

1. *We define* $\texttt{init\_iter}(B)$ *the iteration abstract state such that, for* $u, v \in \mathbb{V}_\pm$*:*

$$\texttt{fst}(\texttt{init\_iter}(B))_{uv} = \begin{cases} +\infty & \text{if } \frac{B_{u\overline{u}} + B_{\overline{v}v}}{2} \leq B_{uv} \text{ and } u \neq \overline{v} \\ B_{uv} & \end{cases}$$
$$\texttt{snd}(\texttt{init\_iter}(B))_{uv} = \texttt{false}$$

2. *We define* $(A, W) \triangledown^0_{weak} B$*, such that, for* $u, v \in \mathbb{V}_\pm$*:*

$$((A, W) \triangledown^0_{weak} B)_{uv} = \begin{cases} A_{uv} & \text{if } \sharp\mathcal{S}(B)_{uv} \leq A_{uv} < +\infty \\ A^{1/2}_{uv} & \text{if } \begin{array}{r} B_{uv} < +\infty \ \wedge \ A_{uv} = +\infty \ \wedge \ W_{uv} = \texttt{false} \\ \wedge \ B_{uv} \leq A^{1/2}_{uv} \ \wedge \ (A_{u\overline{u}} < B_{u\overline{u}} \vee A_{\overline{v}v} < B_{\overline{v}v}) \end{array} \\ +\infty & \text{otherwise} \end{cases}$$

3. *We define* $(A, W) \triangledown^W_{weak} B$*, such that, for* $u, v \in \mathbb{V}_\pm$*:*

$$((A, W) \triangledown^W_{weak} B)_{uv} = \begin{cases} \texttt{true} & \text{if } A_{uv} < +\infty \ \wedge \ ((A, W) \triangledown^0_{weak} B)_{uv} = +\infty \\ W_{uv} & \text{otherwise} \end{cases}$$

4. *We define* $(A, W) \triangledown^1_{weak} B$*, such that, for* $u, v \in \mathbb{V}_\pm$*:*

$$((A, W) \triangledown^1_{weak} B)_{uv} = \begin{cases} A^{1/2}_{uv} & \text{if } \begin{cases} \begin{pmatrix} \vee & A_{u\overline{u}} < B_{u\overline{u}} \wedge A_{\overline{v}v} > B_{\overline{v}v} \\ & A_{u\overline{u}} > B_{u\overline{u}} \wedge A_{\overline{v}v} < B_{\overline{v}v} \end{pmatrix} \\ W_{uv} = \texttt{false} \\ ((A, W) \triangledown^0_{weak} B)_{uv} = +\infty \\ B^{1/2} \leq A^{1/2} < +\infty \end{cases} \\ ((A, W) \triangledown^0_{weak} B)_{uv} & \text{otherwise} \end{cases}$$

5. *We define* $(A, W) \triangledown_{weak} B$ *by:*

$$\texttt{fst}((A, W) \triangledown_{weak} B) = ((A, W) \triangledown^1_{weak} B, (A, W) \triangledown^W_{weak} B)$$
$$\texttt{snd}((A, W) \triangledown_{weak} B) = (A, W) \triangledown^1_{weak} B$$

The following theorem states that this definition of the widening maintains the invariant of the iteration abstract state and computes the same thing as the usual widening operator $\triangledown$:

Theorem 8.2.16. *Let* B *be a DBM, and* (A, W) *an iteration abstract state.*

1. $\texttt{init\_iter}(B)$ *is an iteration abstract state.*
2. ${}^W\mathcal{S}(\texttt{init\_iter}(B))_{uv} = B$
3. $(A, W) \triangledown_{weak} B$ *is an iteration abstract state.*
4. ${}^W\mathcal{S}(\texttt{fst}((A, W) \triangledown_{weak} B)) = {}^W\mathcal{S}(A, W) \triangledown \sharp\mathcal{S}(B)$
5. $\sharp\mathcal{S}(\texttt{snd}((A, W) \triangledown_{weak} B)) = {}^W\mathcal{S}(A, W) \triangledown \sharp\mathcal{S}(B)$

We do not detail the proof of this very technical theorem. As an argument of its correctness, we have proved it correct using the Coq proof assistant. If the reader wanted more detail, she would have to go through many cases: most of them are trivial, and the proofs of the others are inspired by the proof of Theorem 8.2.13.

It should be noted, however, that even if B is weakly closed, there is no guarantee on whether $\mathtt{snd}((A, W) \triangledown_{weak} B)$ is weakly closed. Therefore, a closure algorithm is run on this DBM before any use (but not on $\mathtt{fst}((A, W) \triangledown_{weak} B)$, because that could break the termination of the fixpoint iteration, as explained above).

8.2.8. A Note on Tightening

Miné [Min04] and Bagnara et al. [BHZ09] study the case of the Octagon abstract domain when the considered environments take only values in $\mathbb{Z}$: in contrast with the previous sections, in this case, the strongly closed DBMs are not all canonical, so that modified algorithms need to be used. Even though we do not use these algorithms in Verasco, we explain here that the use of the weakly closed setting is not problematic when considering the integer case.

To this end, we define a different concretization function, $\gamma^{\mathbb{Z}}_{oct}$, that concretizes only to integer environments:

Definition 8.2.16 (Integer concretization of octagons)**.** *Let B be a DBM. We define:*

$$\gamma^{\mathbb{Z}}_{oct}(B) = \{\rho \in \gamma_{oct}(B) \mid \forall u \in \mathbb{V}_+,\ \rho(u) \in \mathbb{Z}\}$$

If we consider only integer environments, best abstractions have a slightly stronger characterization. In particular, all the bounds should be in $\mathbb{Z}$, and bounds on differences of the form $\rho(u) - \rho(\overline{u})$ are necessarily even. Such DBMs are said to be *tightly closed*. We also define the notion of *weakly tightly closed* DBMs, which is the analog of *weakly closed* DBMs for the integer case.

Definition 8.2.17 (Tight closure)**.** *Let B be a DBM. B is* tightly closed *(respectively weakly tightly closed) when:*

- *B is strongly closed (respectively weakly closed)*
- $\forall uv \in \mathbb{V}_\pm,\ B_{uv} \in \mathbb{Z}$
- $\forall u \in \mathbb{V}_\pm,\ \frac{B_{u\overline{u}}}{2} \in \mathbb{Z}$

The following theorem states that tightly closed DBMs are exactly best abstractions for integer environments:

Theorem 8.2.17. *Let B be a DBM. The two following properties are equivalent:*

(i) B is tightly closed

(ii) $\gamma^{\mathbb{Z}}_{oct}(B) \neq \emptyset$ and $B = \alpha(\gamma^{\mathbb{Z}}_{oct}(B))$

Proof. (i) $\Rightarrow$ (ii) is an easy consequence of [Min04, Theorem 4.3.7].

(ii) $\Rightarrow$ (i) We have:

$$B = \alpha(\gamma^{\mathbb{Z}}_{oct}(B)) \overset{\sharp}{\leq} \alpha(\gamma_{oct}(B)) \overset{\sharp}{\leq} B$$

So $B = \alpha(\gamma_{oct}(B))$, and B is strongly closed (Theorem 8.2.5). The other two properties of tight closure are obvious. □

Bagnara et al. [BHZ09, Section 6] give efficient algorithms for computing the tight closure of a DBM. Essentially, it consists in applying a *tightening* operation before strengthening. The *tightening* operation is defined by:

Definition 8.2.18 (Tightening)**.** *Let B be a DBM with elements in $\mathbb{Z}$. We define ${}^{\natural}\mathcal{T}(B)$ to be the DBM with elements in $\mathbb{Z}$ such that, for $u, v \in \mathbb{V}_{\pm}$:*

$$ {}^{\natural}\mathcal{T}(B)_{uv} = \begin{cases} B_{uv} - 1 & \text{if } u = \overline{v} \text{ and } B_{uv} \text{ is odd} \\ B_{uv} & \text{otherwise} \end{cases} $$

The following theorem gives the essential property of the tightening operation:

Theorem 8.2.18. *Let B be a weakly closed DBM with elements in $\mathbb{Z}$. We suppose that $\forall u \in \mathbb{V}_{\pm},\ 0 \leq {}^{\natural}\mathcal{T}(B)_{u\overline{u}} + {}^{\natural}\mathcal{T}(B)_{\overline{u}u}$. Then ${}^{\natural}\mathcal{T}(B)$ is weakly tightly closed.*

Proof. Because B has elements in $\mathbb{Z}$, it is easy to see that:

$$ \forall uv \in \mathbb{V}_{\pm},\ {}^{\natural}\mathcal{T}(B)_{uv} \in \mathbb{Z} $$
$$ \forall u \in \mathbb{V}_{\pm},\ \frac{{}^{\natural}\mathcal{T}(B)_{u\overline{u}}}{2} \in \mathbb{Z} $$

So, it remains to prove that ${}^{\natural}\mathcal{T}(B)$ is weakly closed. The facts that it is coherent and that it has a null diagonal follows easily from the corresponding properties of B. Thus, it remains to prove, for $u, v, w \in \mathbb{V}_{\pm}$:

$$ {}^{\natural}\mathcal{S}({}^{\natural}\mathcal{T}(B))_{uw} \leq {}^{\natural}\mathcal{T}(B)_{uv} + {}^{\natural}\mathcal{T}(B)_{vw} $$

We distinguish several cases:

- If $u \neq \overline{v}$ and $v \neq \overline{w}$, then we have:

$$ {}^{\natural}\mathcal{S}({}^{\natural}\mathcal{T}(B))_{uw} \leq {}^{\natural}\mathcal{S}(B)_{uw} \leq B_{uv} + B_{vw} = {}^{\natural}\mathcal{T}(B)_{uv} + {}^{\natural}\mathcal{T}(B)_{vw} $$

- If $u = \overline{v} = w$, then we have:

$$ {}^{\natural}\mathcal{S}({}^{\natural}\mathcal{T}(B))_{uu} \leq {}^{\natural}\mathcal{S}(B)_{uu} = 0 \leq {}^{\natural}\mathcal{T}(B)_{u\overline{u}} + {}^{\natural}\mathcal{T}(B)_{\overline{u}u} $$

- If $u = \overline{v} \neq w$, then we have, because ${}^{\natural}\mathcal{S}(B)$ is strongly closed:

$$ \begin{aligned} B_{\overline{w}w} = {}^{\natural}\mathcal{S}(B)_{\overline{w}w} &\leq {}^{\natural}\mathcal{S}(B)_{\overline{w}u} + {}^{\natural}\mathcal{S}(B)_{u\overline{u}} + {}^{\natural}\mathcal{S}(B)_{\overline{u}w} \\ &= {}^{\natural}\mathcal{S}(B)_{u\overline{u}} + 2{}^{\natural}\mathcal{S}(B)_{\overline{u}w} \leq B_{u\overline{u}} + 2B_{\overline{u}w} \end{aligned} $$

  Therefore, using the fact that $u \neq w$:

$$ \begin{aligned} {}^{\natural}\mathcal{T}(B)_{\overline{w}w} &\leq 1 + {}^{\natural}\mathcal{T}(B)_{u\overline{u}} + 2{}^{\natural}\mathcal{T}(B)_{\overline{u}w} \\ {}^{\natural}\mathcal{S}({}^{\natural}\mathcal{T}(B))_{uw} \leq \frac{{}^{\natural}\mathcal{T}(B)_{u\overline{u}} + {}^{\natural}\mathcal{T}(B)_{\overline{w}w}}{2} &\leq \frac{1}{2} + {}^{\natural}\mathcal{T}(B)_{u\overline{u}} + {}^{\natural}\mathcal{T}(B)_{\overline{u}w} \end{aligned} $$

  But both ${}^{\natural}\mathcal{S}({}^{\natural}\mathcal{T}(B))_{uw}$ and ${}^{\natural}\mathcal{T}(B)_{u\overline{u}} + {}^{\natural}\mathcal{T}(B)_{\overline{u}w}$ are integers, so it follows:

$$ {}^{\natural}\mathcal{S}({}^{\natural}\mathcal{T}(B))_{uw} \leq {}^{\natural}\mathcal{T}(B)_{u\overline{u}} + {}^{\natural}\mathcal{T}(B)_{\overline{u}w} $$

- If $u \neq \overline{v} = w$, we use a similar argument as in the previous case.

□

This theorem has two consequences. First, as already explained by Bagnara et al. [BHZ09, Section 6], this theorem gives an efficient algorithm to compute tight closure: to this end, one would compute the closure of the input matrix, then tighten it and finally strengthen it. If the DBM before strengthening verifies $\forall u \in \mathbb{V}_{\pm}, 0 \leq B_{u\overline{u}} + B_{\overline{u}u}$, then Theorem 8.2.18 ensures that the result of the algorithm is tightly closed. Otherwise, the initial DBM is not consistent. The second consequence is that our sparse algorithms need only small adjustments when used with integer environments: instead of maintaining the DBMs weakly closed, we just have to make them weakly tightly closed by slightly adapting the abstract operators. In practice, among all the operators we have described, only $\sharp\mathcal{A}$ needs to be adapted.

Note, however, that tightening does not address the case of mixed environments, where some variables are known to have integer values, and some others can have arbitrary real values. To the best of our knowledge, there is no known efficient closure algorithm supporting this use case, even in the dense setting.

In Verasco, environments contain both integers and floating-point values. Therefore, we are in a situation which is very close to the real-integer mixed environment case, so that we see integer variables as if they were real variables, and we do not use tightening.

8.3. Implementation of Octagons in Verasco

In this section, we give the design ideas we used for implementing the Octagon abstract domain in Verasco. We first explain the data structure we used, and then give more detail on the implementation of the different abstract domain primitives.

8.3.1. Data Structures for Octagons in Verasco

From the type `var` of variable identifiers, we build the type `ovar` of *signed* variable identifiers:

```
Inductive ovar :=
| ovPlus (v:var)
| ovMinus (v:var).
```

Difference bound matrices (DBMs) are represented using finite double precision floating-point numbers. Therefore, we first define a type for finite floating-point numbers, using a subset type:

```
Definition fin_float := {f:float | is_finite f = true}.
```

Then, we can define the type of DBMs, which are abstract environments for the Octagon abstract domain:

```
Definition t := To.t (To.t fin_float).
```

where `To.t` is a type for maps using keys of type `ovar`. That is, DBMs are maps indexed by signed variables and containing maps indexed by signed variables and containing finite floating-point numbers.

Another way of seeing this type for abstract environments is by "uncurrying" it. That is, instead of seeing it as a map containing maps, indexed by signed variables, we see it as a map indexed by a pair of signed variables:

```
Definition t := Too.t fin_float.
```

Where `Too.t` is a type of maps using pairs of signed variables as keys. The definition of `Too.t` makes these two definitions *convertible*. We find very convenient to see the type `t` either as nested maps using signed variables as keys, or as a map using pairs of variables as keys.

We should note that by using floating-point bounds, our data structure for DBMs cannot represent bounds with infinite precision. Therefore, in the algorithms of the Octagon abstract domain, we use the safe rounding towards $+\infty$ for all the newly computed bounds. As already discussed by Miné [Min04, Section 7.5.3] and Bagnara et al. [BHZ09, Section 7], this method preserves the soundness of the domain of octagons, but breaks most of the results about the precision. In particular, nearly none of the abstract transfer functions preserve the closure property. For this reason, we did not formally prove any precision result of the Octagon abstract domain[13]. However, we think that the loss of precision is small in practice, given the large accuracy provided by the IEEE double precision floating-point type. As an example, this type is able to represent exactly every integer between -2^{53} and 2^{53}.

These maps are sparse: the absence of a binding corresponds to an infinite bound, and we do a best effort storing as few bounds as possible, while maintaining the invariant that the map is weakly closed (modulo rounding errors). Moreover, when computing an abstract transfer function, we do a best effort trying to exploit and maintain sharing of the different abstract environments. To that end, we use the techniques described in Section 9.1.

Miné [Min04, Section 4.5.1] notices that, when it is strongly closed, nearly half the information contained in a DBM is redundant, because of the coherence property (Definition 8.2.2). Therefore, Miné uses an efficient representation for DBMs where only the lower left part is stored in memory. This optimization could be extended to weakly closed DBMs, by further assuming that DBMs are coherent. However, we do not use this optimization, which significantly complicates accesses to DBMs, and, in particular, prevents using efficiently the `combine` function that iterates simultaneously on two maps.

