THÈSE DE DOCTORAT

présentée à

SORBONNE UNIVERSITÉ

Static analysis of program portability by abstract interpretation

Analyse statique de la portabilité des programmes par interprétation abstraite

---

David DELMAS

28 novembre 2022

Directeur de thèse :	Antoine Miné *Sorbonne Université & CNRS, France*
Rapporteurs :	Eran Yahav *Technion Israel Institute of Technology, Israel* Bor-Yuh Evan Chang *University Colorado Boulder & AWS, États-Unis*
Examinateurs :	Emmanuel Chailloux *Sorbonne Université & CNRS, France* Sylvie Putot *École Polytechnique, France* Jérôme Feret *École normale supérieure, France* Vincent Soumier *Airbus Opérations SAS, France*
Invité :	Abdelraouf Ouadjaout *Indépendant, France*

Laboratoire d'informatique

École doctorale informatique, télécommunications et électronique (Paris)

Résumé

Les logiciels tendent à être utilisés plus longtemps que prévu à leur conception, et dans une plus grande variété d'environnements. Si aucune précaution n'est prise, l'adaptation d'un logiciel à de nouvelles utilisations peut s'avérer très difficile et coûteuse. Assurer la portabilité des programmes est un enjeu majeur : il s'agit de s'assurer que leur compilation et leur exécution dans un environnement différent aura un effet réduit et maîtrisé sur leur sémantique.

Cette thèse vise au développement d'une nouvelle classe d'analyses statiques par interprétation abstraite, permettant de vérifier de telles propriétés de portabilité. Ce but est atteint en trois étapes.

Dans une première étape, nous nous intéressons au problème connexe des patchs logiciels. L'analyse de patchs vise à évaluer l'impact de petites modifications d'un programme sur son comportement. Elle compare les sémantiques de deux versions syntaxiquement proches d'un même logiciel qui s'exécutent dans un même environnement. La principale application est la vérification de non régression. Nous partons d'une sémantique concrète collectrice capable d'exprimer les comportements de deux versions d'un programme simultanément. Cette sémantique est définie par induction sur la syntaxe d'un double programme, une structure syntaxique qui distingue les parties communes des parties spécifiques de chaque version. Supposant d'abord cette structure donnée, nous étudions des programmes numériques prenant leurs entrées dans des flux. Nous proposons une abstraction des flux permettant de prouver que deux versions d'un programme lisant un même flux produisent les mêmes sorties. Puis nous exploitons des domaines numériques classiques pour construire une analyse statique efficace. Nous introduisons un domaine numérique à coût linéaire pour borner les différences entre les variables des deux programmes. Nous proposons enfin un algorithme pour la synthèse d'un double programme à partir de deux versions. Il produit dans la plupart des cas pratiques des doubles programmes suffisants pour des analyses statiques conclusives avec des domaines numériques linéaires.

Dans une seconde étape, nous abordons l'analyse de patchs de programmes C de bas niveau, *i.e.* dont le comportement dépend de la représentation de la mémoire. C'est le cas de la plupart des programmes C, notamment les logiciels embarqués. Nous implémentons l'analyse de patchs sur la plateforme Mopsa, qui permet des analyses relationnelles basées sur des domaines faiblement couplés, et facilite le passage d'un langage jouet à un véritable langage de programmation. Notre implantation tire parti du modèle mémoire implanté dans Mopsa, qui permet l'analyse sûre et précise de programmes C de bas niveau. L'analyse de patchs dans ce modèle permet d'inférer avec succès une première propriété de portabilité des programmes C : la robustesse aux variations des positions des champs scalaires en mémoire, dues *e.g.* à un changement d'interface binaire (ABI), d'options de compilation ou d'extensions C comme des attributs de types ou de variables. L'analyse de patchs de programmes C réalistes dans ce modèle requiert une abstraction numérique expressive pour inférer des égalités entre des déréférencements scalaires dans les deux versions de la mémoire. Nous nous attendons en effet à ce que les deux versions d'un même déréférencement aient presque toujours la même valeur pendant l'exécution, avec seulement des divergences localisées. Nous optimisons donc le modèle mémoire pour ce cas fréquent, en représentant ces égalités symboliquement. Cela permet l'utilisation de domaines numériques non relationnels, d'où un meilleur passage à l'échelle.

Dans une troisième étape, nous considérons une nouvelle propriété de portabilité, motivée par des applications industrielles réelles chez Airbus : la portabilité entre deux plateformes où les représentations en mémoire des scalaires sont différentes (ordres des octets opposés). Nous nous appuyons sur les résultats des deux premières étapes pour construire une analyse statique de cette propriété. Nous étendons notre domaine mémoire pour inférer et représenter symboliquement des égalités modulo le retournement des octets. Nous introduisons un domaine de prédicats symboliques à coût linéaire inférant des relations numériques établies par des calculs arithmétiques bit à bit. L'analyse obtenue permet d'analyser avec succès un logiciel avionique d'un million de lignes de C.

Abstract

Computer programs tend to be used much longer than expected at design time, and in a wider variety of environments. If no care is taken, adapting a software product for new usage may turn out to be difficult and costly. Ensuring the portability of programs is a major stake: it amounts to ensuring that their compilation and execution in a different environment will have small controlled impact on their semantics.

The goal of this thesis is to develop a novel class of static analyses based on abstract interpretation, allowing the verification of such portability properties. This goal is reached in three steps.

In a first step, we consider a related problem: software patches. Patch analysis aims at evaluating the impact of small modifications of a program on its behavior. It compares the semantics of two syntactically close versions of a program running in the same environment. The main application is regression verification: check that the two versions compute the same outputs when run on the same inputs. To address this problem, we start with a novel concrete collecting semantics, expressing the behaviors of both programs at the same time. It is defined by induction on the syntax of a so-called double program, a joint syntactic representation of two versions of a program which distinguishes between common and distinctive parts. We first assume this representation given, and focus on numerical programs reading from input streams. We propose an abstraction of streams able to prove that programs reading from the same stream compute equal outputs. We leverage classic numeric abstract domains to build an effective static analysis. We also introduce a novel numeric domain with linear cost to bound differences between the values of the variables in the two programs. Finally, we propose a heuristic algorithm for constructing a double program from a pair of program versions that allows, in most practical cases, successful patch analyses relying on linear invariants only.

In a second step, we turn to the analysis of patches of low-level C programs, *i.e.* programs whose behaviors depend on the representation of computer memory. This is the case of most C programs, especially in embedded software. We implement our patch analysis on top of the Mopsa platform, which allows relational analyses based on weakly coupled cooperating abstract domains, and eases the lifting of an abstract domain from a toy language to a real-world language such as C. Our implementation benefits from the memory model implemented in Mopsa, whichs allows sound and precise analyses of low-level C programs. Patch analysis in this model allows us to successfully address a first portability property: robustness to variations of the offsets of scalar fields, such as those introduced by changes in the Application Binary Interface (ABI) of the target, compiler options or language extensions such as attributes of types or variables. Analyzing realistic patches of C programs with this memory model requires expressive numerical abstractions that infer equalities between matching scalar dereferences in the two versions of the memory. Such dereferences are indeed expected to be equal most of the time during program execution, with only local deviations. We thus optimize the memory model for this common case, by representing these equalities symbolically. This enables the use of non-relational numerical domains, improving scalability.

In a third step, we address an additional portability property, motivated by real industrial applications at Airbus: portability across platforms with opposite byte-orders (a.k.a. endiannesses). We leverage the results of the two first steps to derive a static analysis of this property. We extend our memory domain to infer and represent symbolically equalities modulo byte-swapping. We introduce a novel symbolic predicate domain with near-linear cost to infer relations between individual bytes of the variables in the two programs, such as those established by bitwise arithmetic. The resulting analysis allows analyzing successfully a large real-world avionics software product of one million lines of C.

Acknowledgments

Let me first thank Patrick Cousot, Radhia Cousot, and Famantanantsoa Randimbivololona. They initiated long-term cooperations between academic research teams and Airbus on industrial applications of Abstract Interpretation. I was introduced to static analysis in this context, with Randim mentoring me in the transfer of practical applications into operational development processes of avionics software. Patrick, Radhia and Randim strongly encouraged my desire to learn more, also on the academic side.

It was also my privilege to cooperate with Antoine Miné in collaborative research projects. His strong orientation towards practical applications, his kindness, generosity, patience and pedagogical skills (not to mention his deep scientific expertise), made him, in my opinion, the ideal PhD advisor. Thank you Antoine, for welcoming me into the Mopsa research team. Thank you for this beautiful topic, where you suspected some low-hanging fruits might allow short-term industrial applications. Thank you for your patience, your dedication, and your brilliant suggestions, without which none of this could have happened.

Another key contributor to the success of this work is Abdelraouf Ouadjaout. As the main developer of Mopsa's framework, he invested significant effort into supporting me while I was developing prototype static analyses on top of it. Thank you very much Abdelraouf. This work would no doubt have had far more limited results without your patient and generous help.

I am also grateful to the Avionics Software Department of Airbus, which enabled me to perform this work as part of my job as an engineer. I especially thank the Heads of Department: Marc Lemeilleur, who made the initial decision, and Vincent Soumier, who supported this work with great kindness, and ensured that it could be performed in great conditions. These great conditions owe a lot to the $^{\nu}WoW$ team, which took over most of my operational tasks: thank you Abderrahmane Brahmi, Marie-José Carolus Monique Dandurand, Pascal Lacabanne, Éric Lacombe, Victoria Moya Lamiel, and Jean Souyris. Let me also thank the avionics software and simulation teams, in particular Amandeepsingh Bhatia and Julien Lensky, for our fruitful collaboration on the analysis of Module A (see Chapter 7).