We do not implement tightening, the specific closure operator for integer environments. We have discussed this choice in Section 8.2.8: this is mainly motivated by the fact that, to the best of our knowledge, there is no efficient tightening algorithm when the considered environments contain integers and reals.

Concretization

In order to define the concretization function for octagons, we first describe how an environment of ideal values (that is, of type `var -> ideal_num`) can be seen as an environment of real values over signed variables. This is done in two steps: first, we project the domain to `R` with the `rρ` function, using automatically inserted coercions that convert floating-point and integer values to real values, and second we extend the codomain to `ovar`:

```
Definition rρ (ρ : var -> ideal_num) (x:var) : option R :=
  match ρ x with
  | INz z => Some (z:R)
  | INf f => if is_finite f then Some (f:R) else None
  end

Definition oρ (ρ : var -> ideal_num) (x:ovar) : option R :=
  match x with
```

[13] The proofs indicated in Section 8.2 as being formally verified were done in a simplified setting, assuming infinite precision.

```
  | ovPlus x => rρ ρ x
  | ovMinus x => option_map Ropp (rρ ρ x)
  end.
```

It is important to understand that not all ideal values can be projected to the set of real numbers. Namely, non-finite floating-point numbers do not have a real interpretation. For that reason, ideal environments are translated to environments valued in `option R` instead of `R`. We see here a difficulty that was unforeseen in Section 8.2: because some variables may be associated with no real concrete value, we have to check for finiteness of floating-point variables each time it can appear in the octagon. In particular, it is *unsound* to set to 0 the whole diagonal of a DBM.

Given `Too.get`, the function reading a binding in a map of type `Too.t`, the concretization function for octagons is defined by:

```
Instance ogamma: gamma_op t (var -> ideal_num) :=
  fun ab ρ =>
    ∀ x y b, Too.get (x, y) ab = Some b ->
    ∃ xv yv, oρ ρ x = Some xv /\ oρ ρ y = Some yv /\ (xv-yv <= b).
```

That is, if the DBM contains a bound `b` for a pair of variables `x` and `y`, then the environment does contain finite values for these variables, and their difference is smaller than `b`.

8.3.2. Implementing Abstract Operations

Given the data structure and concretization function we defined in Section 8.3.1, we are now ready to give more detail on the implementation of the abstract operations over octagons. We first concentrate on lattice operations and widening, and then give details about assignment, treatment of messages and forgetting of variables.

Lattice operations and widening

The implementation of the comparison, join and widening operators is very similar to that described in Section 8.2. We should note, however, that they are all implemented using the `shcombine` and `shcombine_diff` primitives exploiting the sharing in data structures, described in Section 9.1. Therefore, no computation time overhead is incurred when comparing or joining abstract environments sharing most of their contents.

Join is implemented using two steps, as described in Section 8.2.5: the first one iterates over the two maps simultaneously using `shcombine`, while the second one iterates over the list of differences in elements of the form $A_{u\overline{u}}$.

The implementation of widening follows the same two steps, with an additional closure computation at the end of widening, made necessary by the breaking of the weak closedness invariant by the widening property. This closure computation is done by an implementation of the Floyd-Warshall algorithm, improved to leverage the fact that bounds that are left untouched by the widening need not be updated.

Assignment and processing messages

Before evaluating an assignment or processing a message, a linearization of the expression takes places. For an assignment, this is the Octagon abstract domain that queries the linearization abstract domain described in Section 8.1. For messages, the linearization automatically translates the message into the equality of a quasi-linear expression and 0. Therefore, the only messages handled by the Octagon abstract domain are `Linear_zero_msg` messages.

The algorithms we use are inspired from those of Miné [Min04, Definition 4.4.7]. That is, we do not try to be too smart[14] inferring octagonal constraints from quasi-linear expressions, but we infer the simplest ones.

After the linearization step, assignments and processing of messages share a common intermediate data structure. Namely, they use the notion of *constraint row*. A constraint row is a set of difference constraints between a concrete value and the value of variables in an environment: as an example, a DBM can be seen as a collection of constraint rows, one for each variable. Constraint rows and their interpretation are defined as follows:

```
Definition constraint_row := To.t fin_float.

Definition constraints_row_valid (row:constraint_row) (ρ:var->ideal_num) : 𝒫 R :=
  fun (v:R) =>
    ∀ x b, To.get x row = Some b ->
    ∃ xv, oρ ρ x = Some xv /\ (v-xv <= b).
```

That is, a constraint row is a map from signed variables to finite floating-point numbers. The interpretation of a constraint row, given by the `constraint_row_valid` predicate, is the set of real numbers satisfying all the difference constraints in a given ideal environment.

When analyzing a `Linear_zero_msg` message giving a constraint of the form $e = 0$ for e a quasi-linear expression, the Octagon abstract domain first computes an upper bound for all the expressions of the form $e \pm x \pm y$ for x and y appearing in e: these bounds serve as upper bounds in the new octagonal constraints of the form $\pm x \pm y \leq \_$. Also, it computes upper bounds for all the expressions of the form $e \pm x$ and derives octagonal constraints of the form $\pm x \pm x \leq \_$. Among these new octagonal constraints, the ones sharing the same first variable are then bundled in constraints rows. Finally, we use a modified version of the algorithm described in Definition 8.2.12 that is able to insert in the DBM all the constraints contained in a constraint row simultaneously in quadratic time.

To model an assignment $x := e$ of an expression e to a variable x, we use an algorithm that simulates the following procedure: it assumes $x' = e$ for a fresh variable x', then it projects out variable x and finally it renames variable x' into x. The actual algorithm is slightly more complex, due to the impossibility to generate fresh variables.

Forgetting a variable

The last abstract operation we consider here is the `forget` transfer function: its implementation is simple, it consists in removing the entries of the DBM corresponding to the variable. This is similar to what is described in Definition 8.2.8, except that we do not reset the corresponding diagonal elements to 0, because of the possibility of non-finite values..

8.3.3. Cooperation with Other Domains

Each time the Octagon abstract domain computes a new interval for a variable, it broadcasts a `Itv_msg` message or a `Known_value_msg` message to other domains to keep them in sync. As we already explained, another kind of cooperation is with the linearization abstract domain to get quasi-linear forms for expressions.

The Octagon abstract domain does not answer any query. It could answer `get_itv` queries for variables, but, due to its complicated data structures, the answer to the query would be costly. Moreover, this would be redundant with the answer of the interval abstract domain, since the latter is supposed to be kept in sync with the Octagon abstract domain.

[14] E.g., we are not able to build the octagonal constraint $a - b \leq 0$ from the linear constraint $2a - 2b \leq 0$.

IV

Perspectives and Conclusions

Chapter 9

Data Structures with Sharing in Coq[1]

Designing a realistic static analyzer is not only a matter of finding precise approximations of programs. The efficiency of the analyzer is also of prime importance. The need for efficient algorithms and data structures goes beyond what was previously needed for formally proved programs in Coq. It is very common to see state-of-the-art static analyzers run for several hours to prove programs correct. For example, the authors of the very scalable high-performance Astrée static analyzer report analysis time of more than one day and use several gigabytes of memory [CCF+09].

Verasco is far from the performance of these highly-tuned analyzers. Nevertheless, the design of the data structures used for the many abstract domains of Verasco is of prime importance. In particular, we need efficient maps that support relatively fast element access and sharing. The sharing mechanism is not only needed for reducing memory consumption: when computing a join or a widening of very similar abstract values, it is essential that very little computation be needed, because join points are very frequent in real code.

The usual way [BCC+02, Section 6.2][CCF+09, Section 4.2] of implementing such an efficient join operation for maps in OCaml is to use physical equality: in functional languages such as OCaml, there are two equality operators. The structural equality traverses its parameters and compares the information stored in data structures, while the physical equality compares the memory location used to store the given values. In particular, two physically equal objects are necessarily structurally equal, but the converse is false. Roughly, if we assume that the OCaml compiler did not optimize the code, two physically equal values have necessarily been created at the same time.

In our case, if two branches of a tree representing a map are physically equal, we know for sure that they contain the same abstract value, so that the join is trivial. However, introducing physical equality directly in a Coq program would be unsound. Indeed, in Coq, two values having the same structure are indistinguishable, and can be used instead of each other in any expression. However, two structurally equal values that are not physically equal are distinguishable using the physical equality. In Section 9.1, with the help of dependent types, we give a way of using physical equality in Coq programs without introducing unsoundness. We currently use this methodology in Verasco's data structures.

However, this method has some limitations: for example, it does not permit doing maximal sharing, also known as *hash-consing*, in Coq. Hash-consing can be used for several

[1] Apart from Section 9.1, this chapter describes the Coq code available at `https://github.com/braibant/hash-consing-coq`

purposes: first, it provides a fast mechanism for comparing values using physical equality or hash equality. Second, it is easy to use hash-consing to build fast map structures using hash-consed values as keys. Finally, using such maps it is possible to implement memoization.

This assessment led us, in collaboration with Braibant and Monniaux [BJM13, BJM14], to the study of several methods to implement maximal sharing (i.e., *hash-consing*) and memoization in formally verified Coq programs. We used the case study of binary decision diagrams (BDDs), which are one of the well known uses of the hash-consing technique. We tried different approaches and compared them, as reported in the following sections. These ideas are not currently implemented in Verasco, but we believe some of them (especially the SMART and SMART+UID approaches described in Section 9.4) could be adapted to many of its data structures.

9.1. Safe Physical Equality in Coq: the PHYSEQ Approach

The obvious way of introducing physical equality in Coq is to declare it as an axiom in the development, state that physical equality implies Leibniz equality, and ask the extraction mechanism to extract it to OCaml's physical equality:

```
Parameter physEq: ∀ A:Type, A -> A -> bool.
Axiom physEq_correct: ∀ (A:Type) (x y:A), physEq x y = true -> x = y.
Extract Constant physEq => "(==)".
```

However, this appears to be unsound. Let `a` and `b` be two physically different copies of the same value. Then we have `physEq a a = true` and `a = b`, using Coq's Leibniz equality. Thus, we deduce, in Coq's logic, that `physEq a b = true`, which is wrong.

This unsoundness is of a particular kind: in fact, the axioms we postulate are not inconsistent: they can be easily instantiated by posing `physEq x y = false`. However, the OCaml term `(==)` is not a valid extraction for `physEq`, and using it would make it possible to prove properties on programs that will become false after extraction.

In order to circumvent this problem, we propose to avoid having physical equality in the language. Instead, we provide a term that enables us to use physical equality, just like a church Boolean would enable us to use a Boolean without manipulating a term of type `bool`: that is, we expose a function, taking as parameter the values of each branch of the test of physical equality. In order to make sure no use of physical equality is harmful, we add the condition, as an additional dependent parameter in `Prop`, that both branches are actually *equal* in the case of physical equality. Thus, the type of `physEq` becomes:

```
physEq: ∀ {A B:Type} (x y:A) (eq neq:B) (H:x = y -> eq = neq), B.
```

Note that, as is, `physEq` is not useful, because both branches are always executed. There is a simple solution: delaying the evaluation of branches by hiding them behind a function taking the `unit` value as parameter:

```
physEq: ∀ {A B:Type} (x y:A) (eq neq:unit -> B)
          (H:x = y -> eq tt = neq tt), B.
```

This declaration is extracted to a simple OCaml term:

```
Extract Constant physEq =>
  "(fun x y eq neq -> if x == y then eq () else neq ())".
```

Adding such a term in Coq does not add any inconsistency in the logic. Indeed, one can see that any implementation of `physEq` will always return `neq tt`[2], which is a well-typed implementation, so that we can use it to *define* `physEq`:

```
Definition physEq {A B:Type} (x y:A) (eq neq:unit -> B)
                  (H:x = y -> eq tt = neq tt) :=
  neq tt.
```

This is equivalent to the OCaml implementation, because, in the case where `x == y`, then `x = y` and then `eq tt = neq tt`. This reveals a constraint on the use we will be able to do of `physEq`: the only allowed uses are the uses where a physical equality does not change the behavior of the program. It only allows for optimizations (either for space or for time) that gives an alternate implementation when some values are equal.

There is one additional feature in our implementation of `physEq`. Sometimes we need to nest occurrences of `physEq`, and, in this case, the fact that `x = y` in the `eq` branch may be needed to give of proof for the parameter `H` in the nested call. For this reason, we give as parameter of the `eq` branch the precondition that `x = y`. The implementation becomes:

```
Definition physEq {A B:Type} (x y:A)
                  (eq:{_:unit | x = y} -> B)
                  (neq:unit -> B)
                  (H:∀ (e:x = y), eq (exist _ tt e) = neq tt) : B :=
  neq tt.
```

For proving the dependent parameter of `physEq`, we use either the `refine` tactic or the `Program` commands [Soz07]. Both support writing the terms in Gallina syntax, leaving holes, and then filling the holes using tactics.

For convenience, we use the following notation instead of directly using `physEq`:

```
Notation "'ifeq' x == y 'then' A 'else' B" :=
  (physEq x y (fun _ => A) (fun _ => B) _) (at level 200).
```

In Verasco, we use `physEq` mainly for two different purposes. In Section 9.1.1, we will study its use for optimized versions of the `combine` primitive on maps. They are essential for the performance of join and widening. In Section 9.1.2, we will explain that it can be used for building data structure with more sharing.

9.1.1. Exploiting Sharing When Combining Maps

A common data structure in CompCert and Verasco are finite maps. A finite map contains a finite set of bindings from keys to values, using an efficient tree data structure. Finite maps support the `get` and `set` operations for reading efficiently a binding and for modifying or creating a binding. These data structures also provide operations for computing directly on the whole set of bindings. On such primitive is `combine`. It takes two maps as parameters, and a function, and applies this function on all the bindings of the two maps. Here is its type and specification:

```
combine: ∀ (A B C: Type), (option A -> option B -> option C) -> t A -> t B -> t C
gcombine: ∀ (A B C: Type) (f: option A -> option B -> option C),
    f None None = None ->
    ∀ (m1: t A) (m2: t B) (i: elt),
      get i (combine f m1 m2) = f (get i m1) (get i m2).
```

[2] if `x == y`, then `eq tt = neq tt`, so that `neq tt` is returned, otherwise `neq tt` is returned

where `t X` is the type of maps containing data of type `x`.

It is natural to use this primitive, for example, when implementing join and widening in, non-relational domains like intervals or congruences. If we give it a join function operating on one abstract value, `combine` is able to compute a join operation over entire abstract environments.

However, `combine` has a serious performance problem, because it requires iterating over entire abstract environments. Most often, we perform join operations on very similar abstract environments: in the case of a test, for example, the only affected variables are the ones assigned in either branch, which is expected to be a small fraction of the variables of the program. A common way of solving this problem is to skip physically equal branches of the maps. This corresponds to forcing $x \sqcup x = x$ for any abstract value, which seems reasonable.

In order to exploit sharing in join and widening and in other places of the implementation, we added two optimized versions of `combine` to the operations over CompCert maps: `shcombine` and `shcombine_diff`.

The first one, `shcombine`, takes, like `combine`, a function that it applies to each of the bindings of the given maps. However, it requires an additional pre-condition as argument, imposing that the given function is the identity if given two equal parameters[3]:

```
shcombine: ∀ (A:Type) (f:key -> option A -> option A -> option A),
             (∀ x v, f x v v = v) -> t A -> t A -> t A.
```

This primitive is implemented similarly to `combine`, except that, at each node, it checks whether the branch given as its first argument is physically equal to the branch given as its second argument. If so, no computation is done, and the map is returned as-is. This will return the same result as if the computation were done, since the hypothesis on `f` will force the result to stay equal to the parameters in this case.

The `shcombine` primitive gave a dramatic improvement of Verasco's performance (a factor if 100, at least). This, however, adds the constraint $x \sqcup x = x$ on the join operation over abstract values; this has to be proved for each non-relational abstract domain.

Another common operation that can leverage sharing is computing the differences between two abstract domains. For example, it is used when computing the messages that need to be sent in the message channel (see Section 3.3.1): after an abstract transfer function, the abstract domain needs to compute what has been changed, and send messages accordingly. Similarly, `combine` can be used for this purpose, but it needs to traverse entire maps, even if the difference is very small.