Let me finally thank warmly Eran Yahav and Bor-Yuh Evan Chang for accepting to review this manuscript in detail, and Emmanuel Chailloux, Sylvie Putot and Jérôme Feret for accepting to be part of the defense committee. I am honoured by the time and attention that you all put in my work.

Remerciements

De nombreux collègues d'Airbus m'ont soutenu dans cette aventure. Merci encore à Vincent et à l'équipe $^{\nu}WoW$, merci aussi à Michel Angot, Sébastien Cano, Nathalie Crochet, Jérôme Étienne, Habib Essoussi, Olivier Gicquel, Jean-François Menginette, Olivier Raynal, Sylvain Sauvant, Benoît Triquet, Francis Vergara, Marc Vié, Sarah Zennou *et al.*

Merci également à toute l'équipe APR du LIP6, qui m'a accueilli avec beaucoup de chaleur et de générosité. Une pensée émue pour les camarades du bureau 303 : Boubacar, Clément, Francesco, Frédéric, Ghiles, Guillaume, Jules, Martin, Mat(t)hieu, Raphaël, Yi-Ting *et al.* Matthieu et Raphaël, merci de m'avoir prêté vos thèses pour m'aider à rédiger celle-ci. Raphaël, merci de tes conseils et de tes généreux coups de main.

Je voudrais dire aussi ma gratitude aux proches qui ont été présents ces dernières années, et ont su remonter mon moral de la cave au premier étage dans les moments de doute. Certains sont des collègues d'Airbus ou des camarades du LIP6, je les remercie à nouveau. Ma reconnaissance va aussi à tous les autres: Abdou & Joëlle, Alexandrine, Aliénor & Laurent, Brigitte & Joël, Caterina, Émeric & Mélanie, Éric & Alyssa, Gilles & Laura, JB, Luc & Séverine, Marlène & Jean-Luc, Mélanie & Doudou, Sabine, Stéphane, Sunny & Salia, Teddy & Sonia, Yves & Anne. *et al.*

Ce travail n'aurait pas pu aboutir sans le soutien de ma famille, et en particulier celui de ma belle-famille et de ma tante, qui ont bien voulu assumer le soin de nos enfants bien plus souvent qu'à leur tour. Merci Laurence, Hervé, Christian, Danielle, pour votre générosité. Merci Monique, de m'avoir accueilli chez toi à chacune de mes visites au labo, et d'avoir accompagné le *sprint* final de la rédaction de cette thèse. Merci à mes parents, de m'avoir transmis la soif d'apprendre et la détermination d'aller au bout.

Corto et Léna, mes merveilleux enfants, merci pour votre amour, vos encouragements, et vos précieux dessins, textes, colliers et bracelets. Ils m'ont porté chance.

Contents

Chapter 1

Introduction

Industrial societies devote increasingly important roles to software. Besides mediating social interactions, computer programs control the engines and brakes of cars [7] and aircraft, the flights of fly-by-wire airplanes [32] and UAV, emergency systems of nuclear power plants [103], and medical devices such as pacemakers[1]. Such software is termed *safety-critical*, as some failures thereof may have catastrophic consequences, including the loss of human lives. The case occurs typically for industrial software embedded in transportation, military, medical, and power systems. As emphasized by [46, Sec. I.1], failures of such software may additionally cost billions of dollars[2], and ruin the reputation of companies ruled against by courts for defects[3].

1.1 Approaches to reliable software

To avoid such situations, software companies are struggling to apply best practices to produce quality software. Such best practices are typically gathered into international standards, such as IEC 62304 [89] for medical devices, DO-178 [3, 4] for aviation, and IEC 60880 [88] for nuclear power plants.

1.1.1 Best effort process-based assurance

Most of these industrial standards consist in guidelines for process-based assurance, relying on the assumption that a better-defined process is more likely to produce quality software. Such standards emphasize process activities such as software planning, specification, design, unit and integration testing, and peer-reviews. Informal verification methods such as testing and peer-reviews have well-known shortcomings: they may miss bugs. To face this limitation, standards typically mandate that every software artifact

[1] https://public4.pagefreezer.com/content/FDA/16-06-2022T13:39/https://www.fda.gov/medical-devices/safety-communications/cybersecurity-vulnerabilities-affecting-medtronic-implantable-cardiac-devices-programmers-and-home

[2] https://en.wikipedia.org/wiki/Financial_impact_of_the_Boeing_737_MAX_groundings

[3] http://www.nytimes.com/2013/12/14/business/toyota-seeks-settlement-for-lawsuits.html

be verified multiple times, using different techniques, and possibly by different people. The resulting industrial processes are thus intentionally very heavy and redundant. In some domains, such as aviation and railway, software is liable to certification by third parties on behalf of national or international authorities, and applicable standards are used as the basis for certification audits.

1.1.2 Formal methods for product-based assurance

Unlike informal verification methods such as testing and peer-reviews, formal methods strive for exhaustiveness: never miss a bug (in a well-defined class of bugs). Their goal is to provide definite statements about program properties such as correctness, safety and security, so that assurance relies solely on the intrinsic quality of the product, rather than on that of the process or the expertise of the software engineer. These methods are grounded on mathematically well-defined techniques. Formal methods include model-checking [41], symbolic execution [108], program proof [85], typing [38], static analysis [48], and program transformations [152]. They can be unified [55, 52, 56], [46, Chap. 26] in the framework of abstract interpretation [48].

Let us introduce, informally, key notions related to the formal methods of interest in this introduction.

Semantics

Semantics is the art of assigning meanings to programs [75]. The semantics of a program is a mathematical description of the set of its possible executions. It serves as the basis for formal reasoning about its run-time behavior.

Formal methods

Formal methods are techniques relying on mathematical reasoning about the behaviors of programs. A formal method is called *sound* if all its outputs can be proved compatible with a well-defined program semantics, unsound otherwise. A formal method is called *complete* if it allows proving all the properties that are compatible with the semantics. Due to Rice's theorem [158], no formal method can be at the same time sound, complete and automatic. Fig. 1.1 compares testing with two formal methods: program proof and static analysis[4]. Testing is considered complete and unsound, as "testing shows the presence, not the absence of bugs" [34].

Program proof

Program proof, *a.k.a.* deductive program verification, is a formal method that verifies run-time properties of programs at compile-time. It operates on the source code of a program, together with a set of user-specified properties expressed in some program logic. Program proof techniques can be designed to be sound and (relatively) complete,

[4]We thank Matthieu Journault for this figure.

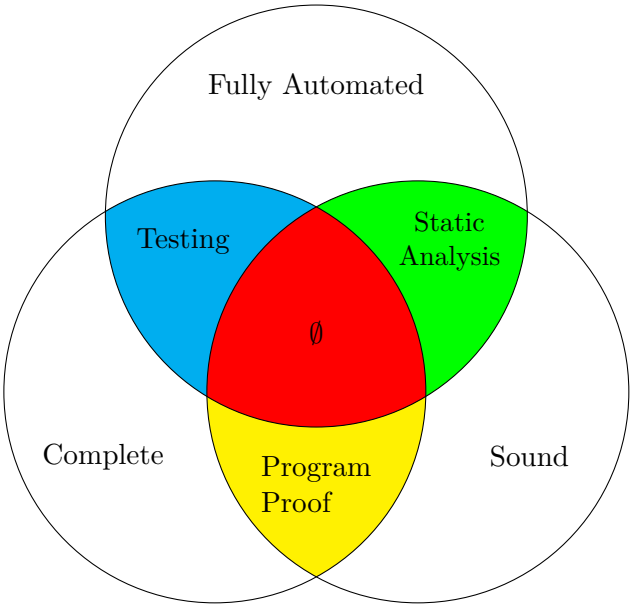


Figure 1.1: Program verification techniques.

but they can only be partially automated. Indeed, these methods require that program parts be annotated with contracts (pre- and post-conditions). In particular, loops must be annotated with inductive invariants. In addition, although theorem provers such as SMT-solvers can be used to automate proofs, these tools require the assistance of the user in some cases, *e.g.* to prove some quantified formulas.

Static analysis

Static analysis is a formal method that infers run-time properties of programs at compile-time. It is typically implemented in tools, called static analyzers, that analyze the source (or compiled) code of a program. Static analyzers can be designed to be sound and automatic, and to always terminate. Static analyzers compute on (conservative) approximations of program semantics designed to infer predefined classes of properties of interest, at the cost of completeness: analyses may be inconclusive.

Abstract interpretation

Abstract interpretation [46] is a unifying theory of the executions of computer programs. It formalizes formal methods, allowing formal reasoning about their properties, such as soundness (all provable facts are true) and completeness (all true facts are provable). Abstract interpretation can be used to design semantics, proof methods, and static analyses of programs. In particular, it allows designing static analyses that are sound by

construction, by formalizing the relationships between a (concrete) program semantics of reference and (abstract) approximations thereof handled by static analyzers.

1.1.3 The case of avionics software at Airbus

A relevant example is the case of avionics software at Airbus. This case is the most familiar to us, as we are writing this thesis after 20 years as an avionics software engineer at Airbus, with practical experience in the verification of certified software with both formal and informal methods [61, 63, 166, 64, 28, 133, 57, 62]. The avionics software department of Airbus is responsible for the development of the software of some of the most safety-critical cockpit avionics systems of Airbus aircraft, such as fly-by-wire control systems, flight-warning, maintenance and communication systems. This department is known for pioneering the industrial deployment of formal methods in the avionics domain [166].

In the aeronautical domain, avionics software running on airborne computers are considered critical components of the systems of civil aircraft. They are thus subject to certification on behalf of authorities such as the European Union Aviation Safety Agency (EASA) and the US Federal Aviation Administration (FAA), and developed according to stringent rules imposed by the applicable DO-178 standard. Three revisions of this standard have been in use since its creation in the beginning of the 1980s, with limited updates to the main document. In the meantime, most aircraft functions have been transferred from hardware to software. This trend, together with the growing effort to optimize fuel consumption and passenger comfort, as well as system configurability and interoperability, has resulted in an exponential growth of the size and complexity of this kind of avionics software, as depicted on Fig. 1.2[5].

Legacy processes

On the other hand, the state of the industrial practice has evolved at a much slower pace during the same time frame. Most avionics software processes are still based on informal specifications and designs, pair reviews, and hand-written test procedures based on equivalence class partitioning.

As shown on Fig. 1.3, all artefacts produced by development processes, *i.e.*, in DO-178 terminology, *High-Level Requirements* (HLR), architecture, *Low-Level Requirements* (LLR), source and executable code, are verified several times. All of them are reviewed for compliance with artefacts from which they are derived on one hand, and for accuracy, consistency, hardware compatibility and conformance to standards on the other hand. In addition, executable code is verified against LLR and HLR by means of unit and integration testing. Test cases and procedures are then also subject to pair reviews.

LLR provide very detailed specifications for every individual C function or assembly routine, down to the specification of every single procedure call or volatile access. These specifications are typically expressed in informal pseudo-code. In this context,

[5]We thank Vincent Soumier for this figure.

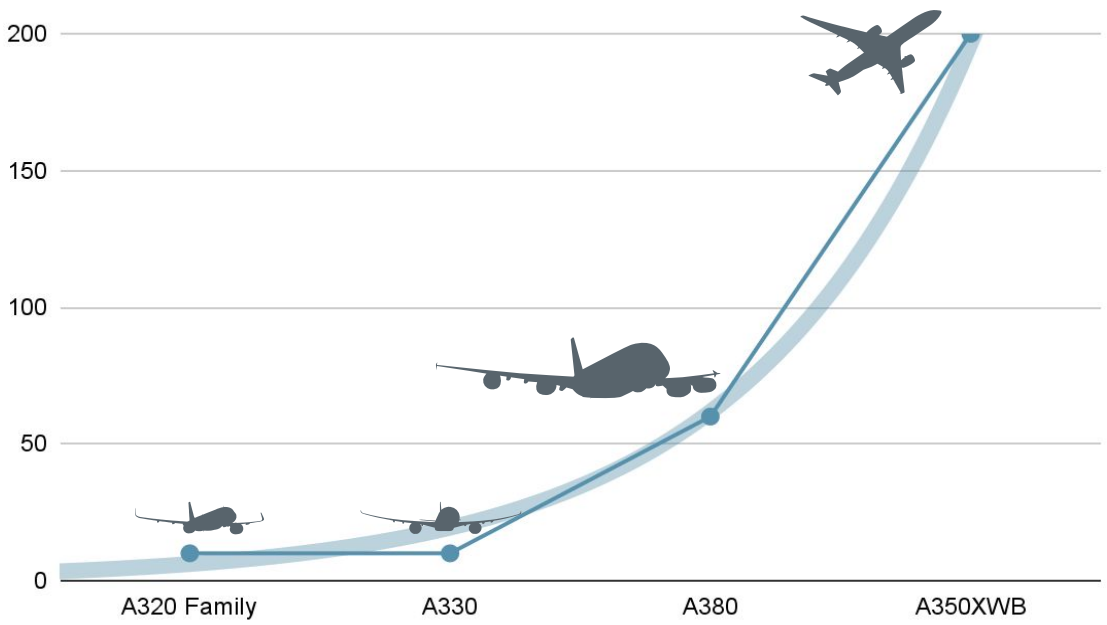


Figure 1.2: Avionics software on board Airbus aircraft (Mloc)

significant effort is invested into unit testing, which roughly amounts to perform grey-box testing of every individual C function or assembly routine on the target hardware, to ensure that it implements the given algorithm correctly. Indeed, test scenarios and expected values of outputs are derived completely by hand, from an intellectual reinterpretation of the informal LLR. Then, the correctness of this reinterpretation and the correct implementation of associate scenarios are verified in additional pair reviews.

Despite the automatic compilation of test scripts into target programs for part of the test procedures, the largest part of these heavy design and verification processes is essentially hand-crafted and relies completely on human expertise. Unfortunately, such legacy processes do not scale up to current avionics software size and complexity within reasonable costs, hence cost issues in both development and maintenance phases. In particular, verification is liable for a steadily growing share of the overall development costs. The 2015 status is about 70% for maximum-criticality software, such as fly-by-wire control/command software developed mostly with legacy methods.

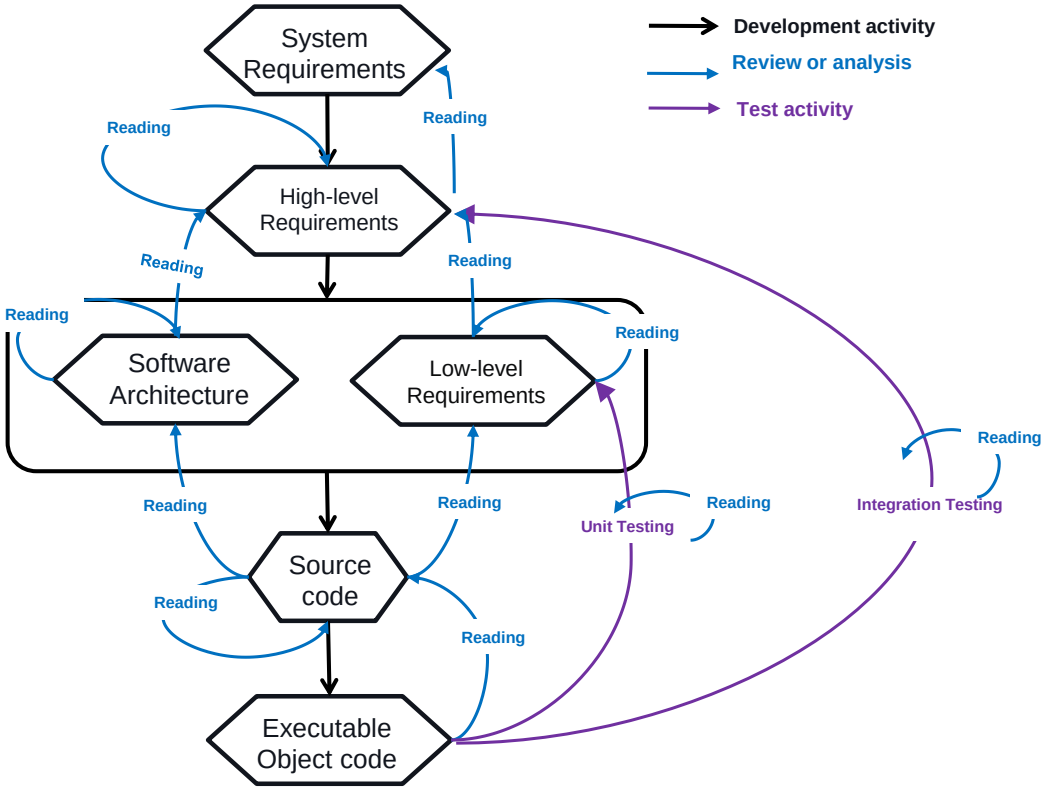


Figure 1.3: Legacy process safety-critical software

Formal methods as cost-cutters

To address this economical risk, Airbus has started introducing formal techniques into the verification processes of several internal avionics software products since 2001 [166], in order to replace or complement legacy methods. Such techniques have indeed the potential to improve automation while preserving safety. Such methods provide indeed strong, mathematical guarantees about systems behaviors.

Several tools based on formal techniques have been shown to cope with real-world industrial software. Some program proof techniques using deductive methods have been shown credible alternatives to unit testing of individual C functions [71]. This approach is cost-efficient when most proofs are automatic. The case occurs when verifying relatively small functions with abstract memory and numerical models [29]. The soundness of this approach can be ensured at source code level by validating the assumptions introduced by the memory and numerical models. Such assumptions include the absence of run-time errors and non-aliasing of function parameters. Transferring this confidence from the source to the compiled program requires trusting the compiler. Fortunately, a formally verified compiler [115] has been shown usable in practice in the avionics context [22]. Moreover, multiple static analyses based on abstract interpretation have been shown to

cope with real-world industrial software. They typically infer non-functional properties, such as the absence of run-time errors [61, 25, 106, 133], reachable data and control flows [57, 105], and safe upper-bounds of the stack consumption [104], of the worst-case execution time [165], and of round-off errors in floating-point computations [64]. Such analyses are sound and automatic alternatives to costly and error-prone code reviews. For instance, ASTRÉE [25] is routinely used at Airbus to prove the absence of run-time errors of fly-by-wire control/command programs [133] up to 800,000 lines of C. In addition, most of them are whole-program analyses that scale up to large software, allowing to validate the assumptions required by unit proofs in all calling contexts.