We introduce a new primitive, `shcombine_diff`, that is a dual of `shcombine`: instead of forcing equal binding to stay, it forces them to disappear:

```
shcombine_diff: ∀ (A B:Type) (f:key -> option A -> option A -> option B),
                  (∀ x v, f x v v = None) -> t A -> t A -> t B.
```

Its implementation follows the same scheme, and it also massively improves the performances of some operations.

9.1.2. Improving Sharing Using Physical Equality

In order to leverage as much as possible the potential of the primitives described in Section 9.1.1, it is clear that improving the sharing in the data structures is profitable. For

[3]For more flexibility, the given function also takes as parameter the key corresponding to the bindings, but this is a detail.

this reason, in many places, we try to use previously existing data structures instead of allocating new ones.

To illustrate this methodology, let's recall how CompCert trees are built: they are tries indexed by Coq's values of type `positive` (i.e., strictly positive integers represented in binary). That is, they are trees, where each node contains the value corresponding to the integer whose binary representation stops at this place, a subtree containing values whose binary representation has a 1 at the corresponding position and the symmetric subtree for integers using 0 at this position. They are represented using a simple Coq Inductive type:

```
Inductive tree (A : Type) : Type :=
| Leaf : tree A
| Node : tree A -> option A -> tree A -> tree A.
```

In the implementation of Verasco's maps (and in many other places that can take advantage of sharing), we try not to directly use the `Node` constructor[4]. Instead, when possible, we use the `Node_sh` constructor, that tries to use a given candidate if its children are physically equal. Its implementation is as follows:

```
Program Definition Node_sh {A:Type}
           (* m = Node l o r is the candidate used if children are
              physically equal. *)
           (m:t A) (l:t A) (o:option A) (r:t A) (eqm:m = Node l o r)

           (* m' tt = Node l' o' r' is the node we are building. *)
           (l':t A) (o':option A) (r':t A) (m':unit -> t A)
           (eqm':m' tt = Node l' o' r')
    : t A :=
  ifeq l == l' then
    match o, o' with
    | None, None => ifeq r == r' then m else m' tt
    | Some v, Some v' =>
      ifeq v == v' then (ifeq r == r' then m else m' tt) else m' tt
    | Some _, None | None, Some _ => m' tt
    end
  else
    m' tt.
Next Obligation. [...] Qed.
```

Note that, in the case it cannot use the given candidate, `Node_sh` does not actually builds the node. Instead, it calls a function, `m'`, that is supposed to do this work. This makes it possible to use another call to `Node_sh` in `m'`, in order to try different candidates.

The use of this smart constructor is a bit cumbersome when programming functions on trees, but the complexity disappears in the proofs: because of the definition of `physEq`, it appears that a call to `Node_sh` is *convertible* to a simpler call to the usual constructor `Node`.

9.2. Introduction to Hash-Consing, Memoization and BDDs

Hash-consing is a programming technique used to share identical immutable values in memory, keeping a single copy of semantically equivalent objects. It is especially useful in order to get a compact representation of abstract syntax trees, and provide fast hash and comparison functions over them, so that it could be advantageously used in Verasco for representing the different kinds of expressions, for example. A hash-consing library maintains a global

[4]The `Leaf` constructor is not allocated in a separated bloc in memory, so that sharing does not make sense.

pool of expressions and never recreates an expression equal to one already in memory, instead reusing the one already present. In typical implementations, this pool is a global hash table. Hence, the phrase *hash-consing* denotes the technique of replacing node *cons*truction by a lookup in a hash table returning a preexisting object, or creation of the object followed by insertion into the table if previously nonexistent. This makes it possible to get *maximal sharing* between objects, if hash-consing is used systematically when creating objects.

Moreover, a unique identifier is given to each object, allowing fast hashing and comparisons. This makes it possible to do efficient *memoization* (also known as *dynamic programming*): the results of an operation are tabulated so as to be returned immediately when an identical sub-problem is encountered.

The typical way of implementing hash-consing (a global hash table) does not translate easily into Coq. The reason is that Coq is a purely applicative language, without imperative traits such as hash tables, pointers, or pointer equality. In the following Sections, we discuss how hash-consing can be implemented using the Coq proof assistant (without modifying it) with two possible use cases in mind:

- Efficient execution when extracted to OCaml, similarly to CompCert and Verasco. Hash-consing could be used for abstract syntax trees; the BDD library could be directly used in a formally proved model-checking or static analysis tool like Verasco.

- Execution inside Coq with reasonable efficiency, e.g., for proofs by reflection.

More precisely, we present a few *design patterns* to implement hash-consing and memoization on various examples, and we use binary decision diagrams (BDDs) as our prime example. We evaluate these design patterns based on several criteria. For instance, we discuss for each pattern how we handle the maximal sharing property and how easy it is to prove the soundness of our implementation. We also discuss whether it is possible to compute inside Coq with a given solution.

9.2.1. A Primer on Binary Decision Diagrams

In the following, we use "reduced ordered binary decision diagrams" (ROBDDs, BDDs for short) as a running example of hash-consed data structures. BDDs are representations of Boolean functions and are often used in software and hardware formal verification tools, in particular *model checkers* [Knu11].

The data structure

A Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be represented as a complete binary tree with $2^n - 1$ decision nodes, labeled by variables x_i according to the depth i from the root (thus the adjective *ordered*); edges are labeled 0 and 1; leaves are labeled `T` (for true) or `F` (for false). The semantics of such a diagram is: $f(x_1, \ldots, x_n)$ is obtained by traversing from the root and following the 0 or 1 edge of a node labeled by x_i according to the value of x_i.

Such a tree can be *reduced* by merging identical subtrees, thus becoming a connected *directed acyclic graph* (DAG; see second diagram in Figure 9.1); furthermore, decision nodes with identical children are removed (see third diagram in Figure 9.1). These transformations preserve semantics. The reduced representation is *canonical*: given a variable ordering $x_1, \ldots, x_n$, a function is represented by a unique ROBDD.

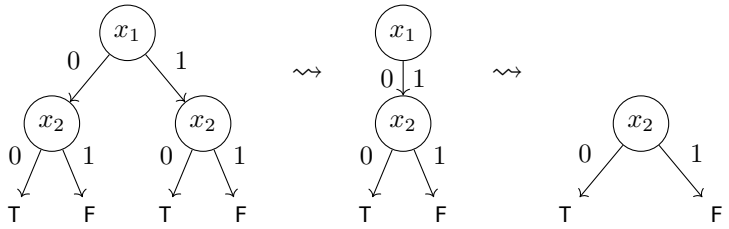


Figure 9.1: The BDD for the function $f(0,0) = f(1,0) = \mathsf{T}$, $f(0,1) = f(1,1) = \mathsf{F}$

Building BDDs

In practice, one directly constructs the reduced tree; let us see how. All BDD nodes allocated so far are stored in a global hash table, so that a new object is created only if not already in the hash table: *there are never two identical objects at two different locations in memory*, a crucial invariant that must be maintained throughout the execution.

Each BDD node is given a unique identifier: for instance, the current value of a global 64-bit counter incremented following each allocation[5]. Since a new object is never created if an identical object already exists, two objects are equal if and only if they have the same unique identifier. The equality test, which is usually expensive over tree structures, since it requires full traversal, is instead implemented by a very fast comparison of unique identifiers.

Unique identifiers are also used for hashing. When attempting to construct a node (v, b_0, b_1) where v is the variable, b_0 is the subtree labeled 0 and b_1 is the subtree labeled 1, one computes the hash value of the node as $H(v, u_0, u_1)$ where u_0 and u_1 are the respective unique identifiers of the subtrees b_0 and b_1. This hash value is then used to look up the node in the hash table. Again, such shallow hashing is considerably faster than having to traverse the tree structures.

Operations on BDDs

The principal way to build BDDs is to combine the diagrams of two functions f and g in order to obtain the BDD for other functions such as $f \wedge g$, $f \vee g$ of $f \oplus g$. The basic combinator of BDDs is called node *melding* [Knu11]. Intuitively, melding corresponds to a traversal of the internal nodes of two BDDs to build a third diagram. This traversal respects the order on variables, and thus, the third diagram will naturally be ordered. Then, all binary operations on BDDs are defined as one particular instance of melding.

Suppose that we have two nodes $\alpha = (v, l, h)$ and $\alpha' = (v', l', h')$. The melding of α and α', denoted by $\alpha \diamond \alpha'$ is defined as follows:

$$\alpha \diamond \alpha' = \begin{cases} (v, l \diamond l', h \diamond h') & \text{if } v = v' \\ (v, l \diamond \alpha', h \diamond \alpha') & \text{if } v < v' \\ (v', \alpha \diamond l', \alpha \diamond h') & \text{if } v > v' \end{cases}$$

Then, the binary operation $\diamond$ is entirely defined by the results of

$$\mathsf{F} \diamond \alpha \quad \alpha \diamond \mathsf{F} \quad \mathsf{T} \diamond \alpha \quad \alpha \diamond \mathsf{T}$$

[5]In certain implementations, the unique identifier is the address of the node; this supposes that objects never change address, which is not the case in OCaml.

For instance, the conjunction operation $f \wedge g$ can be defined by melding the BDDs for f and g, and using the following rewrite rules for the leaf cases:

$$\mathsf{F} \diamond \alpha \to \mathsf{F} \quad \alpha \diamond \mathsf{F} \to \mathsf{F} \quad \mathsf{T} \diamond \alpha \to \alpha \quad \alpha \diamond \mathsf{T} \to \alpha$$

For the sake of clarity, we focus on binary operations in the following: they are representative of the difficulties we had to face[6].

Memoization

The fact that each node has a unique identifier also makes it possible to memoize the results of BDD operations. One keeps a map from sub-problems (a pair of nodes α and α') to nodes (the result of $\alpha \diamond \alpha'$), so as to be returned immediately when a previously solved sub-problem is encountered.

In a BDD library, memoization is crucial to implement the or/and/xor operations with time complexity in $O(|a|.|b|)$ where $|a|$ and $|b|$ are the sizes of the inputs: as the function is executed, its results on the subtrees of the original problem are stored in a structure indexed by (u_a, u_b) where u_a and u_b are the unique identifiers of the a and b inputs. In contrast, the naive approach has exponential complexity, since the function may be evaluated exponentially often on the same couple of subtrees.

9.2.2. Memoization in Coq: the State of the Art

State monad

The first idea to implement memoization in Coq is to use a dedicated data structure to tabulate function calls: that is, use a finite map of some sort to store the pairs *(argument, result)* for a given function. Users of a library implemented in this fashion must, then, thread this state through the program using a state monad.

While this solution always works, it has non-negligible verification cost: we must prove that the bindings in the memoization table are correct, i.e., the values held in the table correspond to the result of the would-be corresponding function calls.

Then, if we memoize a function of type `A -> B`, we get a function of type `A -> M B`, where `M` is the type constructor associated with our state monad. Even if the latter seems equivalent to the former, these types are different: therefore, this tiny modification (memoizing a function) requires modifying every caller of the memoized function to use a monadic style.

Since it is cumbersome to perform all computations inside a state monad, we continue this section with a review of other (partial) solutions to the memoization problem.

Shallow memoization

It is possible in some cases to make a *shallow embedding* of this memoization table in Coq. Melquiond [Mel12] describes how to use co-inductive objects as a cache that stores previously computed values of a given function. In this section, we present his clever idea in Figure 9.2, with a few cosmetic changes[7]. A `lazy` value is defined as a co-inductive, which will effectively be represented as a thunk when executed by Coq's virtual machine, and will be extracted to OCaml as a `Lazy.t` thunk. A *thunk* is a term whose evaluation is frozen until it is actually needed (lazy evaluation), then the computed value is cached in the thunk

[6] Note that we only need two rewrite rules for commutative binary operations.

[7] We found out that similar definitions actually made their way into Coq's standard library under the name `StreamMemo.v`.

```
Section t.
  Context {A : Type}.
  CoInductive lazy : Type :=
  | Thunk : A -> lazy.

  CoInductive lazies : Type :=
  | Cons : lazy -> lazies -> lazies.

  Fixpoint nth n cst :=
    match n, cst with
    | O, Cons (Thunk xi) _ => xi
    | S p, Cons _ c => nth p c
    end.

  CoFixpoint mk_lazy {B} f (n : B) :=
    Thunk (f n).

  CoFixpoint memo' f n :=
    Cons (mk_lazy f n) (memo' f (S n)).

  Definition memo f := memo' f 0.
End t.

(* A costly version of the identity *)
Fixpoint big {A} n : A -> A:=
match n with
| 0 => fun x => x
| S n =>fun x =>  big n (big n x)
end.

Definition id n := big n n.
Definition k := memo id.
Definition run x := nth x k.
```

Figure 9.2: Memoization using co-inductives

so that it is instantly returned in case the term is evaluated again (thus, performing a kind of memoization).

Then, `lazies` represents a stream (an infinite list) of thunks, that can be peeked at using `nth`. Each time the virtual machine has to destruct the `cst` argument of `nth`, it will remember the two arguments of this cons cell. Then, it suffices to build a stream `memo` that computes the value of a function `f` for each possible natural number: in practice, `f` will be evaluated lazily, and only once per input value.

As an example, the following snippet of code memoizes the time-consuming function `id`. (Remark that it also demonstrates that the intermediate calls to `id` are not memoized.)

```
Time Eval vm_compute in run 26. (* 6s *)
Time Eval vm_compute in run 26. (* 0s *)
Time Eval vm_compute in run 25. (* 3s *)
Time Eval vm_compute in run 25. (* 0s *)
Time Eval vm_compute in id 25.  (* 3s *)
Time Eval vm_compute in id 25.  (* 3s *)
```

Note that this clever trick is not limited to functions over Peano numbers, and could be adapted for branching structures, e.g., memoizing functions over binary numbers. Unfortunately, it seems hard to adapt this idea to memoize recursive calls: consider for instance the function

```
Fixpoint exp n := match n with 0 => 1 | S n => exp n + exp n end.
```

In order to compute `exp` using a linear number of recursive calls we need a memoizing fixpoint combinator rather than a way to memoize a predefined function.

(To be complete, let us add that functions `f : nat -> A` that can be expressed as the iteration of a given function `g : A -> A`, can actually be memoized using the aforementioned technique. While we obviously could rewrite `exp` to fit into that scheme, this is not the case of the other functions that we use through our developments.)

This problem is somewhat representative of the one that we have to face when it comes to BDDs: first, memoizing fixpoint combinators are crucial; second, the function that we memoize depends on the global state of the hash-consing machinery, which evolves through times. In both cases, we fall out of the scope of the above trick: there is no predefined function to memoize.

Adjustable references

Now, we turn to a partial solution to the memoization problem, that works when the user is solely interested in executing the code extracted from a Coq program.

Vafeiadis [Vaf13] introduced a restricted form of mutable state called *adjustable references*. The idea is that adjustable references store some internal value that can only be updated in innocuous ways. That is, adjustable references are equipped with an observation function; and the update function ensures that the result of the observations remain equals through updates. It is therefore possible to adjust the values held in the references in a way that changes, e.g., the costs of some computations, yet does not change the result of these computations.

Vafeiadis demonstrated how to use adjustable references to memoize a function `f` by tabulating its results: the adjustable reference internal state is a finite map, while the observation function simply returns `f`. Updating the contents of the finite map does not change the observation function; but subsequent reads may make use of the fact that a value was previously computed and stored inside the memoization cache. We refer the reader to Vafeiadis' article for more details about this idea.

Remark that it is again the case that this technique does not scale to the definition of memoizing fixpoint combinators. Therefore, adjustable references do not quite fit our needs either.

9.2.3. Implementing BDDs in Coq.

The following two Sections 9.3 and 9.4 describe a total of *four* implementations of BDDs in Coq. To make things clear for the reader, we give each of these implementations a reference name, a pointer to the relevant section of the chapter and a short description.

PURE-DEEP. See Section 9.3.1. A pure Coq implementation of BDDs that makes a deep embedding of memory as finite maps and uses indices as surrogates of pointers.

PURE-SHALLOW. See Section 9.3.2. A pure Coq implementation of BDDs that uses a shallow embedding of memory.

SMART. See Section 9.4.1. An "impure" implementation of BDDs in Coq: we implement hash-consing and memoization through the extraction mechanism of Coq.

SMART+UID. See Section 9.4.2. A variation on the previous approach in which we discuss how to expose and axiomatize the operations on the unique identifiers associated with BDD nodes.