Transforming industrial processes with formal methods

In a first step, some program proof [71] and static analysis [165, 61, 133, 64, 57] techniques were introduced on some avionics projects, until 2015. However, their impact on industrial efficiency has been limited by a lack of formalization of design processes, by an incompatibility of previous testing frameworks with formal methods and by a lack of interoperation between tools, resulting in significant efforts dedicated to preparing inputs for formal verification processes.

In a second step [28, 29], Airbus avionics processes have been transformed to maximize automation, while relying on formal techniques to maintain the highest standards for safety. Fig. 1.4 gives a simplified overview of the *New Ways of Working* ($^{\nu}WoW$) automated process. We refer the reader to [28, 29] for a complete description. The design phase, presented in the central box, is deeply revisited. In legacy processes, this phase was mostly a preparation for the coding phase, producing informal documentation expressing LLR in the form of pseudo-code, and their traceability to HLR. In the $^{\nu}WoW$ approach, the design phase is extended to prepare verification. Dedicated domain-specific languages and compilers have been created to enable the formalization of all design artifacts. A first advantage is that it allows automating a large part of reviews of design data for accuracy, consistency, or conformance to standards. A second, key interest is that it also allows a tight interoperation between design and verification tools. For instance, LLR are formalized as first-order contracts expressed in an internal behavioral interface description language, coined DCSL [28]. The DCSL compiler translates them to unit proof or test contracts and generates associate verification environments. This enables a hybrid approach to unit verification: individual procedures can be verified either by unit test or unit proof, for the same design contracts. This approach has been shown to be very cost-efficient [28]. Note that Fig. 1.4 shows the case of unit proof only, for conciseness. Also note that the proof engine is used to check design consistency, even when the implementation is tested. The DCSL compiler additionally generates expected variables ranges and data-flows to be verified by static analysis tools. Moreover, the functional ranges guaranteed by static analysis allow focusing the scope of unit verification to reachable input ranges. Some static analyzers compute configuration files for other static analyzers, *e.g.* a points-to analysis on source code resolves computed calls, enabling stack analysis on machine code. Static analysis tools validate

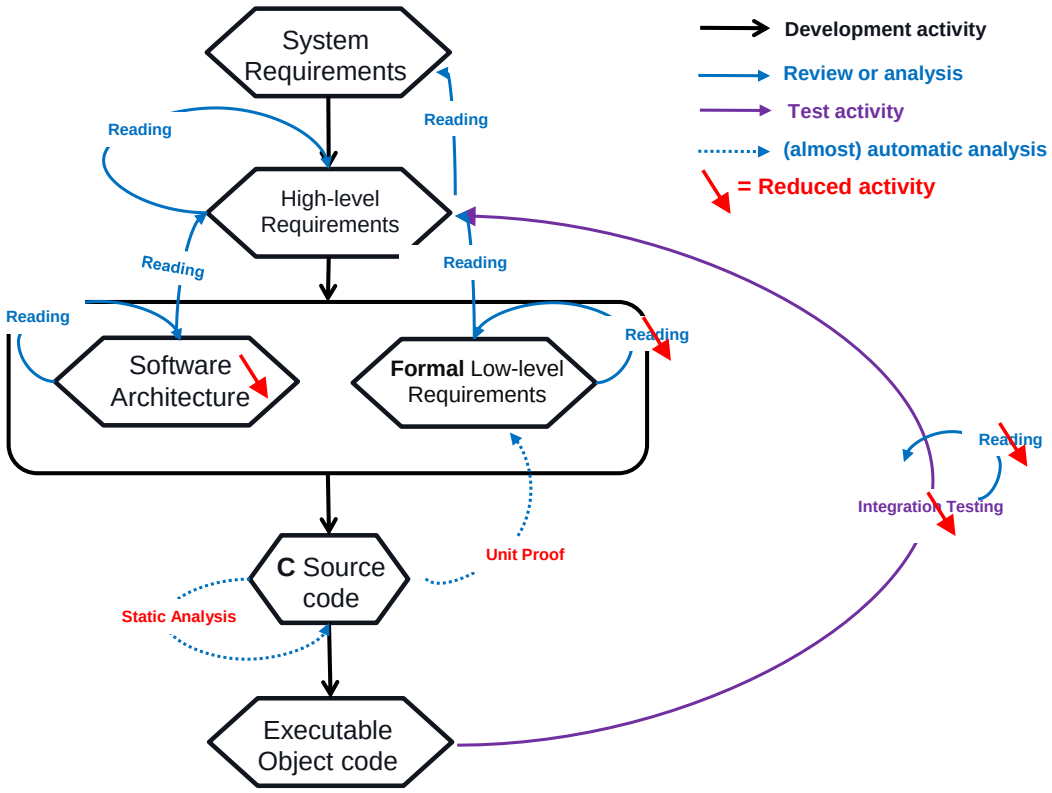


Figure 1.4: $^{\nu}WoW$ process with unit proof

assumptions used by design and unit verification tools, such as the absence of run-time errors, variable ranges and non-aliasing.

Moreover, reviews for conformance of C source code to LLR are eliminated where unit proof is applicable, thanks to the exhaustiveness of this verification technique. Reviews for conformance with the software architecture are automated using a mixture of syntax-based and semantics-based static analysis techniques, such as data flow analysis [57]. Reviews for conformance to standards are automated by syntax-based static analysis tools [62]. Reviews for accuracy and consistency are automated by semantics-based static analysers by abstract interpretation [133, 64, 104, 165]. Note that the traceability analysis between source and compiled code, required for the most critical software products, is also avoided thanks to a formally verified optimizing C compiler [22]. As a consequence, code and design reviews are mostly limited to tool warning analyses.

The $^{\nu}WoW$ process has been deployed on three large avionics projects, with an estimated productivity gain of 50% on the subset of the activities related to design and unit verification [28]. As a side effect, integration testing is also alleviated, as some properties that could only be verified in integration tests with legacy processes, *e.g.* volatile accesses, are verified at unit level, by unit proof or unit test on virtualized hardware.

Updating standards

Standards have evolved to welcome such alternative techniques into industrial processes. For instance, the latest revision of DO-178 [4] introduces a technical supplement, DO-333 [6], providing guidance on the use of formal techniques to meet certification objectives. This supplement introduces standard categories of formal analysis techniques: deductive methods, model-checking, and abstract interpretation. Abstract interpretation, in particular, is presented as a method for constructing semantics-based analysis algorithms for the automatic, static, and sound determination of dynamic properties of infinite-state programs. It emphasizes soundness as the key criterion for an analysis to be considered compliant: the applicant is required to provide justifications that the method never asserts that a property is true when it may not be true.

1.1.4 Combining testing and formal verification

Nonetheless, the current state of the industrial practice is that completely formal development methods are still not quite tractable for most avionics software products. Current approaches rely on hybrid processes [28, 29, 44, 142], that integrate design activities with static analysis, program proof, and testing.

Testing is still needed

Indeed, while some whole-program static analyses [104, 165, 133, 105] based on abstract interpretation successfully scale to large real-world industrial software, program proofs with deductive methods are currently not yet considered applicable, in an industrial context such as Airbus avionics, to the complete functional verification of large software. The reason for this situation is that proving large software requires users to annotate program parts with inductive invariants, which require specific expertise and may be as large as the program itself. In addition, as explained in Sec. 1.1.3, program proof is only considered an efficient alternative to testing when most of the proofs are automatic. The case occurs when proving relatively small programs with abstract memory and numerical models, which is not applicable to all parts of an industrial software that contains low-level code. As a consequence, deductive program proofs are only used, for now, as a replacement of unit testing at the level of small, well-typed individual C functions, in order to verify, locally, the correct implementation of algorithms.

As a consequence, testing is still required for the functional verification of large industrial software. Large whole-program test scripts are thus developed in parallel to the initial version of an industrial software products. Their primary objective is to detect and fix bugs, and gain confidence in the behavior of the product. They serve as the standard means to verify the correct implementation of the HLR in the context of certified avionics software.

Regression verification

Then, new versions of the product are developed, and most test procedures are reused, in hope that they might detect any "regressions", *i.e.* any unintended changes in program behavior. The maintenance of such test procedures is very costly. First, they are typically affected by any change in software interfaces. Second, they have been developed with the behaviors of the initial version of the product in mind. As a consequence, intended changes in some program behaviors typically break large subsets of the test cases, including many test cases that intended to target unrelated program behaviors. Discriminating such situations from actual regressions is a difficult and error-prone task.

In the end, the maintenance of large test bases over time is a costly and ineffective approach to regression verification. This situation is not specific to safety-critical software. It generalizes to most software development projects. Moreover, it is made worse by the trend, in industrial software, to develop software in product lines [117, 90]. This approach aims at developing generic software components that are shared by multiple products. While this approach eases the initial development of a new product from available components, it may complicate its long-term maintenance. Indeed, changes to generic components affect the behaviors of multiple products. Therefore, there is a strong need to ensure freedom from potential regressions when incorporating a new revision of a generic component into a product. Note that a similar situation occurs when incorporating commercial off-the-shelf (COTS) or open source software that evolves over time[6].

Example 1 (A patch from the GNU core utilities)**.** Consider, as a motivating example, the commit shown on Fig. 1.5, extracted from a revision control repository of the GNU core utilities. It describes a change in a library implementing core functions for removing files and directories, and used by the POSIX `rm` command. The main function of this library uses the POSIX `fstatat` function to read information on the file to delete. As the same status information is needed in several contexts, the library implements a caching mechanism. At initialization, the main function calls a `cache_stat_init` function, which initializes the `st_size` field of the stat structure `*st` to `-1`. Then, it calls the `cache_fstatat` function shown on Fig. 1.5 repeatedly, whenever status information is needed. Indeed, `cache_fstatat` caches the results of the `fstatat` function. In revision v6.10 of Coreutils, this function used the `st_size` field of the stat structure `*st` to store information on the error value returned by `fstatat` upon the first call. It did it in a way that ensures that `st_size<0` whenever `errno>0`, so as to use the sign of `st_size` upon subsequent calls to distinguish between successful and erroneous executions. This scheme works for operating systems where `errno` is always set to positive values. However, some systems, such as BeOS [1] and Haiku [2], allow for negative `errno` values. The fix displayed on Fig. 1.5 aims at accommodating such systems. It consists in storing `errno` directly in the `st_ino` field of the stat structure.

[6] `https://www.bleepingcomputer.com/news/security/big-sabotage-famous-npm-package-deletes-files-to-protest-ukraine-war/`
`https://www.theregister.com/2021/12/14/log4j_vulnerability_open_source_funding/`

```
172 172    /* Like fstatat, but cache the result.  If ST->st_size is -1, the
173 173       status has not been gotten yet.  If less than -1, fstatat failed
174     -     with errno == -1 - ST->st_size.  Otherwise, the status has already
    174 +     with errno == ST->st_ino.  Otherwise, the status has already
175 175       been gotten, so return 0.  */
176 176    static int
177 177    cache_fstatat (int fd, char const *file, struct stat *st, int flag)
178 178    {
179 179      if (st->st_size == -1 && fstatat (fd, file, st, flag) != 0)
180     -      st->st_size = -1 - errno;
    180 +      {
    181 +        st->st_size = -2;
    182 +        st->st_ino = errno;
    183 +      }
181 184      if (0 <= st->st_size)
182 185        return 0;
183     -    errno = -1 - st->st_size;
    186 +    errno = (int) st->st_ino;
184 187      return -1;
185 188    }
```

Figure 1.5: Patch on `remove.c` of Coreutils (between v6.10 and v6.11)

On this example, regression verification amounts to proving that the behavior of the main function of the library is unchanged on systems with only positive error values. We will come back to this example in Sec. 3.1.

First problem statement: patch analysis

More generally, regression verification [79] aims at proving the equivalence of two syntactically close versions of a program [92] running in the same environment. It is a particular instance of the problem of proving the functional equivalence of programs, or program parts, which is fundamental [76]. In this thesis, we develop an approach based on abstract interpretation to address this sub-problem, in particular in the context of low-level C programs such as those used in embedded software. We term our approach *patch analysis*.

For instance, our analysis is able to prove the absence of regression in the case of Example 1.

Portability

Portability is a related problem. Computer programs tend to be used much longer than expected at design time, and in a wider variety of environments. If no care is taken, adapting a software product for new usage may turn out to be difficult and costly. Ensuring the portability of programs is a major stake: it amounts to ensuring that their compilation and execution in a different environment will have small controlled impact on their semantics, and that the safety of executions is not jeopardized.

In practice, developers maintain multiple, syntactically close versions of the same software products to accommodate multiple environments. Typical examples are the Linux kernel and drivers, that run on a vast variety of machines. Such software products are typically composed of generic parts, written in portable C, and target-specific parts activated or de-activated by conditional compilation. The case occurs also in industrial embedded software developed in product line approaches, where generic components should be portable to the environments of the software products that incorporate them.

A third, less well-known occurrence of portability is the simulation of embedded systems. For instance, Airbus develops simulators for crew training and system validation. These simulators are based on commodity hardware, as embedded hardware may be a rather scarce and expensive resource. In some cases, there is a strong need that simulations be "representative" of the behavior of the real embedded system, *i.e.* that they feature the same observable behaviors. The case occurs especially in the context of system validation, when applicants would like to replace part of the tests of the real software on the real hardware with simulations, and use the results of these simulations for certification credit. The related supplement [5] to the applicable standard [4] mandates, in this case, that "*an analysis should provide compelling evidence that the simulation approach provides equivalent defect detection and removal as testing of the Executable Object Code*". In this case, the simulator is generated from the C source code of the real system. Yet, this is not enough to guarantee that it will behave as the real system. The C standard [91] leaves indeed the encoding of scalar types and the layout of fields in structures partly unspecified. The precise representation of types is standardized in implementation-specific *Application Binary Interfaces* (ABI), such as the System V ABI [11], to ensure the interoperability of compiled programs, libraries, and operating systems. Although it is possible to write fully portable, ABI-neutral C code, the vast majority of C programs rely on assumptions on the ABI of the platform. This is especially the case in embedded programs, that often require a low-level access to the system. Such programs tend to rely on low-level programming constructs that abuse unions and pointers to bypass the type system of the language. As a consequence, differences between the ABI of the simulation platform and that of the embedded platform may result in the embedded system and the simulator behaving differently.

Endianness is an important source of portability errors in this context. Simulators are typically run on little-endian x86 machines, while some embedded processors such as PowerPC are big-endian.

Example 2 (Endianness)**.** For instance, Fig. 1.6 shows a snippet of code for reading network input. Recall that the order of bytes in the representation of network data is standardized, platform-independent: it follows the big-endian byte-order. Assume variables `x` and `y` have the same multi-byte integer type. The sequence of bytes read from the network is first stored into `x`. `x` is then either copied, or byte-swapped into `y`, depending on whether the endianness of the platform matches the (big-endian) network byte-order, or is the opposite endianness.

On this example, portability verification amounts to proving that the two endian-specific variants of the snippet compute the same value for `y`, whatever the values of the

```
1   read_from_network((uint8_t *)&x, sizeof(x));
2 # if __BYTE_ORDER == __LITTLE_ENDIAN
3     uint8_t *px = (uint8_t *)&x, *py = (uint8_t *)&y;
4     for (int i=0; i<sizeof(x); i++) py[i] = px[sizeof(x)-i-1];
5 # else
6     y = x;
7 # endif
```

Figure 1.6: Reading input in network byte-order (Example 2)

```
struct { char c; int x; } s = {0};
char *p = (char *) &s;
p[4] = 1;
```

(a) Type puning

	System V alignments	Packed structs
Little-endian	1	2^{24}
Big-endian	2^{24}	1

(b) Final value of `s.x`

Figure 1.7: Offsets of fields and endianness (Example 3)

bytes read from the network. We will study this example in detail in Sec. 7.1.

Additional sources of portability errors include the offsets of fields in data structures, which are typically ABI-dependent. Moreover, embedded systems often use C extensions such as type attributes or `#pragma` directives to fine-tune the memory layouts of some data structures, *e.g.* to memory-map hardware resources. Such extensions may be interpreted differently by different compilers.

Example 3 (Offsets of fields and endianness)**.** For instance, Fig. 1.7(a) shows a snippet that abuses pointers to bypass the C type system, a common practice in low-level programming known as *type punning*. The final value of `s.x` depends on the offset of field `x` in struct `s` and on the size and representation of type `int`. Let us assume a 32-bit architecture: `sizeof(int)=4`. If this snippet is compiled for a platform implementing the System V ABI, then 3 padding bytes are introduced between fields `c` and `x`, and value 1 is written to the lowest-address byte of `s.x`. This is the default behavior of GCC when compiling for a host Linux x86 machine. In contrast, value 1 is written to the highest-address byte of `s.x` if the structure `s` is packed. Such packing can be obtained, for instance, using GCC's `-fpack-struct` option or `__attribute__((packed))` attribute. In either case, the final value of `s.x` depends on the weight of individual bytes in scalar values (*a.k.a.* endianness of the platform). The final values of `s.x` for the 4 possible ABI are shown on Fig. 1.7(b).

On this example, portability verification amounts to proving that this snippet computes the same value for `s.x` on the platforms which have both opposite endianness and different alignments (packed versus System V).

We will address the two previous sources of portability errors in this thesis: endianness and offsets of fields and data structures. Nonetheless, multiple other sources sources of portability issues exist, which we will not address. For instance, ensuring the portability of software across platforms where scalar types have different sizes is a difficult task which Linux developers face daily. As another example, ensuring the portability of performance-critical parallel programs across platforms with different weak memory consistency models is very challenging [9, 114].

Second problem statement: portability analysis

More generally, portability verification aims at proving the equivalence of two syntactically close versions of a program running in different environments. As program equivalence, it is a fundamental problem [85]. Portability and non regression are related properties. Indeed, they both deal with comparing the semantics of two program versions. Regression verification compares the semantics of two programs running in the same environment. They may have different semantics, as they have different syntax. Portability verification compares the semantics of two programs running in different environments. They may have different semantics, even if they share exactly the same syntax. They may additionally exhibit syntactic differences. It is thus natural to view regression verification as a particular case of portability verification, where program versions run in the same environments.

In this thesis, we develop an approach to portability analysis of C programs that is designed as an extension of patch analysis. We focus on two portability properties, related to the representation of the computer memory specified by the ABI: portability against changes in the offsets of scalar fields of C structs, and portability against the order of bytes in the representation of scalars on the platform, which we term *endian portability*.

For instance, our analysis is able to prove the portability of Examples 2 and 3.