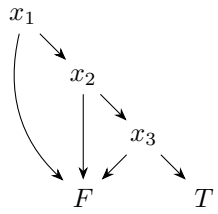
9.3. Pure BDD Implementations

We describe two pure Coq implementations of BDDs libraries in this section, that differ mainly by the way we model the memory and the allocation of nodes.

9.3.1. The PURE-DEEP Approach

The idea here is to model the memory using finite maps inside Coq and use indices in the maps as surrogates for pointers, implementing BDD operations as manipulations of these persistent maps. Such an implementation[8] was described in [VGLPAK00, VGL00], but we propose a new one here for the sake of completeness, and because the old one did not age well w.r.t. the evolution of Coq. Our implementation is defined as follows.

First, we assign a unique identifier to each decision node. Second, we represent the directed acyclic graph underlying a BDD as a Coq finite map from identifiers to decision nodes (that is, tuples that hold the left child, the node variable and the right child). For instance, the following graph, on the left, can be represented using the map on the right.



$$\begin{array}{lcl} 1 & \mapsto & (\mathrm{F},\, x_1,\, \mathrm{N}\ 2) \\ 2 & \mapsto & (\mathrm{F},\, x_2,\, \mathrm{N}\ 3) \\ 3 & \mapsto & (\mathrm{F},\, x_3,\, \mathrm{T}) \end{array}$$

Then, we implement the hash-consing pool using another map from decision nodes to node identifiers and a `next` counter that is used to assign a unique identifier to a fresh node. In the situation above, `next` is equal to 4 and the hash-consing map is defined as follows:

$$\begin{array}{lcl} (\mathrm{F},\, x_1,\, \mathrm{N}\ 2) & \mapsto & 1 \\ (\mathrm{F},\, x_2,\, \mathrm{N}\ 3) & \mapsto & 2 \\ (\mathrm{F},\, x_3,\, \mathrm{T}) & \mapsto & 3 \end{array}$$

Notations

We allow us some liberty when it comes to finite maps (that are pervasive in our code): the notation `t ⇝ s` denotes a type of efficient maps from type `t` to type `s`; and we use indiscriminately the notations `get` and `set` that respectively access and update such finite maps. Implicitly, these two notations have the following types:

```
get  :  A -> (A ⇝ B) -> option B
set  :  A -> B -> (A ⇝ B) -> (A ⇝ B)
```

Note that we do use the efficient finite maps from the Coq standard library; but we chose here to abstract from the particular module definitions and names that we have to use in our code, for the sake of legibility. In the following, we use Coq's positive numbers `positive` as unique identifiers, and use Patricia trees for finite maps.

Figure 9.3 shows our inductive definitions and the code of the associated allocation function `mk_node`, knowing that `alloc n st` allocates the fresh node `n` in the hash-consing state `st` (taking care of updating both finite maps and incrementing the "next fresh" counter). Then, equality between BDDs (`eqb`) is provided by decidable equality over node identifiers.

[8] One should also mention the seminal work by E. Ledinot in 1993 on the canonicity of binary decision dags, available from the Coq contributions.

```
Inductive expr := F | T | N : positive -> expr.

Definition eqb a b :=
  match a,b with
  | T,T => true
  | F,F => true
  | N x, N y => (x =? y)
  | _, _ => false
  end.

Definition node := (expr * var * expr).

Record hashcons_state := {
  graph : positive ⇝ node;
  hmap  : node ⇝ positive;
  next  : positive
}.

Definition alloc u st :=
  (st.(next), {|
    graph := set st.(next) u st.(graph);
    hmap  := set u st.(next) st.(hmap);
    next  := st.(next) + 1
  |}).

Definition mk_node (l : expr) (v : var) (h : expr) st :=
  if eqb l h then (l,st)
  else match get (l,v,h) st.(hmap) with
       | Some x => (N x, st)
       | None => let (x, st) := alloc (l,v,h) st in (N x, st)
       end.
```

Figure 9.3: Hash-consing in pure Coq, using a deep-embedding

All operations thread the current global state in a monadic fashion. The correctness of BDD operations corresponds to the facts that 1) the global state is used monotonically (that is the structure of the resulting global state is a refinement of the input one, see Figure 9.4); 2) the resulting global state is well-formed; and 3) the denotation of the resulting BDD expression is correct.

Invariants and well-formedness properties

We present the well-formedness properties we preserve over expressions (`wf_expr`) and over hash-consing states (`wf_hashcons`) in Figure 9.5. The inductive predicate `wf_expr st v e` depends on a variable level `v` that indicates a bound over the variable identifiers appearing in the BDD `e`. This ensures that variables are kept ordered. Then, a well-formed expression is either a leaf (cases `wf_T` and `wf_F`) or a pointer to a hash-consing node with a head variable `w` less than the bound `v`. (Remark that this definition of well-formed expression is not directly recursive, because the actual recursion takes place in the definition of `wf_hashcons` below; this notion of well-formed expressions coupled with well-formedness of the state suffice to prove that expressions are DAGs with a suitable ordering on the variables.)

The predicate `wf_hashcons` is the general well-formedness property over hash-consing states. The predicate `wf_bijection` ensures that the two maps `hmap` and `graph` form a bijection; `wf_expr_lt_next` ensures that the next fresh variable counter will indeed produce fresh variables; `wf_map_wf_expr_l` and `wf_map_wf_expr_r` ensure that a node stored in the

```
Record incr (st1 st2 : hashcons_state) : Prop := {
  incr_lt : st1.(next) ≤ st2.(next);
  incr_find : ∀ p x, get p st1.(graph) = Some x
                ->  get p st2.(graph) = Some x
}.
```

Figure 9.4: Monotonicity predicate over hash-consing states

```
Inductive wf_expr st : var -> expr -> Prop :=
| wfe_T : ∀ v, wf_expr st v T
| wfe_F : ∀ v, wf_expr st v F
| wfe_N : ∀ (p : positive) (l : expr) (w: var) (h : expr),
            get p st.(graph) = Some (l, w, h) ->
            ∀ (v : var), (w < v) ->
            wf_expr st v (N p).

Record wf_hashcons (st : hashcons_state) : Prop := {
  wf_bijection : ∀ p n, get n st.(hmap) = Some p
                   <-> get p st.(graph) = Some n;
  wf_expr_lt_next : ∀ p v, wf_expr st v (N p)
                        -> p < st.(next);
  wf_map_wf_expr_l : ∀ p x v y, get p st.(graph) = Some (x, v, y)
                             -> wf_expr st v x;
  wf_map_wf_expr_h : ∀ p x v y, get p st.(graph) = Some (x, v, y)
                             -> wf_expr st v y;
  wf_reduced : ∀ p l v h, get p st.(graph) = Some (l, v, h)
                       -> l <> h
}.
```

Figure 9.5: Well-formedness of hash-consing state

hash-consing structure has well-formed children, respecting the variable order; `wf_reduced` ensures that all those nodes are reduced (ie. their left child is different from their right child).

Note that the statement that the hash-consing structure is correct corresponds to the components `wf_bijection` and `wf_expr_lt_next`. The other statements correspond to properties about BDDs, namely the facts that they are reduced and ordered.

The denotation of BDD expressions.

Recall that BDD expressions, as we have defined them in this section, do not have an inductive structure we can follow: this means that we cannot define the semantics of BDD expressions as a Coq fixpoint. Rather, we have to define the semantics of BDD expressions as an inductive predicate that uses the state of the hash-consing data structure to go through the graph. The inductive `value` is shown on Figure 9.6. It is defined as a binary relation with two parameters (the valuation of variables `env` and the state of the hash-consing structure `st`) and two arguments: the expression and its denotation (a Boolean).

We only use the type of environments `env` to establish a semantics, thus efficiency is not an issue. However, we prefer to use finite maps of some kinds to represent environments: using a function of type `var -> bool` would force the client to reason about the fact that some BDD expressions are insensitive to the valuation of some variables. Using a finite map make this reasoning internally in our library and eases the client's life.

```
Inductive value env st : expr -> bool -> Prop :=
| value_T : value env st T true
| value_F : value env st F false
| value_N : ∀ (p : positive) (l : expr) (v : var) (h : expr),
              get p st.(graph) = Some (l, v, h) ->
              ∀ (vv : bool), get env v = Some vv ->
              ∀ (vhl : bool), value env st (if vv then h else l) vhl ->
                              value env st (N p) vhl.
```

Figure 9.6: The denotation of BDD expressions

```
Record memo := {
  mand : (positive * positive) ⇝ expr;
  mor  : (positive * positive) ⇝ expr;
  mxor : (positive * positive) ⇝ expr;
  mneg : positive ⇝ expr}.

Record state := { ... :> hashcons_state; ... :> memo}.
```

Figure 9.7: Adding memoization tables to the global state

Termination of BDD operations

The first problem that we have to solve in our Coq representation is that, as can be expected from our data structure, BDD operations cannot be defined using structural recursion (there is no inductive structure to follow). Unfortunately, we cannot easily use well-founded recursion here because the well-founded relation involves both parameters of the function and the global state.

The problem is that the termination of the BDD operations relies on the fact that the graph of nodes is acyclic; but the graph is not fixed through an execution of the melding operation! Rather, the global state is threaded in the recursive calls of $\diamond$. Therefore, proving the recursive calls to be well-founded requires to prove that the $\diamond$ operation is monotonic w.r.t. graph inclusion. In order to prove termination, we would have to define the fixpoint and prove this monotonicity property *at the same time*. This would involve embedding invariants directly in the global state, using dependent types. This has two major drawbacks: first, defining terms containing complex dependent types is cumbersome and proving properties about them is often very challenging. Second, we would have to pay close attention to prevent Coq from normalizing those proof terms if we want to use this library for reflection purposes.

In the end, we resorted to define partial functions that use a *fuel* argument to ensure termination: that is, they use an explicit bound on the number of iterations to do.

Memoizing operations

Finally, it is possible to enrich our hash-consing structure with memoization tables in order to tabulate the results of BDD operations.

The memoization tables (see Figure 9.7) are passed around by the state monad, just as the hash-consing structure. It is then necessary to maintain invariants on the memoization information.

In the definition of `state`, we use two *field coercions* (denoted `:>`) to declare coercions from the type `state` to the types `hashcons_state` and `memo`. In practice, this means that one can use a `state` in any expression that would expect the result of one of these two

```
Record wf_memo2 (st : hashcons_state) (m : (positive * positive) ⇝ expr)
               (opb : bool -> bool -> bool) : Prop :=
{ wf_memo2_find_wf_res :
    ∀ x y e, get (x, y) m = Some e
          -> wf_expr st v (N x) -> wf_expr st v (N y)
          -> wf_expr st v e;
  wf_memo2_find_wf :
    ∀ x y e, get (x, y) m = Some e ->
    ∃ s v, wf_expr st v (N x) /\ wf_expr st v (N y);
  wf_memo2_find_sem :
    ∀ x y res, get (x, y) m = Some res ->
    ∀ env vx vy, value env st (N x) vx -> value env st (N y) vy ->
                 value env st res (opb va vb)
}.

Record wf_memo_neg (st : hashcons_state) (m : positive ⇝ expr) : Prop :=
  { ... }.

Record wf_memo (st : state) : Prop :=
{ wf_memo_mand : wf_memo2 st st.(mand) Datatypes.andb;
  wf_memo_mor  : wf_memo2 st st.(mor) Datatypes.orb;
  wf_memo_mxor : wf_memo2 st st.(mxor) Datatypes.xorb;
  wf_memo_mneg : wf_memo_neg st st.(mneg)
}.

Record wf_st (st : state) : Prop :=
{ ... : wf_hashcons st;
  ... : wf_memo b
}.
```

Figure 9.8: Invariant over the memoization information

projections.

We present the invariants over the memoization maps for the binary operations[9] in Figure 9.8. First, we have to prove that the nodes referenced in the domain and in the codomain of those tables are well-formed and that those tables keep the bounds over the variables correct. Then, we have to state that the memoization information is semantically correct. One downside of this data structure definition is that we are forced to define one table per operation that we want to memoize, and that this is not modular: adding a new operation requires modifying the definition of the memoization state and add the corresponding field in the `wf_memo` record.

(Note that finding the correct pattern of memoization for a program is still an art rather than a science: using the data structure above, we keep the memoized values from one run of an operation to the other. In this section, we will settle on this conservative strategy, but other memoization strategies are possible that yield different performance and memory consumption profiles.)

A mouthful of code

The final version of our code is shown on Figure 9.9. We use a do-notation à la Haskell to make it more palatable.

Then, we prove that under some hypotheses, this combinator is correct: that is, it produces well-formed hash-consing structures and memoization tables, and the denotation of

[9]The case of the negation operation is similar and is not detailed.

```
Section combinator.
  (* fx is the base case when one operand is F *)
  Variable fx : expr -> state -> option (expr * state).
  (* tx is the base case when one operand is T *)
  Variable tx : expr -> state -> option (expr * state).
  Variable memo_get : positive -> positive -> state -> option expr.
  Variable memo_update : positive -> positive -> expr -> state -> state.

  Definition memo_node a b l v h st :=
    let (res, st) := mk_node l v h st in
    let st := memo_update a b res st in
    (res,st).

  Fixpoint combinator depth :
      expr -> expr -> state -> option (expr * state) :=
    fun a b st =>
    match depth with
    | 0 => None
    | S depth =>
      match a,b with
      | F, _ => fx b st
      | _, F => fx a st
      | T, _ => tx b st
      | _, T => tx a st
      | N na, N nb =>
        match memo_get na nb st with
        | Some p => Some (p,st)
        | None =>
          do nodea <- get na st.(graph);
          do nodeb <- get nb st.(graph);
          let '(l1,v1,h1) := nodea in
          let '(l2,v2,h2) := nodeb in
          match Pos.compare v1  v2 with
          | Eq =>
            do x, st <- combinator depth l1 l2 st;
            do y, st <- combinator depth h1 h2 st;
            Some (memo_node na nb x v1 y st)
          | Gt =>
            do x, st <- combinator depth l1 b st;
            do y, st <- combinator depth h1 b st;
            Some (memo_node na nb x v1 y st)
          | Lt =>
            do x, st <- combinator depth a l2 st;
            do y, st <- combinator depth a h2 st;
            Some (memo_node na nb x v2 y st)
          end
        end
      end
    end.
End combinator.
```

Figure 9.9: Deep combinator for binary operations

the resulting expression is correct. For the sake of clarity, we will not expose these hypotheses nor the resulting correctness theorem here, and refer the interested reader to the Coq development[10].

Implementing the conjunction

However, we demonstrate the use of this combinator on the particular example of the `and` function; all other binary operations follow the same pattern. First, we have to define a function `upd_and` that updates the memoization state: it is simply a wrapper that adds an element to the right memoization table, and leaves the others untouched. Then, we define the function `mk_and` as a simple call to the binary combinator. The crux here is the choice of the functions `Fx` and `Tx` that specify the behavior of the combinator at the leaves of the DAG.

```
Definition upd_and na nb res (st : state) :=
  mk_state st
  {| mand := set (na,nb) res st.(mand);
     mor  := st.(mor);
     mxor := st.(mxor);
     mneg := st.(mneg)
  |}).

Definition mk_and :=
  combinator
    (fun x st => Some (F,st) )  (* Fx *)
    (fun x st => Some (x,st) )  (* Tx *)
    (fun a b st => get (a,b) st.(mand))
    upd_and.
```

Then, the semantic correctness theorem for this operation is defined as follows.

```
Theorem mk_and_correct depth v0 (st: state) a b :
  wf_st st ->
  wf_expr st v0 a -> wf_expr st v0 b ->
  ∀ res st', mk_and depth a b st = Some (res, st') ->
    wf_st st' /\ incr st st' /\
    wf_expr st' v0 res /\
    ∀ env va vb,
      value env st a va ->
      value env st b vb ->
      value env st' res (va && vb).
```

This statement proves three things under the hypotheses that the given state and BDDs are well-formed: first, `mk_and` returns a well formed state. Second, it manipulates the state monotonically (and, in particular, previously well-defined BDDs will still be well-defined and keep their semantic values). Third, the returned BDD has the expected semantic interpretations.

Canonicity

We can prove that this representation of BDDs is canonical: that is, well-formed equivalent expressions are mapped to the same nodes.

```
Definition equiv st e1 e2 :=
  ∀ env v1 v2, value env st e1 v1 ->  value env st e2 v2 -> v1 = v2.
```

[10] https://github.com/braibant/hash-consing-coq

```
Lemma canonicity st v e1 e2 :
  wf_st st -> wf_expr st v e1 -> wf_expr st v e2 ->
  equiv st e1 e2 -> e1 = e2.
```

From this result, it follows that the (non-recursive) `eqb` function from Figure 9.3 is a correct and complete characterization of semantic equivalence of expressions.

```
Lemma eqb_correct st v e1 e2 :
  wf_st st -> wf_expr st v e1 -> wf_expr st v e2 ->
  (eqb e1 e2 = true <-> equiv st e1 e2).
```

Garbage collection

In the above version of BDDs, we have not implemented garbage collection. That is, allocated nodes are never destroyed, until the allocation map becomes unreachable as a whole. Garbage collection could be added, e.g., using a stop and copy operation that preserve a set of roots. This is beyond the scope of this paper.

9.3.2. The PURE-SHALLOW Approach

The previous implementation uses a *deep embedding* of the representation of the BDD in memory via the `graph` map. This is a natural way to encode a directed acyclic graph, but, as we saw, makes it difficult (if not unfeasible) to deal properly with termination. Therefore, we would like to be able to reason about BDDs as if they formed an inductive type, while keeping the ability to share sub-terms at runtime.

There is actually no need to look further than inductive types to do that. The standard intuition about inductive types is that they define the smallest type closed under application of constructors: the mental image that we get from that is a *tree*. Yet, there is nothing that prevent us from using the system to share sub-terms.

In this section, we demonstrate that we can encode BDDs in Coq using a representation that looks like binary decision *trees*, yet has runtime performances similar to the PURE-DEEP implementation (see Section 9.3.1), using explicit sharing. We present on Figure 9.10 our inductive definitions, and the associated allocation function `mk_node`. This has to be compared with Figure 9.3: the difference mainly lies in the deletion of the `graph` map, and the explicit recursive structure of the `expr` type.

The reader may wonder why we chose to define `expr` as two mutual inductive types `expr` and `hc_expr`. Indeed, we explain in [BJM14, Section 8.3] that it is possible to inline `hc_expr` inside the definition of `expr`. The mutual inductive solution is inspired by the hash-consing library in OCaml by Conchon and Filliâtre [FC06]. This library makes a clear distinction between hash-consing nodes (the equivalent of our `hc_expr` inductive) and the actual values that are hash-consed. In Coq, this makes some inductive proofs a little more involved: we need to use mutual induction on the two data-types.

The code of the `hc_node` function is subtle: a call to `hc_node e bdd` will perform a lookup in the hash-consing map `hmap`: if the same expression was previously allocated, then we return the old version; otherwise, we update the map `hmap` with a mapping from the expression to its hash-consed version. The lookup ensures that equal expressions (modulo the comparison function used to index `hmap`) are mapped to the same hash-consed expression. Then, `mk_node l v h bdd` will first test the identifiers of `l` and `h` for equality: if it is the case, then there is no need to introduce a new node; otherwise, we perform a call to `hc_node`.

As an example, assume that `x` and `y` are expressions with different identifiers. The following code

```
Inductive expr := | F | T | N : hc_expr -> var -> hc_expr -> expr
with hc_expr := HC : expr -> positive -> hc_expr.

(* Two extra definitions that are used as coercions in the
   following code *)
Definition unhc (e : hc_expr) := let 'HC res _ := e in res.
Definition ident (e : hc_expr) := let 'HC _ res := e in res.

Definition eqb a b :=
  match a,b with
  | T,T | F,F => true
  | N (HC _ id1) x (HC _ id2), N (HC _ id1') x' (HC _ id2') =>
    (id1 =? id1') && (x =? x') && (id2 =? id2')
  | _, _ => false
  end.

Record hashcons_state := {hmap : expr ⇝ hc_expr; next : positive}.

Definition alloc u st :=
  let r := HC u b.(next) in
  (r, {| hmap := set u r b.(hmap);  next := b.(next) + 1|}).

Definition hc_node (e : expr) (st : hashcons_state) :=
  match get e st.(hmap) with
  | Some x => (x, st)
  | None => alloc e st
  end.

Definition mk_node (l : hc_expr) (v : var) (h : hc_expr) st :=
  if (ident l =? ident h) then (l,st) else hc_node (N l v h) st.
```

Figure 9.10: A shallow-embedding of sharing

```
let '(a,st) := mk_node x v y st in
let '(b,st) := mk_node x v y st in
let '(c,st) := mk_node a v' b st in ...
```

will make `a`, `b` and `c` point to the same memory location! However, if `x` and `y` were not shared maximally, then neither are `a`, `b` nor `c`.

A word on memoization

There is no difference at all in the way we handle memoization in this implementation w.r.t. Section 9.3.1. That is, we implement the same `state` record as before; and pass the same memoization tables around.

A word on termination

It is now easier to define recursive functions that operate on BDDs by taking advantage of the inductive definition of `expr`. We have to stress that *easier* is not *easy* because the Coq termination checker requires that recursive calls are made on a structurally smaller argument: there is no built-in support for recursive definitions with pairs of arguments that are decreasing w.r.t. a lexicographic order. Therefore, we have to use nested fixpoints or prove that the recursive calls are well-founded. In this section, we choose the former.

Re-implementing the combinator

We are now ready to describe the code of the implementation of the binary combinator presented in Figure 9.11. The code is similar to the one in Figure 9.9 with a few key differences: thanks to the inductive definition of expressions, we do not have to perform lookups in the `graph` map and we do not use fuel anymore. This makes the combinator function *total*; and we can get rid of the Maybe monad.

Canonicity

Again, we prove that this representation of BDDs is canonical: well-formed equivalent expressions are mapped to the same nodes. Again, we have the corollary that the (non-recursive) equality test from Figure 9.10 that inspects the (top-level) node identifiers is a complete and correct characterization of semantic equivalence:

```
Definition equiv e1 e2 :=
  ∀ env v1 v2, value env e1 v1 -> value env e2 v2 -> v1 = v2.

Lemma eqb_correct st e1 e2 v :
  wf_st st -> wf_expr st v e1 -> wf_expr st v e2 ->
  (eqb e1 e2 = true <-> equiv e1 e2).
```

Comparison with the previous approach

The implementation presented in this section is derived from the previous one, with the following improvements: the proofs are roughly 20% shorter; the performances are slightly better when executing the code inside Coq (there is less administrative book-keeping to do in our data structures); the functions that operates on BDDs are total. Furthermore, it would probably be easier to implement garbage collection in this setting than in the previous one, thanks to the simpler definition of the global state.

The situation is almost ideal for the equality test: we prove that the (non-recursive) equality function that inspects the top-level identifiers of nodes is a correct and complete characterization of semantics equivalence of BDD expressions. However, we have no way to prove that it corresponds to physical equality. Actually, we cannot *state* within Coq that it is never the case that two identical representations of the same term coexists, even if we could argue at a meta-level that it is indeed not the case.

9.4. From Pure Data Structures to Persistent Data Structures Via Extraction

In the previous section, we use a state monad to store information about hash-consing and memoization. However, one can see that, even if these programming constructs use a mutable state, they behave transparently with respect to the pure Coq definitions.

We have seen earlier (see Section 9.2.2) that if we abandon (efficient) executability inside Coq, we can express new idioms. In the following, we implement the BDD library in Coq as if manipulating decision trees with neither sharing, nor hash-consing tables, nor memoization tables, then add the hash-consing and memoization code by using special features of the extraction mechanism to remap some constants arbitrarily to custom OCaml code.

```
Section combinator.
  Variable fx : hc_expr -> state -> hc_expr * state.
  Variable tx : hc_expr -> state -> hc_expr * state.
  Variable memo_get : positive -> positive -> state -> option (hc_expr).
  Variable memo_update :
    positive -> positive -> hc_expr -> state ->  state.

  Fixpoint combinator : hc_expr -> hc_expr -> state -> hc_expr * state :=
    fun a =>
    fix combinator_rec b st :=
    match unhc a, unhc b with
    | F, _ => fx b st
    | _, F => fx a st
    | T, _ => tx b st
    | _, T => tx a st
    | N l1 v1 h1, N l2 v2 h2 =>
      let ida := ident a in
      let idb := ident b in
      match memo_get ida idb st with
      | Some p => (p,st)
      | None =>
        let '(res, st) :=
          match Pos.compare v1 v2 with
          | Eq =>
            let '(x, st) := combinator l1 l2 st in
            let '(y, st) := combinator h1 h2 st in
            mk_node x v1 y st
          | Gt =>
            let '(x, st) := combinator l1 b st in
            let '(y, st) := combinator h1 b st in
            mk_node x v1 y st
          | Lt =>
            let '(x, st) := combinator_rec l2 st in
            let '(y, st) := combinator_rec h2 st in
            mk_node x v2 y st
          end
        in
        let '(_, st) := memo_update ida idb res st in
        (res, st)
      end
    end.
End combinator.
```

Figure 9.11: Shallow combinator for binary operations

```
Inductive bdd: Type :=
| T | F | N : var -> bdd -> bdd -> bdd.

Extract Inductive bdd =>
  "bdd" ["hT" "hF" "hN"] "bdd_match".
```

(a) Coq side

```
type bdd =
| T of tag | F of tag | N of positive * bdd * bdd * tag

module HCbdd = Hashcons.Make(...)

let hT = HCbdd.hashcons (T Weakhtbl.dummy_tag)
let hF = HCbdd.hashcons (F Weakhtbl.dummy_tag)
let hN (v, t, f) = HCbdd.hashcons (N (v, t, f, Weakhtbl.dummy_tag))

let bdd_match fT fF fN t =
  match t with
  | T _ -> fT ()
  | F _ -> fF ()
  | N (v, t, f, _) -> fN v t f
```

(b) OCaml side

Figure 9.12: Implementing BDDs in Coq, extracting them using smart constructors

9.4.1. The SMART Approach

More precisely, we define our BDDs as binary decision trees (see top of Figure 9.12), and implement operations in Coq on this simple data structure. Then, we tell Coq to extract the `bdd` inductive type to a custom `bdd` OCaml type (see bottom left of Figure 9.12) and to extract constructors into *smart constructors* that maintain the maximal sharing property. The type defined in OCaml is identical to the Coq one, except that it carries one extra field of type `tag`, morally containing the associated unique identifier. The smart constructors make use of the hash-consing library used in Why3, a recent version of a library by Conchon and Filliâtre [FC06]. It defines the `Hashcons.Make` functor, that we instantiate. The generated module provides a `HCbdd.hashcons` function that returns a unique hash-consed representative for the parameter.

The reader may notice that we choose to name `bdd` in Coq what is clearly a representation of a binary decision tree, and which corresponds to what was previously named `expr`. We believe that this particular choice of name makes sense if we consider values of type `bdd` to represent Boolean functions.

Discussion: the status of the equality test

In Coq, we define the obvious recursive function `bdd_eqb` of type `bdd -> bdd -> bool`, that decides structural equality of BDDs. Then, we extract this function into OCaml's physical equality:

```
Fixpoint bdd_eqb (b1 b2:bdd): bool :=
  match b1, b2 with
  | T, T | F, F => true
  | N v1 b1t b1f, N v2 b2t b2f =>
    Pos.eqb v1 v2 && bdd_eqb b1t b2t && bdd_eqb b1f b2f
```

```
  | _, _ => false
  end.

Extract Inlined Constant bdd_eqb => "(==)".
```

From a meta-level perspective, we argue that the two are equivalent thanks to the physical unicity of hash-consed structures, provided that all values are constructed using our smart-constructors, which is the case if we create all nodes from code extracted from Coq (of course, handwritten OCaml code may break this invariant). The key point here is that the way we build terms enforces the fact that the BDDs they build are maximally shared.

Keeping the BDD reduced.

One could be tempted to put in the OCaml code of the smart constructor `hN` a test that would enforce that BDDs are reduced. That is, it would not build a node if its two children were identical.

```
let hN (v,t,f) =
  if t == f then t
  else HCbdd.hashcons (N (v, t, f, Weakhtbl.dummy_tag))
```

To see the problem with this idea, consider the following Coq code.

```
match N (v,T,T) with
| T => false
| N (_,_,_) => true
end
```

In Coq, the above expression reduces to `true` while the extracted version would reduce to `false`. Therefore, this idea is wrong: avoiding the node construction makes subsequent case analyses behave inconsistently between Coq and OCaml.

In the end, the correct way to implement reduction is to use the following helper function, written in Coq, that builds a node only when necessary.

```
Definition N_check (v : var) (bt bf : bdd) : bdd :=
  if eqb bt bf then bt else N v bt bf.
```

Note that the user is forced to use this function instead of using the `N` constructor[11]. Failing to do so will result in code that is correct, but does not build *reduced* binary decision diagrams. More precisely, extracting `eqb` to `==` would make it possible to prove that BDD operations are semantically correct, but this would make diagrams non-canonical: well-formed expressions could be represented by several different nodes.

Implementing the combinator

The last ingredient needed to transform a decision tree library into a BDD library is memoization. We use the same kind of ideas: we define our functions as if not memoized, but we use a special well-founded fixpoint combinator, that we extract to a memoizing fixpoint combinator. The details can be seen on Figure 9.13: we declare an abstract type class `memoizer` of types `A` such that we know how to memoize functions of type `∀ x : A, P x`. This is extracted in OCaml to the type of polymorphic fixpoint combinators, with an extra technical `int` parameter used to specify the initial size of the used hash map. In Coq, we then define a memoizing combinator `memo` and a memoizing fixpoint combinator `memo_rec` as if they were not using memoization, but we ask the extraction mechanism to map them

[11] Still, the `N` constructor that occurs in the definition `N_check` must be extracted using a smart-constructor.

```
Parameter memoizer : ∀ A : Type, Type.
Existing Class memoizer.
Extract Constant memoizer "'key" => "'key Helpers_common.memoizer".

Definition memo [A] {H : memoizer A} [P] := @id (∀ x : A, P x).
Extract Inlined Constant memo => "Helpers_common.memo".

Definition memo_rec [A] {H : memoizer A} := @Coq.Wf.Fix A.
Extract Inlined Constant memo_rec => "Helpers_common.memo_rec".
```

(a) Coq side

```
type 'key memoizer =
    { memo : 'a. int -> (('key -> 'a) -> ('key -> 'a)) -> ('key -> 'a) }

let memo m f = m.memo 5 (fun _ x -> f x)

let memo_rec m f = m.memo 5 (fun frec x -> f x (fun y _ -> frec y))
```

(b) OCaml side

Figure 9.13: Memoizing combinators

to special OCaml functions, that make use of the type class instance given as parameter. These functions are observationally equivalent to the Coq ones, provided that the type class instance is correct, and that the memoized function is pure.

It is worth noting that directly exposing the type `'key memoizer` in Coq would be unsound, because this allows to use its instances in a non-terminating manner: the `memo_rec` wrapper makes sure the recursive calls are well-founded. Moreover, it is important to understand that we have not axiomatized these combinators. Instead, we give real implementations, semantically equivalent to their OCaml counterparts. This is important, because it then becomes clear that we do not introduce any logical inconsistencies in Coq, and these terms keep a computational contents which could be very useful in proofs.

This part of the code is modular, and can be used to memoize functions of any domain. It is up to the user to give instances of the `memoizer` type class: to do so, he can provide dedicated code, as shown in Figure 9.14 for the type `N` of binary natural integers, using simple hash tables. Alternatively, for the `bdd` type, our type class instance is just an OCaml wrapper around the hash-consing library built-in memoization mechanisms.

Again, we use the example of the `bdd_and` operation, shown in Figure 9.15. As in Figure 9.11, we define this function using two nested fixpoints, in order to handle the special recursion scheme of this function (decreasing on one of its two parameters). The definition of `bdd_and` uses `memo_rec` twice, in a nested fashion.