1.2 Overview of the thesis

In this thesis, we wish to contribute to the development of formal methods applicable to the verification of real-world software, such as low-level C avionics software. We target the two problems introduced in Sec. 1.1.4 : patch and portability analysis. We address these problems by proposing a novel class of sound, semantics-based static analyses by abstract interpretation.

1.2.1 Outline

We start by introducing static analysis by abstract interpretation in Chapter 2, in the context of a simple numerical imperative language called Nimp. We rely on a standard approach used throughout the thesis: we define a denotational concrete collecting semantics by induction on the syntax, and construct an analysis that is parametric in the choice of a numerical abstract domain.

Then, we address patch analysis of numerical programs in Chapter 3. We define a language called NIMP$_2$ for so-called double programs. A double program is a joint syntactic representation of two versions of a NIMP program, which distinguishes between common and distinctive parts. We assume this representation given in Chapter 3, and focus on analyzing infinite-state numerical programs reading from infinite input streams. We introduce a novel concrete collecting semantics, defined by induction on the syntax of double programs, that expresses the behaviors of both program versions at the same time. We propose an abstraction of input streams able to prove that program versions reading from the same stream compute equal outputs. Then, we show how to leverage classic numerical abstract domains, such as polyhedra or octagons, to build an effective static analysis. Finally, we introduce a novel numerical domain to bound the differences between the values of the variables in the two programs, which has linear cost, and the right amount of relationality to express useful properties of software patches.

Then, we propose in Chapter 4 an algorithm to automate the synthesis of a double program from a pair of program versions. For most practical patches, the double programs obtained suffice to enable conclusive static analyses with linear numerical domains, even when program invariants are non-linear.

In Chapter 5, we turn to the analysis of patches of low-level C programs, *i.e.* programs whose behaviors depend on the representation of computer memory. This is the case of most C programs, especially in embedded software. To this aim, we implement our patch analysis on top of the MOPSA platform [97], which allows relational analyses based on weakly coupled cooperating abstract domains, and eases the lifting of an abstract domain from a toy language to a real-world language such as C. Our implementation benefits from the cell domain [127] implemented in MOPSA, a memory model permitting sound and precise analyses of low-level C programs. This domain represents the memory as a dynamic collection of scalar variables, termed cells, holding values for the scalar memory dereferences discovered during the analysis. It maintains a consistent abstract state despite the presence of overlapping cells introduced when the analyzed program by-passes the type-system of C to get byte-level access to memory. Patch analysis in this model allows us to successfully address a first portability property: robustness to variations of the offsets of scalar fields, such as those introduced by changes of the Application Binary Interface (ABI) of the target, compiler options or language extensions such as attributes of types or variables. Our implementation analyses successfully real patches from the repositories of the GNU core utilities and the Linux kernel.

To analyze realistic patches of C programs with this memory model requires expressive numerical abstractions that infer equalities between cells associated to the same scalar dereferences in the two versions of the memory. Such cells are indeed expected to be equal most of the time during program execution, with only local deviations. We thus optimize the memory model for this common case in Chapter 6, by representing these equalities symbolically. To this aim, we introduce so-called shared bi-cells, which represent pairs of cells from different program versions holding equal values. This optimization of the memory model enables successful analyses of some real-world patches using only non-relational numerical domains, which improves scalability dramatically.

Finally, we extend our memory domain in Chapter 7 to support a second portability property: portability across platforms with opposite byte-orders, *a.k.a.* endiannesses. We infer and represent symbolically equalities between cells, modulo byte-swapping. We introduce a novel symbolic predicate domain with near-linear cost to infer relations between individual bytes of the variables in the two programs, such as those established by bitwise arithmetic operations. Bitwise arithmetic is indeed used in programming patterns that are commonly used in endian-portable programs, *e.g.* to byte-swap scalar data. The resulting analysis allows analyzing successfully a real-world avionics software product of a million lines of C.

1.2.2 Contributions

The main contributions presented in this thesis are:

- We introduce a novel concrete collecting semantics, expressing the behaviors of two versions of a program at the same time. This semantics deals with programs reading from unbounded input streams. In the context of patch analysis, both program versions run in the same environment. In the context of portability analysis, they run on platforms with different ABIs, hence syntactically equal programs may have different semantics. We construct pairs of semantics to support these cases uniformly.
- We propose abstractions of input streams able to prove that programs that read from the same stream compute equal output values.
- We propose a joint memory abstraction able to infer equivalence relations between the memories of the two program versions, and represent them symbolically. This memory model deals soundly with ABI-dependent low-level C programs.
- We design structure layout portability analysis and endian portability analysis as extensions of patch analysis.
- We introduce a novel numerical domain to bound differences between the values of the variables in the two program versions, which has linear cost, and the right amount of relationality to express useful properties of software patches.
- We introduce a novel symbolic predicate domain to infer relations between individual bytes of the variables in the two programs, which has near-linear cost, and enough relationality to express (bitwise) arithmetic properties relevant to endian portability.
- We implemented our analyses on the Mopsa platform. Our prototype endian portability analysis is able to scale to large real-world industrial software, with zero false alarms. It is also able to analyze successfully smaller slices of open source software, such as Linux drivers.

Some of the results described in Chapters 3 to 7 have been subject to publications in workshops and symposiums [65, 66, 67, 68], and are presented here with multiple extensions.

1.2.3 Context of the work

Airbus avionics software. During the research presented in this thesis, I shared my time between research and my work as an avionics software engineer at Airbus[7], specializing in design and verification processes, methods and tools. The research was therefore geared towards applications to practical engineering problems of the Airbus avionics software department. In addition, this unique position gave me access to relevant industrial use cases to evaluate experimentally the scalability and the precision of static analyses.

The Mopsa project. I was also part of the Mopsa [134, 97] project[8], which aims at the modular development of a family of static analyzers for multiple languages and multiple properties. The development of Mopsa was started by Antoine Miné, Abdelraouf Ouadjaout, Matthieu Journault and Raphaël Monat. I implemented most of the results presented in this thesis on top of the Mopsa platform.

Collaborations. All the research described in this thesis is joint work with Antoine Miné. None of it could have happened without his brilliant suggestions. All the results from Chapter 4 are also joint work with Abdelraouf Ouadjaout. The successful implementation of our analyses on top of Mopsa owes a lot to his patient, dedicated support.

[7]This work was performed as part of a collaborative partnership between Sorbonne Université / CNRS (LIP6) and Airbus.

[8]This work was partially supported by the European Research Council under the Consolidator Grant Agreement 681393 – MOPSA.

Chapter 2

Background

In this chapter, we introduce a standard approach based on abstract interpretation to design a static analysis, that will be extended in the following chapters to construct our patch and portability analyses. Starting from a concrete collecting semantics, this approach constructs a computable abstract semantics through a sequence of abstraction steps. The resulting abstraction is sound by construction, and can be implemented into a static analyzer to infer automatically sound approximations of program properties.

We demonstrate this approach by a classic construction of a static analysis of numerical programs by induction in the syntax. We therefore start by introducing the syntax and (concrete) semantics of a simple numerical language called Nimp in Sec. 2.1. Then, we propose a lightweight introduction to the theory of Abstract Interpretation in Sec. 2.2, limited to the elements used in this thesis. We illustrate the concepts by constructing a simple abstraction of the Nimp language into a classical uncomputable invariant semantics of environments. Finally, we introduce numerical abstraction in Sec. 2.3.2, and illustrate it by abstracting the semantics of Nimp further, into a computable interval semantics.

This background chapter reuses material from comprehensive textbooks such as [132] and [46]. In particular, Sec. 2.1 follows broadly the structuration of [132, Sections 3.1 to 3.3], and Sec. 2.2 follows that of [132, Chapter 2].

2.1 Language

We introduce a simple numerical language called Nimp. Nimp is a toy language featuring standard imperative statements, and less classic operators for reading from an infinite input stream, and writing to an output stream. These operators will be abstracted away in the present background chapter to focus on a classic numerical semantics. They will however be of critical importance in Chapter 3, where they will be used to compare the semantics of two versions of a Nimp program. In this section, we describe the syntax of Nimp, formalize the semantics of statements and programs, and discuss the properties that can be proved from the concrete semantics. We broadly follow the structuration of [132, Sections 3.1 to 3.3].

$$
\begin{array}{rcll}
\textit{stat} & ::= & V \leftarrow \textit{expr} & V \in \mathcal{V} \\
& | & V \leftarrow \textbf{input}(a, b) & a, b \in \mathbb{Z} \\
& | & \textbf{output}(V) & \\
& | & \textbf{assert}(\textit{cond}) & \\
& | & \textbf{if } \textit{cond} \textbf{ then } \textit{stat} \textbf{ else } \textit{stat} & \\
& | & \textbf{while } \textit{cond} \textbf{ do } \textit{stat} & \\
& | & \textit{stat}; \textit{stat} & \\
& | & \textbf{skip} &
\end{array}
$$

Figure 2.1: Statements of Nimp programs

$$
\begin{array}{rcll}
\textit{expr} & ::= & V & V \in \mathcal{V} \\
& | & c & c \in \mathbb{Z} \\
& | & -\textit{expr} & \\
& | & \textit{expr} \diamond \textit{expr} & \diamond \in \{+, -, \times, /, \%\} \\
& | & \textbf{rand}(a, b) & a, b \in \mathbb{Z}
\end{array}
$$