Garbage collection

The strategy we use in Section 9.3.2 for garbage collection is very naive: we kept everything alive, forbidding any garbage collection. Here, the hash-consing library we use allows the garbage collector to reclaim any node that is not referenced by the program, by using suitable weak hash tables. Moreover, it reclaims any memoized value that is associated with a dead key (we will not give more details here and refer the reader to [FC06]). Although it avoids memory leaks, this strategy does not necessarily give the best performances (see Section 9.5). We believe the changes necessary to implement a new strategy are small, and

```
Parameter memoizer_N : memoizer N.
Existing Instance memoizer_N.
Extract Inlined Constant memoizer_N => "Helpers_common.memoizer_N".
```

(a) Coq side

```
module NHT =
  Hashtbl.Make (struct
    type t = coq_N
    let equal = (=)
    let hash = N.to_int
  end)

let memoizer_N =
  { memo = fun n f ->
      let h = NHT.create n in
      let rec aux x =
        try NHT.find h x with Not_found ->
          let r = f aux x in NHT.replace h x r; r
      in aux }
```

(b) OCaml side

Figure 9.14: An instance of `memoizer` for `Coq.Numbers.BinNums.N`

do not involve rewriting the proofs: only OCaml code is involved.

It is important to note that we have to memoize the functions one argument at a time (keeping them curried). Indeed, one should be tempted to memoize a function `bdd * bdd -> bdd`. This is not a good idea, since we use weak hash tables for the memoization: in this case, the pairs containing the arguments are no longer accessible after the function call, so that the garbage collector is going to collect most of the memoized data at each cycle. Using our pattern, we make sure a memoized datum is not reclaimed as long as both parameters are still alive in memory. (Note that if we choose to use regular hash tables instead of weak ones, this argument does not stand anymore, but the program has different performance and memory consumption profiles.)

Discussion: comparison with previous approaches

In this instance of our BDD library, all Coq definitions are kept simple and proofs are straightforward. That is, we can prove semantic correctness of all operations directly using structural induction on decision trees and there is no state holding structures. There is a nice separation between the hash-consing and memoization code, that is generic, written in OCaml and not proved and the BDD code, mostly written and proved in Coq.

We do not need to use monads in the Coq code, so that the interface of the BDD library remains modular and easy to use. Moreover, it is straightforward to implement garbage collection strategies in order to avoid memory leaks.

9.4.2. The SMART+UID Approach

The previous SMART approach totally hides the unique identifiers from the Coq code. Yet, exposing these unique identifiers may be useful at times.

Consider the following use case: from a BDD B we would like to build an equivalent propositional logic formula *of linear size*, for instance for feeding into a satisfiability mod-

```
Definition bdd_and : bdd -> bdd -> bdd.
refine
  (memo_rec (well_founded_ltof _ bdd_size) (fun x =>
     match x with
     | T => fun _ y => y
     | F => fun _ _ => F
     | N xv xt xf => fun recx =>
       memo_rec (well_founded_ltof _ bdd_size) (fun y =>
         match y with
         | T => fun _ => x
         | F => fun _ => F
         | N yv yt yf => fun recy =>
           match Pos.compare xv yv with
           | Eq => N_check xv (recx xt _ yt) (recx xf _ yf)
           | Lt => N_check yv (recy yt _) (recy yf _)
           | Gt => N_check xv (recx xt _ y) (recx xf _ y)
           end
         end)
     end));
  unfold ltof; simpl; clear; abstract omega.
Defined.
```

Figure 9.15: Conjunction on bdds

ulo theory solver. In order to avoid an exponential blow-up, each shared sub-tree should generate one single sub-formula, used in a "let" binder so that its value can be used in multiple occurrences. The obvious way to implement such a transformation is to first detect which sub-trees are shared, using a set of shared subtrees seen so far, then to perform the transformation, using a table of mappings from subtrees to bound variables.

It seems therefore highly desirable to be able to build sets and maps over our hash-consed type. Generic functional sets and maps are usually implemented using balanced trees over a totally ordered data type; for efficiency, the comparison function should be very fast. An obvious choice would be to expose the unique identifiers to the Coq code (through a function `bdd -> uid`), or at least the total order that they induce (through a function `bdd -> bdd -> comparison`).

Unfortunately, doing so without precautions can lead to unsoundness. Consider a program where, in succession, two nodes A and B are allocated, then a node A' isomorphic to A is created; let $u_A < u_B$, $u_{A'}$ be the successive unique identifiers. If A is collected between the allocations of B and A', then A' will be allocated, with $u_{A'} > u_B$. Yet, A' and A are, from the point of view of the Coq code, identical; thus $u_{A'} = u_A$, yielding an inconsistency.

The workaround is to use a normal hash table, as opposed to a weak hash table, which prevents the collection of unreachable nodes. Then, two identical nodes created within the same execution are necessarily physically equal and thus share the same identifier.

One difficulty remains. Gallina is a purely functional language; the evaluation of a given term always yields the same result, and one expects the same property to extend to the extracted OCaml program, as long as it does not interact with the external world (e.g., reading from files). Yet, this is not necessarily the case if one exposes the unique identifiers. Consider a program P_1, such that the extracted OCaml code allocates two nodes A and B in this order. If P_1 is run stand-alone, then $u_A < u_B$. Yet, if another program P_2 allocating B' isomorphic to B is first run, and B' is not collected, then B' is the same as B and $u_B = u_{B'} < u_A$.

It seems questionable that the result of an evaluation should depend on whether some other (unrelated) evaluation has taken place, if only because it makes debugging difficult.

12

9.5. Comparison

In this section, we compare our design patterns on various aspects:

Executability inside Coq

The PURE-DEEP and PURE-SHALLOW implementations can be executed inside Coq, and have decent performances. The SMART and PHYSEQ approaches can also be executed inside Coq, but have dreadful performance because they do not exploit sharing. The SMART+UID approach cannot be executed inside Coq.

Trust in the extracted code

Unsurprisingly, the SMART and the SMART+UID approaches yield code that is harder to trust, while the PURE-DEEP and PURE-SHALLOW approaches leave the extracted code pristine. The PHYSEQ approach lies in between: it consists in a small tweak in extraction, that is easier to trust than the hash-consing library of SMART and SMART+UID.

Proof

From a proof-effort perspective, the SMART approach is by far the simplest. The SMART+UID approach involves the burden of dealing with axioms. The PHYSEQ approach requires proving that the result of a physical equality test does not change the result of the computation. Apart from that, the `physEq` term is convertible to a very simple term, so that the proofs are not affected by its use. By comparison, the PURE-DEEP and PURE-SHALLOW approaches required considerably more proof-engineering in order to check the validity of invariants on the global state. Note however that our proof arguments are much simpler in the latter one.

Garbage collection

Implementing (and proving correct) garbage collection for the PURE-DEEP or PURE-SHALLOW approaches would require a substantial amount of work. By contrast, the SMART and PHYSEQ approaches make it possible to use OCaml's garbage collector to reclaim unreachable nodes "for free".

Operations

The PHYSEQ approach cannot implement full hash-consing, so that we were not able to implement BDD algorithms. As we have shown, binary operations can be handled with a single parametric combinator. All our implementations (except `physEq`) use such a combinator to implement conjunction, disjunction, exclusive-or and so on. There is little work to

[12] One could argue that, with certified programs in the Coq fashion, there is no need to debug: each function or module comes with a proof of its correctness, which compositionally entail the correctness of the whole program. Yet, commonly one only proves the results to be *correct*, not necessarily *optimal*, and one proves very seldom that the computation has the expected complexity. Furthermore, some computations are split between an untrusted solving procedure, and a trusted checker; a failed check entails having to debug the untrusted procedure, which may be hard if behaviors are hard to reproduce independently of the rest of program. These two situations appear in Verasco: the only proofs concern soundness, and no formal guarantees are given about completeness; and some parts, like the polyhedral domain, are written in untrusted OCaml code.

do to add support for any binary operation we may have overlooked. The situation is more complicated when it comes to ternary operations (such as the if-then-else Boolean operation). Implementing it using the SMART approach, and proving it correct requires around 80 lines of code. It would require a non-trivial amount of work to implement it using our PURE-DEEP or PURE-SHALLOW approaches, and we have not conducted this experiment yet. Other operations that are relevant in a BDD library are function composition and quantification. We have yet to implement these in any of our libraries. Quantification over one variable can be defined as a structural recursion over one BDDs. As such, we think it can be easily defined in all our frameworks. Unary composition (that is, substitution of one variable with another Boolean function) can be defined as a structural recursion over two BDDs. As a consequence, implementing it in the SMART framework seems simple. However, this particular recursion scheme does not fit the combinator of the PURE-DEEP or PURE-SHALLOW approaches, and some additional work would be needed in order to implement it using those approaches. Moreover, a non-negligible amount of work would be needed to support variable arity composition and quantification in all libraries.

Performances of the Extracted Code

We evaluate the performances of the OCaml code that is extracted from our "pure" (see Section 9.3) and "impure" (see Section 9.4) libraries, and we pit them against a reference library implemented in OCaml (available from J.C. Filliâtre's web page). No BDD library was implemented using PHYSEQ, so that we won't include it in the comparison. This reference library does not keep the memoization table alive from one execution of an operation to another: for instance, each time the `and` function is called, a new memoization hash table is allocated. Therefore, to make up for a fair benchmark with our implementations, we modified this reference library to use a memoization strategy closer to ours. This alternative reference implementation is designed "reference (conservative)" in what follows.

Then, this evaluation, we use two standard benchmarks [VGLPAK00]. The first one is Urquhart's formula $U(n)$ defined by

$$U(n) \triangleq x_1 \iff (x_2 \iff \ldots (x_n \iff (x_1 \iff \ldots (x_{n-1} \iff x_n))))$$

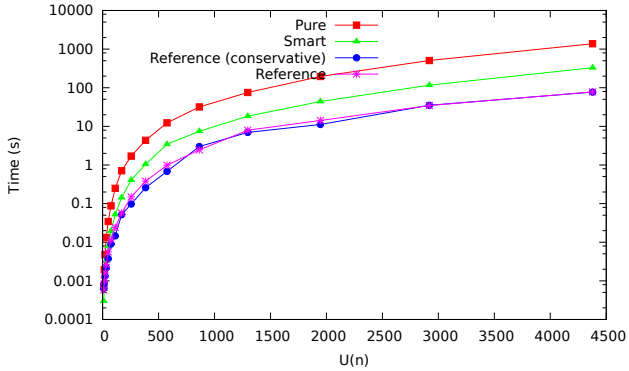
The second kind of formula states the pigeonhole principle $P(n)$ for $n+1$ pigeons: that is, if there is $n+1$ pigeons in n pigeonholes, then there is at least one hole that contains two pigeons. The plots of the execution time required to check that these formulas are tautology are given in Figure 9.16.

First, we remark that the reference implementation that we use is (almost always[13]) 4 times faster than our best implementation SMART. There is no significant difference in execution time between the reference implementation with the conservative memoization strategy and the reference implementation. We have not conducted a detailed measurement of the memory consumption[14], though, and the memory consumption profile is likely to be different between the two.

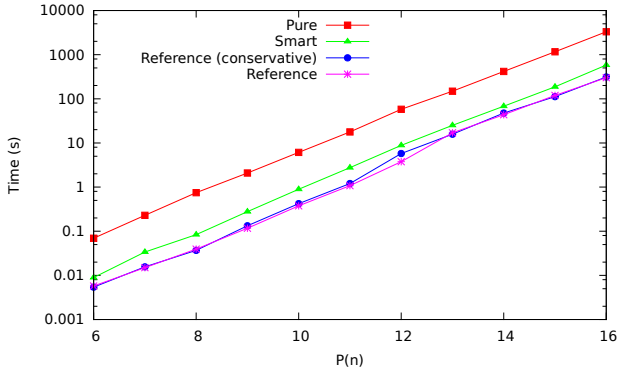
Then, our SMART implementation is roughly 4 times faster than the PURE-DEEP one: interestingly enough this value is consistent over our benchmark, while we could have expected logarithmic factors to show up. Indeed, recall that the pure implementation uses functional finite maps, while the smart one uses OCaml hash-maps.

[13] This is not necessarily true for small formulas due to differences in the start-up costs of these libraries.

[14] These particular benchmarks were run on a computer with 1 TB of ram, which made it possible to avoid swapping during our experiments. Indeed, memory usage is the limiting factor on bigger instances of the tests.



(a) Urquhart's formula



(b) Pigeonhole principle

Figure 9.16: Execution time for the BDD benchmarks

Also, we have run the same benchmark on our PURE-SHALLOW implementation, and the resulting plot is close to the PURE-DEEP plot: for small instances, the difference of execution times are of about 10%. For our largest instance of the pigeonholes problem, we observe that the PURE-SHALLOW is faster by about 30%. We do not delve too much on this result: we wonder for instance to what extent changing the implementation of finite maps that we use could result in similar differences of performances.

Primary investigation shows that the main performance difference between the reference implementation and our SMART approach comes from the use of the generic hash-consing library: the reference library uses a specialized hash-consing system, much faster than the generic library we use. Similarly, the difference between the SMART and the PURE-SHALLOW seems to be the use of AVL trees to represent dictionaries.

We did not investigate much the memory behavior. Our simple experiments show that it depends on the benchmark we consider. The bottom line is that all our approaches consume less than twice as much memory as the reference implementation, conservative version. Moreover, we observe that PURE-SHALLOW consumes 10%-20% less memory than PURE-DEEP. Concerning the SMART approach, it seems like our curried memoization scheme creates a lot of very small hash tables in the OCaml heap, which is not the optimized use case of OCaml hash tables. Thus, there might be room for improvement here.

9.6. Related Work

Purely functional implementations of BDDs

Bradley and Davies [BD98] implemented *lazy* BDDs in Haskell in a manner similar to our "pure-shallow" approach. In their approaches, not only are BDDs maximally shared, they are also lazily evaluated "on demand". We did not investigate this possibility.

Imperative features inside the specification language

An obvious solution to implement and reason about imperative algorithms is to have these imperative features present in the modeling language of the prover. Some provers directly target high-level programming languages with data structures, references and imperatives features: an example is KeY [BHS07], which targets a large subset of Java. While such features are not available in Coq, there are two conceptual difficulties with this approach that would have made it impractical in our case-studies. First, BDD algorithms implemented using hash-consing are functional in a high-level view: BDD operations are very clearly given functionally, by induction; but also because hash-consing is suitable only for immutable structures[15]. It therefore seems strange to have to program them in an imperative language, furthermore one that complicates common functional idioms (e.g., pattern-matching). Second, the meta-theory for such languages is typically huge, with intricate proof rules having to deal with mutable data, references, late binding etc. It is not obvious how much we can trust the proof system with respect to the semantics of the language.

A second option is to add to an existing functional specification language certain imperative traits as *monads*, then modify the extraction function from the specification language to the target language so as to translate monadic operations into imperative calls, as has been done for Isabelle/HOL [BKH+08]; this approach has been used to verify a BDD package [GS12]. Special proof idioms have to be used for monadic programs, in addition to the general difficulty of programming in monadic style (see Section 9.2.2). To the best of our knowledge, this approach has not been investigated in Coq.

A third option is to integrate into the specification language some essentially imperative data structures (e.g., mutable arrays, from which hash tables can be implemented), but present them in a functional fashion (e.g., an update to a mutable array is treated as returning a new array, same as the previous except for the updated location). The imperative features are then implemented efficiently for evaluation of expressions inside the prover, and are mapped to native imperative features of the target language during extraction. For instance, an experimental version of Coq exists, with native integers and arrays [AGST10]. Again, all difficulties with monads (see Section 9.2.2) apply, plus there is the problem of running a nonstandard version of Coq.

A fourth option is to *deeply embed* a subset of an imperative language into Coq: the programs of this imperative language are given a semantics inside Coq, and correctness properties are proved with respect to this semantics. This idea is discussed by Vafeiadis [Vaf13], but, as he remarks, this largely precludes the use of regular proof tactics: we have to develop proof steps specific to the language being embedded, and prove these steps correct with respect to the semantics.

[15] Or at least for structures behaving as though they were immutable; for instance, we can perform hash-consing on a structure if the mutable information in the structure is just used for caching and does not affect the hashcode.

Verification of BDD algorithms in Coq

Verma et al. [VGLPAK00, VGL00] implemented and proved correct in Coq a BDD library featuring efficient negation and disjunction; other operations like conjunction, implication and so on are implemented as derived operations. The BDDs produced are reduced and shared.

As we said, we chose to implement a fresh one because the code associated to their paper did not age well w.r.t. the evolution of Coq. Beside the fact that they investigated garbage collection, there is no conceptual difference between their library and our PURE-DEEP approach.

Verification of BDD algorithms in other theorem provers

In Isabelle/HOL, Ortner and Schirmer [OS05] verified the implementation of a normalization algorithm for binary decision diagrams. That is, their algorithm takes as input a BDD, and outputs the corresponding ROBDD. Their formalization is built on the Burstall-Bornat memory model: they build one heap of type `ref` $\rightsquigarrow$ `value` for each component of a BDD node, with `ref` the abstract type of memory addresses. Using a split-heap model makes it easier to reason about heap-allocated data structures in tools such as Why3. We believe however, that our formalization is more direct than theirs, and more suitable for efficient implementations in Coq.

Boyer and Hunt [BHJ06] developed an extension of ACL2 that uses hash-consing to give canonical representatives to ACL2 objects. This makes it possible to memoize some ACL2 user defined functions. As a case study, they implemented a BDD library in ACL2. Remark that this implementation is based on the fact that ACL2 exposes hash-consing primitives and the associated reasoning principles to the user. It is unclear to what extent these features could be added to the Coq proof assistant.

Hash-consing in the execution language

Jean Goubault-Larrecq's HimML programming language [Gou94a, Gou94b] is an extension of core Standard ML with primitive finite set and map data types with a run-time system designed around the concept of maximal sharing, or systematic hash-consing.

Chapter 10

Conclusions

10.1. Achievements

In this thesis, we described the design, implementation and formal verification of Verasco, a static analyzer based on abstract interpretation using the Coq proof assistant, available at `http://compcert.inria.fr/verasco/`. This static analyzer is able to establish the absence of erroneous behavior in programs written in a large subset of C99. Verasco shares the formal semantics of its input language with the CompCert formally verified compiler, and hence the guarantees it provides provably extends to the assembly code generated by CompCert. Our static analyzer enjoys a modular architecture with well-specified interfaces between its different components. It includes an abstract interpreter operating on the source code, a state abstract domain handling the memory structure of the program, and a complex hierarchy of numerical abstract domains able to infer complex numerical properties. The numerical hierarchy contains a generic abstract domain functor for handling the boundedness of integers, a symbolic abstract domain, the interval and congruence non-relational abstract domains, the Octagon abstract domain and a third-party Polyhedron abstract domain. This combination of numerical domains builds on a cooperation mechanism, inspired from that of Astrée, which makes possible the exchange of information between domains without breaking their modularity.

We presented several contributions that could be reused independently of the formal verification of a static analyzer: we developed novel techniques to prove properties of our specially crafted Hoare logic; we designed new algorithms for the octagon abstract domain, enabling the static analyzer to operate on sparse representations of abstract environments; and we investigated several methods to implement hash-consing and memoization in programs written using the Coq proof assistant.

10.1.1. Formalization Effort

The full Coq development of Verasco is about 47000 lines long, excluding blanks and comments. An additional 6000 lines of Caml implement the operations over polyhedra that are validated a posteriori (see the work by Fouilhé et al. [FMP13, FB14]). The parts reused from CompCert (e.g., syntax and semantics of C#minor) and Flocq are not counted. The following table summarizes the relative sizes of the various components of Verasco:

	Code & spec.	Proofs	Total	
Abstract domain interfaces	1234	143	1377	3%
Abstract interpreter	1667	1631	3298	7%
State abstraction (from [Lap15])	4396	5793	10189	22%
Numerical domains	12336	14459	26795	57%
Machine arithmetic functor	912	1536	2448	5%
Abstract domain combinators	715	346	1061	2%
Intervals	1698	2493	4191	9%
Congruences (from [Lap15])	682	1003	1685	4%
Octagons and linearization	1349	2815	4164	9%
Polyhedra (from [FMP13, FB14])	5947	4258	10205	22%
Symbolic equalities	1033	2008	3041	7%
Miscellaneous libraries	2946	1909	4855	10%
Total	**22579**	**23935**	**46514**	

The largest parts of the development include the state abstract domain, owing to the complexity of the memory abstraction, and the numerical abstract domain hierarchy, with many abstract domains, some of which containing difficult proofs about integer and floating-point arithmetic.

The difficulty of the development of such a large formally verified program not only resides in the complexity of the algorithms and their proofs: several times, we faced software engineering issues, such as the need for large refactorings because of wrong early design decisions or for staying compatible with newer CompCert and Flocq releases.

10.1.2. Expected Impact

Even if the basic building blocks of an industrial-scale static analyzer such as Astrée relies on strong mathematical foundations, their implementation implies, in practice, some informal reasoning. In this thesis, we give formal specifications and proofs to the various components of the static analyzer. Therefore, we "filled the gap" between the formal description of an abstract domain in an academic paper and its implementation. Hence, we hope that this work will be useful to developers of unverified static analyzers, by giving them an example of formal specification of abstract domains in the context of a real static analyzer.

By demonstrating that formal verification tools are mature enough to develop realistic formal verification tools themselves, we consider this work, together with other recent achievements in the domain, as a new milestone. Not only we showed that static analysis techniques can be formalized, but we also proved that the overhead of their formal verification in the context of realistic, complex tools is no longer necessarily prohibitive in terms of implementation and formalization effort.

10.2. Perspectives and Future Work

Verasco is still an ongoing experiment: much work is still needed before scaling up to analyzing real, large, industrial software. We now outline possible future improvements for Verasco.

10.2.1. Using New Abstract Domains and Analysis Techniques

Trace partitioning. Trace partitioning [RM07] is an abstract interpretation technique consisting in partitioning the approximated set of states at a given program location depending on the trace that led to this state. This technique is used in static analyzers in order to improve the precision of the analysis in many cases. It supersedes loop unrolling, and can be used for many other purposes. In Verasco, trace partitioning should be implemented as an additional abstraction layer between the state abstract domain and the abstract iterator or even as part of the latter. A significant gain in precision in Verasco can be expected from using trace partitioning, at the cost of a potentially slower analysis.

Other numerical abstract domains. The numerical abstract domain hierarchy of Verasco can be extended with new abstract domains. If we follow the example of Astrée, we might be interested in implementing Boolean partitioning techniques, or an abstract domain dedicated to linear filters [Fer04] or arithmetic-geometric progressions [Fer05]. All these additions would improve the precision of the analysis, potentially decreasing its performance. On the other hand, we may want to implement other precision versus performance tradeoffs in linear relational abstract domains, such as Pentagons [LF08b] or Subpolyhedra [LL09].

Supporting Recursion and Dynamic memory allocation. Dynamic memory allocation and recursion are features of C pervasively used in non-critical code. Therefore, it seems important to support them in order to analyze a larger class of programs. However, these two features create difficult challenges in sound static analysis. Basic support is already difficult, but precise analysis of pointer-based dynamically allocated data structures demands more complex techniques such as shape analysis.

10.2.2. Improving Performance

We did preliminary experiments of testing Verasco on real programs. Specifically, we ran Verasco on several short unit tests designed to check that its abstract domains are indeed able to infer the desired properties. Additionally, we tested Verasco on numerical simulation programs using floating-point numbers, taken for the CompCert benchmark suite, called `nbody.c` and `almabench.c`. We also ran Verasco on cryptographic routines on elliptic curves, taken from the NaCl library. This latter benchmark is called `smult.c`. Here are the computation times needed by Verasco to prove the absence of erroneous behavior in these programs with our Intel Xeon E3-1240 at 3.4GHz (the memory consumption during the analysis, in all our tests cases, is negligible):

Program	Size (lines of C code)	Analysis time (seconds)
`smult.c`	347	11.8
`nbody.c`	176	6.3
`almabench.c`	362	5.3

These results are encouraging, first because they represent a significant improvement compared to previous versions of Verasco [JLB+15], and second because they show that analyses times for small programs are reasonable. Nevertheless, a lot of progress still needs to be done in order to approach the performance of industrial-strength static analyzers.

Summarizing arrays. The state abstract domain needs improvement to make it able to compute summaries of arrays instead of representing each cell independently. A simple method is to represent the whole arrays using one non-relational abstract value, on which we perform weak updates, but better method exists, involving *array segmentation* [CCL11].

Variable packing. Relational abstract domains, such as Octagons or Polyhedra, always have a higher computational cost than non-relational abstract domains, even with our sparse representations for Octagons. Therefore, they cannot be directly used in programs containing many variables. A technique called *variable packing* is often used in order to use the relational abstract domains only on specific *packs* of variables, thus limiting the high computational cost. This packing technique can be implemented in Verasco, but it would require some tuning to implement good heuristics to determine the contents of these packs.

Limiting the inefficiencies of Coq as a programming language. Implementing a static analyzer in Coq has many implications, leading to radical choices. Examples include the representation of integers as lists of bits in Coq, which is very inefficient, and the implementation of floating-point numbers using these inefficient integers to represent mantissas and exponents. Another example is the absence of side effects, and the lack of some data structures like arrays (which would be useful for implementing Octagons, typically). Some of these limitations of Coq as a programming language have been mitigated: we circumvent the absence of a physical equality operator (see Section 9.1), and we replace, at extraction time, slow Coq integers with a fast implementation of unbounded integers arithmetic for most integer operations. Others limitations still need work: we need to use a fast implementation of floating-point arithmetic[1], and improve the data structures used in the different components of the analyzer.

On a more general note, we wonder if the use of a deductive verification tool such as Why3 or CFML instead of Coq could ease the implementation by removing the limitations of Coq as a programming language. However, using Coq as a specification language is very convenient (and necessary due to the dependence over CompCert), and its logical power is essential for Verasco. Therefore, we wonder whether a methodology could be designed to get the best of both worlds, without endangering the formal guarantees provided by Verasco. As an example, this would make us able to use the results of the VOCaL ANR project [VOC] aiming at formally verifying algorithms and data structures.

Improving algorithms and removing redundant computations. Some algorithms used in Verasco are not optimal in terms of efficiency. For example, every numerical expression is analyzed at least twice: the first time, Verasco determines whether it could have an erroneous behavior, and the second time it does the actual analysis (e.g., abstract assignment, backward analysis for tests...). Similarly, when there are several pointer references in an expression (or array accesses, which is equivalent in C), the state abstract domain considers all possible combinations of memory references for each of these accesses. This can lead to a combinatorial explosion in the case of expressions with several memory accesses that cannot be statically resolved. Finally, when analyzing nested loops, the fixpoint iteration for the inner loop starts from scratch at every iteration of the outer loop. The inferred invariant at the first iteration of the outer loop could be reused as starting point for the inner loop iteration, leading to a shorter inner loop iteration.

[1] Using the host machine implementation of floating-point numbers is not as easy as it seems, because we need several different rounding modes, and many C compilers are not, in fact, fully compliant to the IEEE-754 standard [BJLM15].

All these inefficiencies could be mitigated by implementing better algorithms in Verasco. We do not foresee fundamental difficulties in doing so: this is mostly a matter of work.

10.2.3. Leveraging the Results of the Analyses

Currently, the result of the analysis that is guaranteed to be correct is just a Boolean: if the Boolean is `true` (i.e., there is no alarm), then the program has no erroneous behavior. Otherwise, no guarantee is given.

This is unfortunate, because Verasco actually computes much more about the program. Indeed, we could, without much additional work, return a certified annotated version of the program with the result of the analysis, including, e.g., bounds for numerical variables, or points-to information for pointers. This information could be reused by other analyzers or even advanced compiler optimizations.

Actually, there is no need for Verasco to prove the absence of undefined behavior in order to return such an annotated version of the program: with small modifications to Verasco, we could make it return valuable static analysis results even if it raises alarms. These results would be valid, provided that the input program does not have an erroneous behavior (or a behavior unsupported by Verasco, such as an external function call), just like any static analysis included in a modern compiler does.

Another kind of information that could be returned by Verasco is verified information about the trace generated by the program. In particular, it would be easy to change the abstract interpreter so that it returns an interval for each global volatile variable approximating the set of values written to this variable.

10.2.4. Experimenting Verasco on Real Code

Perhaps the most important ingredient missing to Verasco when compared to an industrial-scale static analyzer such as Astrée is a long phase of experiments on real industrial code for tuning the analyzers and adding to it the many small but necessary features needed to make it usable in a realistic context. This phase is very time-consuming, and tends to specialize the analyzer towards specific applications.

Bibliography

[AB07] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor. In *Theorem Proving in Higher Order Logics (TPHOL)*, volume 4732 of *LNCS*, pages 5–21. Springer, 2007. [Cited on pages 83 and 94.]

[Agd] The Agda language. `http://wiki.portal.chalmers.se/agda/`. [Cited on page 18.]

[AGST10] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with imperative features and its application to SAT verification. In *Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 83–98. Springer, 2010. [Cited on page 210.]

[Alt] The Alt-Ergo SMT solver. `http://alt-ergo.lri.fr/`. [Cited on page 16.]

[AM01] Andrew W Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):657–683, 2001. [Cited on page 94.]

[App14] Andrew W Appel. *Program logics for certified compilers*. Cambridge University Press, 2014. [Cited on pages 79 and 106.]

[ari96] Ariane 5 flight 501 failure. Report by the inquiry board, July 1996. [Cited on page 1.]

[BBC+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010. [Cited on page 43.]

[BC04] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development. Coq'Art: the calculus of inductive constructions*. Springer, 2004. [Cited on page 18.]

[BCC+02] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 85–108. Springer, 2002. [Cited on pages 2, 43, 122, and 181.]

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI '03*, pages 196–207. ACM, 2003. [Cited on pages 43 and 49.]

[BCJP09] Frédéric Besson, David Cachera, Thomas Jensen, and David Pichardie. Certified static analysis by abstract interpretation. In *Foundations of Security Analysis and Design V*, volume 5705 of *LNCS*, pages 223–257. Springer, 2009. [Cited on page 44.]

[BD98] Jeremy T. Bradley and Neil J. Davies. Compositional BDD construction: A lazy algorithm. Technical Report CSTR-98-005, Department of Computer Science, University of Bristol, 1998. [Cited on page 210.]

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Formal Methods*, volume 4085 of *LNCS*, pages 460–475. Springer, 2006. [Cited on pages 12 and 24.]

[BDP14] Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an ssa-based middle-end for compcert. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(1):4, 2014. [Cited on page 24.]

[Ber09] Yves Bertot. Structural abstract interpretation: A formal study using Coq. In *Language Engineering and Rigorous Software Development*, volume 5520 of *LNCS*, pages 153–194. Springer, 2009. [Cited on pages 44, 105, and 106.]

[BHJ06] Robert S. Boyer and Warren A. Hunt Jr. Function memoization and unique object representation for acl2 functions. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 81–89. ACM, 2006. [Cited on page 211.]

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNAI*. Springer, 2007. [Cited on page 210.]

[BHZ09] Roberto Bagnara, Patricia M Hill, and Enea Zaffanella. Weakly-relational shapes for numeric abstractions: Improved algorithms and proofs of correctness. *Formal Methods in System Design*, 35(3):279–323, 2009. [Cited on pages 146, 151, 154, 155, 158, 172, 173, 174, and 175.]

[BJLM13] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In *22nd IEEE Symposium on Computer Arithmetic (ARITH)*, pages 107–115, 2013. [Cited on pages 61 and 131.]

[BJLM15] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning (JAR)*, 54(2):135–163, 2015. [Cited on pages 61, 131, and 216.]

[BJM13] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. Implementing hash-consed structures in Coq. In *Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 477–483. Springer, 2013. [Cited on page 182.]

[BJM14] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. Implementing and reasoning about hash-consed data structures in Coq. *Journal of Automated Reasoning*, 53(3):271–304, 2014. [Cited on pages 182 and 198.]

[BJS15] Martin Bodin, Thomas Jensen, and Alan Schmitt. Certified abstract interpretation with pretty-big-step semantics. In *CPP '15*, pages 29–40. ACM, 2015. [Cited on page 45.]

[BKH+08] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with Isabelle/HOL. In *Theorem Proving in Higher Order Logics (TPHOL)*, volume 5170 of *LNCS*, pages 134–149. Springer, 2008. [Cited on page 210.]

[BLMP13] Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *Static Analysis (SAS)*, volume 7935 of *LNCS*, pages 324–344. Springer, 2013. [Cited on pages 44, 111, and 121.]

[BM11] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *20th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 243–252, 2011. [Cited on pages 27, 61, and 131.]

[Boo] The Boogie tool. `http://research.microsoft.com/en-us/projects/boogie/`. [Cited on page 16.]

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, pages 128–141. Springer, 1993. [Cited on pages 39, 40, 45, and 77.]

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, 1976. [Cited on pages 29, 42, 49, and 126.]

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252. ACM, 1977. [Cited on pages 29, 49, and 105.]