Figure 2.2: Expressions of Nimp programs

$$
\begin{array}{rcll}
\textit{cond} & ::= & \textit{expr} \bowtie \textit{expr} & \bowtie \in \{\leq, \geq, =, \neq, <, >\} \\
& | & \neg \textit{cond} & \\
& | & \textit{cond} \diamond \textit{cond} & \diamond \in \{\wedge, \vee\}
\end{array}
$$

Figure 2.3: Conditions of Nimp programs

2.1.1 Syntax

Let us describe the syntax of Nimp. Statements *stat* are presented in Fig. 2.1. They are built on top of numeric expressions *expr* and Boolean conditions *cond*, defined in Fig. 2.2 and Fig. 2.3, respectively. We assume a given finite set of global $\mathbb{Z}$-valued integer variables $\mathcal{V}$. Nimp features classic imperative statements *stat*, such as assignments $V \leftarrow e$ of expression $e \in$ *expr* to variable $V \in \mathcal{V}$, conditionals **if** c **then** s_1 **else** s_2, **while** loops, and run-time assertions. The less standard $V \leftarrow$ **input**(a, b) statement reads a fresh value in the range $[a, b]$ from an infinite input stream, and stores it in variable V. The **output**(V) statement writes the value of variable V to an output stream. Numeric expressions *expr* include arithmetic expressions over variables and integer constants. They additionally support non-determinism: the expression **rand**(a, b) evaluates to a random value in the range $[a, b]$. Finally, the Boolean conditions *cond* used in conditionals, loops and assertions are comparisons of numeric expressions, or arbitrary Boolean combinations thereof.

$$\mathbb{E}[\![\, \mathit{expr} \,]\!] \in \mathcal{E} \to \mathcal{P}(\mathbb{Z})$$

$$\begin{array}{ll}
\mathbb{E}[\![\, V \,]\!]\rho & \triangleq \{\rho(V)\} \\
\mathbb{E}[\![\, c \,]\!]\rho & \triangleq \{c\} \\
\mathbb{E}[\![\, \mathbf{rand}(a,b) \,]\!]\rho & \triangleq \{\, x \mid a \leq x \leq b \,\} \\
\mathbb{E}[\![\, -e \,]\!]\rho & \triangleq \{\, -v \mid v \in \mathbb{E}[\![\, e \,]\!]\rho \,\} \\
\mathbb{E}[\![\, e_1 \diamond e_2 \,]\!]\rho & \triangleq \{\, v_1 \diamond v_2 \mid v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho,\ v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho,\ \diamond \notin \{\, /, \% \,\} \vee v_2 \neq 0 \,\} \quad \diamond \in \{+, -, \times, /\}
\end{array}$$

Figure 2.4: Semantics of numerical expressions

2.1.2 Semantics

In this section, we define the formal semantics of NIMP, *i.e.* a mathematical description of the possible behaviors of NIMP programs. A semantics is chosen according to a class of program properties of interest. We are interested in numerical invariants relating the inputs, the memory and the outputs of programs. We thus choose a suitable semantic domain to represent the possible memory states of programs.

Consider NIMP programs with integer-valued variables in $\mathcal{V}$: every memory state may be modeled as a function from $\mathcal{V}$ to $\mathbb{Z}$. We thus let $\mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$ denote the set of possible memory states.

Expression semantics

Let $e \in \mathit{expr}$ be some numerical expression. The semantics of e describes the result of its evaluation. Consider the syntax of expressions from Fig. 2.2. The result of evaluation may depend on the syntax of e, and on the memory state ρ at the time of evaluation:

- If e is some constant $c \in \mathbb{Z}$, then it evaluates always to the same value: c.
- If e is a non-deterministic value $\mathbf{rand}(a, b)$ where $a, b \in \mathbb{Z}^2$, then it may evaluate to several values. For instance, expression $[1, 3]$ may evaluate to values 1, 2 or 3.
- If e is some variable $V \in \mathcal{V}$, then its value depends on the memory state $\rho \in \mathcal{E}$. For instance, expression V evaluates to 42 if $\rho(V) = 42$.
- Finally, if e executes some illegal operation, such as a division by zero, then the result of evaluation cannot be defined by an integer value. For instance, expression $1/0$ does not evaluate to any integer value.

As a consequence, we define the semantics of expression e as an evaluation function $\mathbb{E}[\![\, e \,]\!] \in \mathcal{E} \to \mathcal{P}(\mathbb{Z})$. $\mathbb{E}[\![\, e \,]\!]\rho$ is the set of possible values of expression e in memory state $\rho \in \mathcal{E}$. It is defined on Fig. 2.4, by induction on the syntax of numeric expressions. For instance, $\mathbb{E}[\![\, e_1 + e_2 \,]\!]\rho$ is obtained by recording the sums of all possible values of sub-expressions e_1 and e_2 in memory ρ, which are given by $\mathbb{E}[\![\, e_1 \,]\!]\rho$ and $\mathbb{E}[\![\, e_2 \,]\!]\rho$.

Remark 1 (Semantics and run-time errors). $\mathbb{E}[\![\, e \,]\!]\rho$ filters away any evaluation which may divide by zero. For instance, $\mathbb{E}[\![\, 1/0 \,]\!]\rho = \emptyset$. In practice, a static analyzer based on this semantics would print an alarm message in this case. We could enrich the semantics with an error state, in addition to the memory state, to record any possible illegal operations.

$$\mathbb{C}[\![\, cond \,]\!] \in \mathcal{E} \to \mathcal{P}(\{\text{true}, \text{false}\})$$

$$\begin{array}{lll}
\mathbb{C}[\![\, \neg c \,]\!]\rho & \triangleq \{\, \neg v \mid v \in \mathbb{C}[\![\, c \,]\!]\rho \,\} & \\
\mathbb{C}[\![\, c_1 \diamond c_2 \,]\!]\rho & \triangleq \{\, v_1 \diamond v_2 \mid v_1 \in \mathbb{C}[\![\, c_1 \,]\!]\rho,\ v_2 \in \mathbb{C}[\![\, c_2 \,]\!]\rho \,\} & \diamond \in \{\wedge, \vee\} \\
\mathbb{C}[\![\, e_1 \bowtie e_2 \,]\!]\rho & \triangleq \{\, \text{true} \mid \exists v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho,\ v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho :\ v_1 \bowtie v_2 \,\} & \bowtie \in \{\leq, \geq, =, \neq, <, >\} \\
& \cup \{\, \text{false} \mid \exists v_1 \in \mathbb{E}[\![\, e_1 \,]\!]\rho,\ v_2 \in \mathbb{E}[\![\, e_2 \,]\!]\rho :\ v_1 \not\bowtie v_2 \,\} &
\end{array}$$

Figure 2.5: Semantics of conditional expressions

We do not do it here for the sake of conciseness, and because the work presented in this thesis does not aim at run-time error analysis.

Condition semantics

Conditions are comparisons between numeric expressions, or Boolean combinations thereof. As a consequence, their evaluations also depend on the memory state, and may return several possible results due to non-determinism. The semantics for some condition $c \in cond$ is thus an evaluation function $\mathbb{C}[\![\, c \,]\!] \in \mathcal{E} \to \mathcal{P}(\{\text{true}, \text{false}\})$. It is defined in Fig. 2.5, by induction on the syntax of conditions. For instance, the semantics $\mathbb{C}[\![\, \neg c \,]\!]\rho$ of condition $\neg c$ in memory ρ is defined as the set of the negations of the possible values condition c in ρ, given by $\mathbb{C}[\![\, c \,]\!]\rho$. Note that:

- $\text{true} \in \mathbb{C}[\![\, \neg c \,]\!]\rho \Leftrightarrow \text{false} \in \mathbb{C}[\![\, c \,]\!]$;
- $\text{true} \in \mathbb{C}[\![\, c_1 \wedge c_2 \,]\!]\rho \Leftrightarrow \text{true} \in \mathbb{C}[\![\, c_1 \,]\!] \cap \mathbb{C}[\![\, c_2 \,]\!]\rho$;
- $\text{true} \in \mathbb{C}[\![\, c_1 \vee c_2 \,]\!]\rho \Leftrightarrow \text{true} \in \mathbb{C}[\![\, c_1 \,]\!] \cup \mathbb{C}[\![\, c_2 \,]\!]\rho$.

Statement semantics

Let us now introduce the semantics of statements of Nimp programs. The semantics for some statement $s \in stat$ aims at describing the effect of the execution of s, *i.e.* the relation between the states of the program before and after executing s. The former are called pre-states of s, the latter are called post-states of s.

Nimp programs have memory states $\rho \in \mathcal{E}$. In addition, such programs write to some output stream with **output** statements. We therefore let program states record the (finite) sequence $o \in \mathbb{Z}^\star$ of output values. Moreover, Nimp programs read from some input stream, using **input** statements. Standard program semantics use non-deterministic choice to model such inputs. Yet, we anticipate that we would like to compare the behaviors of several versions of the same program, reading from the same input stream: see Chapter 3. As a consequence, we parameterize the semantics $\mathbb{S}[\![\, s \,]\!]_\iota$ of statement s by a sequence $\iota \in \mathbb{Z}^\omega$ of input values, and let program states record the current index $n \in \mathbb{N}$ in this sequence. Note that this sequence has to be infinite: indeed, due to non-determinism, the semantics maps every input stream to a (possibly infinite) set of executions, which can execute an unbounded number of input statements.