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79*, pages 269–282. ACM, 1979. [Cited on pages 29, 49, 64, and 73.]

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992. [Cited on page 49.]

[CCF+06] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the Astrée static analyzer. In *Advances in Computer Science - ASIAN 2006*, volume 4435 of *LNCS*, pages 272–300. Springer, 2006. [Cited on pages 49, 55, 65, and 73.]

[CCF+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009. [Cited on pages 107, 122, 127, 146, and 181.]

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL '11*, pages 105–118. ACM, 2011. [Cited on page 216.]

[CCM11] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *Foundations of Software Science and Computation Structures (FOSSACS)*, volume 6604 of *LNCS*, pages 456–472. Springer, 2011. [Cited on page 73.]

[Cer] Certikos: Certified kit operating system. `http://flint.cs.yale.edu/certikos/`. [Cited on pages 3 and 25.]

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78*, pages 84–96. ACM, 1978. [Cited on page 50.]

[Chl13] Adam Chlipala. *Certified programming with dependent types: A pragmatic introduction to the Coq proof assistant*. MIT Press, 2013. [Cited on page 18.]

[CJPR04] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. In *Programming Languages and Systems*, volume 2986 of *LNCS*, pages 385–400. Springer, 2004. [Cited on page 44.]

[CJPS05] David Cachera, Thomas Jensen, David Pichardie, and Gerardo Schneider. Certified memory usage analysis. In *FM 2005: Formal Methods*, volume 3582 of *LNCS*, pages 91–106. Springer, 2005. [Cited on page 44.]

[CKCY13] Sungkeun Cho, Jeehoon Kang, Joonwon Choi, and Kwangkeun Yi. Sparrowberry: A verified validator for an industrial-strength static analyzer. `http://ropas.snu.ac.kr/sparrowberry/`, 2013. [Cited on pages 2 and 45.]

[CLCVH94] Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. Combinations of abstract domains for logic programming. In *POPL '94*, pages 227–239. ACM, 1994. [Cited on page 73.]

[CN08] Boutheina Chetali and Quang-Huy Nguyen. Industrial use of formal methods for a high-level security evaluation. In *FM 2008: Formal Methods*, volume 5014 of *LNCS*, pages 198–213. Springer, 2008. [Cited on page 26.]

[Com] The CompCert C verified compiler. `http://compcert.inria.fr/`. [Cited on page 24.]

[Coqa] The Coq proof assistant. `https://coq.inria.fr/`. [Cited on pages 3, 16, and 18.]

[Coqb] The Coq reference manual. `https://coq.inria.fr/distrib/current/refman/`. [Cited on page 18.]

[Cou02] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1):47–103, 2002. [Cited on page 36.]

[CP10] David Cachera and David Pichardie. A certified denotational abstract interpreter. In *Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 9–24. Springer, 2010. [Cited on page 44.]

[CRK14] Aziem Chawdhary, Ed Robbins, and Andy King. Simple and efficient algorithms for octagons. In *Programming Languages and Systems (APLAS)*, volume 8858 of *LNCS*, pages 296–313. Springer, 2014. [Cited on pages 146 and 151.]

[CVC] The CVC4 SMT solver. `http://cvc4.cs.nyu.edu/web/`. [Cited on page 16.]

[CZC+15] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *SOSP '15*, pages 18–37. ACM, 2015. [Cited on pages 3 and 26.]

[DGP+09] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of Fluctuat on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems (FMICS)*, volume 5825 of *LNCS*, pages 53–69. Springer, 2009. [Cited on page 43.]

[DVH15] David Darais and David Van Horn. Constructive galois connections. `http://arxiv.org/abs/1511.06965`, 2015. [Cited on page 45.]

[F*] The F* language. `https://www.fstar-lang.org/`. [Cited on page 18.]

[FB14] Alexis Fouilhé and Sylvain Boulmé. A certifying frontend for (sub)polyhedral abstract domains. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, volume 8471 of *LNCS*, pages 200–215. Springer, 2014. [Cited on pages 3, 57, 146, 213, and 214.]

[FC06] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *Proceedings of the 2006 workshop on ML*, pages 12–19. ACM, 2006. [Cited on pages 198, 202, and 204.]

[Fer04] Jérôme Feret. Static analysis of digital filters. In *Programming Languages and Systems (ESOP)*, volume 2986 of *LNCS*, pages 33–48. Springer, 2004. [Cited on page 215.]

[Fer05] Jérôme Feret. The arithmetic-geometric progression abstract domain. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *LNCS*, pages 42–58. Springer, 2005. [Cited on page 215.]

[Fig95] Samuel A. Figueroa. When is double rounding innocuous? *ACM SIGNUM Newsletter*, 30(3):21–26, 1995. [Cited on pages 111 and 115.]

[FL10] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software*, volume 6528 of *LNCS*, pages 10–30. Springer, 2010. [Cited on page 43.]

[FMP13] Alexis Fouilhé, David Monniaux, and Michaël Périn. Efficient generation of correctness certificates for the abstract domain of polyhedra. In *Static Analysis (SAS)*, volume 7935 of *LNCS*, pages 345–365. Springer, 2013. [Cited on pages 3, 57, 146, 213, and 214.]

[Fv] The Frama-C tool, Value plug-in. `http://frama-c.com/value.html`. [Cited on page 43.]

[Fwp] The Frama-C tool, WP plug-in. `http://frama-c.com/wp.html`. [Cited on page 16.]

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013. [Cited on page 27.]

[GKR+15] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *POPL '15*, pages 595–608. ACM, 2015. [Cited on page 25.]

[GLGP12] Eric Goubault, Tristan Le Gall, and Sylvie Putot. An accurate join for zonotopes, preserving affine input/output relations. In *Proceedings of the Fourth International Workshop on Numerical and Symbolic Abstract Domains (NSAD)*, volume 287 of *ENTCS*, pages 65–76. Elsevier, 2012. [Cited on pages 50 and 146.]

[Gon08] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008. [Cited on page 27.]

[Gou94a] Jean Goubault. HimML: Standard ML with fast sets and maps. In *Workshop on ML and its Applications*, 1994. [Cited on page 211.]

[Gou94b] Jean Goubault. Implementing functional languages with fast equality, sets and maps: an exercise in hash consing. Technical report, Bull S.A. Research Center, 1994. [Cited on page 211.]

[Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30(3-4):165–190, 1989. [Cited on page 134.]

[GS12] Mathieu Giorgino and Martin Strecker. Correctness of pointer manipulating algorithms illustrated by a verified bdd construction. In *FM 2012: Formal Methods*, volume 7436 of *LNCS*, pages 202–216. Springer, 2012. [Cited on page 210.]

[GT06] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. In *PLDI '06*, pages 376–386. ACM, 2006. [Cited on page 73.]

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. [Cited on page 12.]

[Idr] The Idris language. `https://www.idris-lang.org/`. [Cited on page 18.]

[IEE08] IEEE Computer Society. 754-2008 – IEEE standard for floating-point arithmetic, 2008. [Cited on page 27.]

[Isa] The Isabelle proof assistant. `https://isabelle.in.tum.de/`. [Cited on page 16.]

[ISO99] ISO. Programming languages – C, 1999. ISO Working Group 14. [Cited on pages 12 and 128.]

[JLB+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *POPL '15*, pages 247–259. ACM, 2015. [Cited on pages 4 and 215.]

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *SOSP '09*, pages 207–220. ACM, 2009. [Cited on page 25.]

[KN06] Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(4):619–695, 2006. [Cited on page 24.]

[Knu11] Donald E. Knuth. *The Art of Computer Programming*, volume 4A, chapter 7.1.4. Addison-Wesley, 2011. Binary decision diagrams. [Cited on pages 186 and 187.]

[Kre15] Robbert Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, Dec. 2015. [Cited on pages 12 and 106.]

[Lap15] Vincent Laporte. *Vérification d'analyses statiques pour langages de bas niveau*. PhD thesis, Université de Rennes 1, November 2015. [Cited on pages 3, 4, 57, and 214.]

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06*, pages 42–54. ACM, 2006. [Cited on pages 12 and 24.]

[Ler09a] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. [Cited on pages 3 and 24.]

[Ler09b] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009. [Cited on pages 24 and 78.]

[Ler12] Xavier Leroy. Proving a compiler: Mechanized verification of program transformations and static analyses. Oregon Programming Language Summer School, 2012. `http://gallium.inria.fr/~xleroy/courses/Eugene-2012/`. [Cited on page 44.]

[LF08a] Francesco Logozzo and Manuel Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *Compiler Construction (CC)*, volume 4959 of *LNCS*, pages 197–212. Springer, 2008. [Cited on page 137.]

[LF08b] Francesco Logozzo and Manuel Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *SAC '08*, pages 184–188. ACM, 2008. [Cited on pages 146 and 215.]

[LL09] Vincent Laviron and Francesco Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5403 of *LNCS*, pages 229–244. Springer, 2009. [Cited on pages 146 and 215.]

[LT93] Nancy G Leveson and Clark S Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993. [Cited on page 1.]

[Mel12] Guillaume Melquiond. Floating-point arithmetic in the Coq system. *Information and Computation*, 216:14–23, 2012. [Cited on page 188.]

[Min04] Antoine Miné. *Weakly relational numerical abstract domains*. PhD thesis, École Polytechnique, Dec. 2004. [Cited on pages 55, 127, 137, 146, 148, 150, 151, 154, 155, 160, 164, 169, 170, 172, 175, and 177.]

[Min06a] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006. [Cited on pages 146 and 151.]

[Min06b] Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *LNCS*, pages 348–363. Springer, 2006. [Cited on pages 137 and 138.]

[miT] miTLS: A verified reference implementation of TLS. `https://mitls.org/`. [Cited on page 26.]

[Mon98] David Monniaux. Réalisation mécanisée d'interpréteurs abstraits, 1998. In french. [Cited on page 43.]

[MP14] Alexandre Maréchal and Michaël Périn. Three linearization techniques for multivariate polynomials in static analysis using convex polyhedra. Technical Report TR-2014-7, Verimag, 2014. [Cited on page 148.]

[Nip12] Tobias Nipkow. Abstract interpretation of annotated commands. In *Interactive Theorem Proving (ITP)*, volume 7406 of *LNCS*, pages 116–132. Springer, 2012. [Cited on page 44.]

[NK14] Tobias Nipkow and Gerwin Klein. Chapter 13: Abstract interpretation. In *Concrete Semantics with Isabelle/HOL*, pages 219–280. Springer, 2014. [Cited on page 44.]

[NO79] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979. [Cited on page 73.]

[NSSS12] Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In *Programming Languages and Systems*, volume 7705 of *LNCS*, pages 115–130. Springer, 2012. [Cited on page 111.]

[O'H04] Peter W O'Hearn. Resources, concurrency and local reasoning. In *CONCUR 2004 – Concurrency Theory*, volume 3170 of *LNCS*, pages 49–67. Springer, 2004. [Cited on page 106.]

[OS05] Veronika Ortner and Norbert Schirmer. Verification of BDD normalization. In *Theorem Proving in Higher Order Logics (TPHOL)*, volume 3603 of *LNCS*, pages 261–277. Springer, 2005. [Cited on page 211.]

[PCG+15] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. `http://www.cis.upenn.edu/~bcpierce/sf`, 2015. [Cited on page 18.]

[PG06] Sylvie Putot and Eric Goubault. Static analysis of numerical algorithms. In *Static Analysis (SAS)*, volume 4134 of *LNCS*, pages 18–34. Springer, 2006. [Cited on page 146.]

[Pic05] David Pichardie. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*. PhD thesis, Université Rennes 1, 2005. In french. [Cited on pages 2, 44, 49, 54, 121, and 127.]

[Pic08] David Pichardie. Building certified static analysers by modular construction of well-founded lattices. In *Proceedings of the First International Conference on Foundations of Informatics, Computing and Software (FICS)*, volume 212 of *ENTCS*, pages 225–239. Elsevier, 2008. [Cited on page 44.]

[Rey02] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002. [Cited on page 14.]

[RM07] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5), 2007. [Cited on pages 106 and 215.]

[Rou14] Pierre Roux. Innocuous double rounding of basic arithmetic operations. *Journal of Formalized Reasoning (JFR)*, 7(1):131–142, 2014. [Cited on pages 111 and 115.]

[SBCA15] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W Appel. Compositional CompCert. In *POPL '15*, pages 275–287. ACM, 2015. [Cited on page 24.]

[seL] The seL4 microkernel. `https://sel4.systems/`. [Cited on pages 3 and 25.]

[SK10] Axel Simon and Andy King. The two variable per inequality abstract domain. *Higher-Order and Symbolic Computation*, 23(1):87–143, 2010. [Cited on page 146.]

[Soz07] Matthieu Sozeau. Subset coercions in Coq. In *Types for Proofs and Programs*, volume 4502 of *LNCS*, pages 237–252. Springer, 2007. [Cited on page 183.]

[SPV15] Gagandeep Singh, Markus Püschel, and Martin Vechev. Making numerical program analysis fast. In *PLDI 2015*, pages 303–313. ACM, 2015. [Cited on pages 146 and 151.]

[ŠVZN+13] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *Journal of the ACM*, 60(3):22, 2013. [Cited on page 24.]

[Vaf13] Viktor Vafeiadis. Adjustable references. In *Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 328–337. Springer, 2013. [Cited on pages 190 and 210.]

[VGL00] Kumar Neeraj Verma and Jean Goubault-Larrecq. Reflecting BDDs in Coq. Rapport de recherche RR-3859, INRIA, 2000. [Cited on pages 191 and 211.]

[VGLPAK00] Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *Advances in Computing Science (ASIAN)*, volume 1961 of *LNCS*, pages 162–181. Springer, 2000. [Cited on pages 191, 208, and 211.]

[VOC] VOCaL project. `https://vocal.lri.fr/`. [Cited on page 216.]

[VST] The verifiable software toolchain. `http://vst.cs.princeton.edu/`. [Cited on page 24.]

[Why] The Why3 tool. `http://why3.lri.fr/`. [Cited on page 16.]

[YH11] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. *Communications of the ACM*, 54(12):123–131, 2011. [Cited on page 25.]

[Z3] The Z3 SMT solver. `https://github.com/Z3Prover/z3`. [Cited on page 16.]

Résumé

Afin de développer des logiciels plus sûrs pour des applications critiques, certains analyseurs statiques tentent d'établir, avec une certitude mathématique, l'absence de certains types de bugs dans un programme donné. Une limite possible à cette approche est l'éventualité d'un bug affectant la correction de l'analyseur lui-même, éliminant ainsi les garanties qu'il est censé apporter.

Dans cette thèse, nous proposons d'établir des garanties formelles sur l'analyseur lui-même : nous présentons la conception, l'implantation et la preuve de sûreté en Coq de Verasco, un analyseur statique formellement vérifié utilisant l'interprétation abstraite pour le langage ISO C99 avec l'arithmétique flottante IEEE754 (à l'exception de la récursion et de l'allocation dynamique de mémoire). Verasco a pour but d'établir l'absence d'erreur à l'exécution des programmes donnés. Il est conçu selon une architecture modulaire et extensible contenant plusieurs domaines abstraits et des interfaces bien spécifiées. Nous détaillons le fonctionnement de l'itérateur abstrait de Verasco, son traitement des entiers bornés de la machine, son domaine abstrait d'intervalles, son domaine abstrait symbolique et son domaine abstrait d'octogones. Verasco a donné lieu au développement de nouvelles techniques pour implémenter des structures de données avec partage dans Coq.

Abstract

In order to develop safer software for critical applications, some static analyzers aim at establishing, with mathematical certitude, the absence of some classes of bug in the input program. A possible limit to this approach is the possibility of a soundness bug in the static analyzer itself, which would nullify the guarantees it is supposed to deliver.

In this thesis, we propose to establish formal guarantees on the static analyzer itself: we present the design, implementation and proof of soundness using Coq of Verasco, a formally verified static analyzer based on abstract interpretation handling most of the ISO C99 language, including IEEE754 floating-point arithmetic (except recursion and dynamic memory allocation). Verasco aims at establishing the absence of erroneous behavior of the given programs. It enjoys a modular extendable architecture with several abstract domains and well-specified interfaces. We present the abstract iterator of Verasco, its handling of bounded machine arithmetic, its interval abstract domain, its symbolic abstract domain and its abstract domain of octagons. Verasco led to the development of new techniques for implementing data structure with sharing in Coq.