We thus let triples $(\rho, n, o) \in \Sigma \triangleq \mathcal{E} \times \mathbb{N} \times \mathbb{Z}^\star$ describe program states, where ρ defines the memory state, n is the current index in the input stream, and o records

$\mathbb{S}[\![\, stat \,]\!] \in \mathbb{Z}^\omega \to \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$

$$
\begin{array}{ll}
\mathbb{S}[\![\, \textbf{skip} \,]\!]_\iota X & \triangleq X \\
\mathbb{S}[\![\, V \leftarrow e \,]\!]_\iota X & \triangleq \{\, (\rho[V \mapsto v], n, o) \mid (\rho, n, o) \in X \wedge v \in \mathbb{E}[\![\, e \,]\!]\rho \,\} \\
\mathbb{S}[\![\, V \leftarrow \textbf{input}(a, b) \,]\!]_\iota X & \triangleq \{\, (\rho[V \mapsto \iota_n], n+1, o) \mid (\rho, n, o) \in X \wedge a \leq \iota_n \leq b \,\} \\
\mathbb{S}[\![\, \textbf{output}(V) \,]\!]_\iota X & \triangleq \{\, (\rho, n, o \cdot \rho(V)) \mid (\rho, n, o) \in X \,\} \\
\mathbb{S}[\![\, \textbf{assert}(c) \,]\!]_\iota & \triangleq \mathbb{S}[\![\, c? \,]\!] \\
\mathbb{S}[\![\, s \,;\, t \,]\!]_\iota & \triangleq \mathbb{S}[\![\, t \,]\!]_\iota \circ \mathbb{S}[\![\, s \,]\!]_\iota \\
\mathbb{S}[\![\, \textbf{if } c \textbf{ then } s \textbf{ else } t \,]\!]_\iota & \triangleq \mathbb{S}[\![\, s \,]\!]_\iota \circ \mathbb{S}[\![\, c? \,]\!] \dot\cup \mathbb{S}[\![\, t \,]\!]_\iota \circ \mathbb{S}[\![\, \neg c? \,]\!] \\
\mathbb{S}[\![\, \textbf{while } c \textbf{ do } s \,]\!]_\iota & \triangleq \mathbb{S}[\![\, \neg c? \,]\!] \circ (\mathbb{S}[\![\, s \,]\!]_\iota \circ \mathbb{S}[\![\, c? \,]\!])^* \\
\quad \text{where} \quad \mathbb{S}[\![\, c? \,]\!] X & \triangleq \{\, (\rho, n, o) \in X \mid \text{true} \in \mathbb{C}[\![\, c \,]\!]\rho \,\}
\end{array}
$$

Figure 2.6: Semantics of NIMP statements

the sequence of past outputs. Given some statement $s \in \mathit{stat}$ and some input stream $\iota \in \mathbb{Z}^\omega$, we then let $\mathbb{S}[\![\, s \,]\!]_\iota$ describe the relation between pre-states $(\rho, n, o) \in \Sigma$ and post-states $(\rho', n', o') \in \Sigma$ of statement s. A standard choice of semantic domain would be to let $\mathbb{S}_\mathcal{R}[\![\, s \,]\!]_\iota \in \mathcal{P}(\Sigma \times \Sigma)$. Yet, we anticipate that we will compute abstractions of this semantics using numerical domains that represent sets of states. We therefore tailor the formalization of the semantic domain to handle sets of states rather than relations over states. As $\mathcal{P}(\Sigma \times \Sigma) \simeq \Sigma \to \mathcal{P}(\Sigma)$, we may represent the relation over Σ with its image function $\mathbb{S}_\mathcal{F}[\![\, s \,]\!]_\iota \in \Sigma \to \mathcal{P}(\Sigma)$, defined by $\mathbb{S}_\mathcal{F}[\![\, s \,]\!]_\iota \rho \triangleq \{\, \rho' \mid (\rho, \rho') \in \mathbb{S}_\mathcal{R}[\![\, s \,]\!]_\iota \,\}$. However, this semantic function has different domain and codomain. This is inconvenient, as semantic functions should be composed to reflect the composition of statements, e.g., in sequences. We thus extend the semantic function to $\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$, as a $\cup$-morphism:

$$\mathbb{S}[\![\, s \,]\!]_\iota X \triangleq \cup\{\, \mathbb{S}_\mathcal{F}[\![\, s \,]\!]_\iota \rho \mid \rho \in X \,\} \tag{2.1}$$

Given a set of program states $(\rho, n, o) \in X \in \mathcal{P}(\Sigma)$ before executing statement s, the semantic function $\mathbb{S}[\![\, s \,]\!]_\iota$ returns the set of possible program states $(\rho', n', o') \in \mathbb{S}[\![\, s \,]\!]_\iota X$ after executing statement s.

The semantics $\mathbb{S}[\![\, s \,]\!] \in \mathbb{Z}^\omega \to \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ is defined on Fig. 2.6, by induction on the syntax of statements $s \in \mathit{stat}$ of NIMP programs. It builds on the semantics $\mathbb{E}[\![\, e \,]\!] \in \mathcal{E} \to \mathcal{P}(\mathbb{Z})$ of expressions $e \in \mathit{expr}$, and $\mathbb{C}[\![\, c \,]\!] \in \mathcal{E} \to \mathcal{P}(\{\text{true}, \text{false}\})$ of conditions $c \in \mathit{cond}$.

Atomic statements. We first describe the semantics of base cases of the syntax of statements: the empty statement, assignments, input and output statements, and the **assert**(c) statement.

The semantics of the **skip** statement is the identity function, as the statement has no effect on the program state. The semantics of an assignment $V \leftarrow e$ on a set of program states X updates the memory state ρ in each program state $(\rho, n, o) \in X$ independently:

$X \leftarrow \mathbf{input}(-10, 10);\ Y \leftarrow \mathbf{input}(-10, 10);\ X \leftarrow X + Y;$
$Y \leftarrow \mathbf{rand}(-10, 10) + \mathbf{rand}(-10, 10);$

Figure 2.7: Input statement versus non-determistic choice

it evaluates e in memory ρ, and changes the current program state to a set of new states (ρ', n, o), such that $\rho'(V) \in \mathbb{E}[\![e]\!]\rho$ is a possible value for e, and $\rho'(W) = \rho(W)$ if $W \neq V$. The semantics of an input statement $V \leftarrow \mathbf{input}(a, b)$ assigns the value at the current index in the input stream to V, and increments the index in all program states. Note that $\mathbf{input}(a, b)$ returns only if the input value at the current index is in the range $[a, b]$.

Remark 2 (Input statement versus non-determistic choice). Atomic statements $V \leftarrow \mathbf{input}(a, b)$ and $V \leftarrow \mathbf{rand}(a, b)$ are similar, in that they both assign to V a value in the range $[a, b]$. Yet, the former establishes a relation between the current memory state and the contents of the input stream, while the latter does not. Consider for instance the program P shown on Fig. 2.7. Starting from the initial state $\Sigma_0 = ([X \mapsto 0, Y \mapsto 0], 0, \emptyset)$, the set of reachable program states when reading from a given stream $\iota \in \mathbb{Z}^\omega$ is $\mathbb{S}[\![P]\!]_\iota \{ \Sigma_0 \} = \{ ([X \mapsto i_0 + i_1, Y \mapsto v], 2, \emptyset) \mid i_0, i_1 \in [-10, 10] \wedge v \in [-20, 20] \}$. This semantics expresses properties such as "P reads the two first values of ι, and stores their sum into X". In contrast, the only property expressed for Y is that its value ranges in $[-20, 20]$.

The semantics of an output statement $\mathbf{output}(V)$ extends the sequence of outputs in each program state with the value of variable V in this state. Note that we write $o \cdot v$ to denote the sequence extending sequence o with value v.

Remark 3 (Generalization to multiple streams). We model one input stream and one output stream only, to simplify the presentation, while realistic programs may use several input and output streams. This does not restrict the applicability of our approach, as the generalization to multiple input and output streams is straightforward.

$\mathbf{assert}(c)$ statements do not depend on the input stream. We use the dedicated filter $\mathbb{S}[\![c?]\!] \in \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ to formalize their semantics. $\mathbb{S}[\![c?]\!]$ filters away any program state where condition c cannot evaluate to true. In practice, a static analyzer based on this semantics would print an alarm message in this case. We could enrich the semantics with an error state, in addition to the memory state, to record any possible illegal operations. We do not do it here for the sake of conciseness. Note that $\mathbb{S}[\![c_1 \wedge c_2?]\!] = \mathbb{S}[\![c_1?]\!] \cap \mathbb{S}[\![c_2?]\!]$ and $\mathbb{S}[\![c_1 \vee c_2?]\!] = \mathbb{S}[\![c_1?]\!] \cup \mathbb{S}[\![c_2?]\!]$. Furthermore, $\mathbb{S}[\![c?]\!]$ is a complete $\cup$-morphism: $\mathbb{S}[\![c?]\!](\cup_{i \in I} X_i) = \cup_{i \in I} \mathbb{S}[\![c?]\!] X_i$ for arbitrary families of sets $(X_i)_{i \in I}$.

Compound statements. We now describe the semantics of inductive cases of the syntax of statements: sequences of statements, selection and iteration statements.

The semantics of a sequence of statements $s\,;\,t$ is defined as the composition of semantic functions of s and t. This models accurately the fact that $s\,;\,t$ adds the effects of t to those of s.

The semantics of a selection statement **if** c **then** s **else** t is defined as the union of the semantics of the **then** branch s and the **else** branch t. The semantics of the **then** branch applies the semantics of s to the pre-states that may satisfy condition c, which are selected using the $\mathbb{S}[\![c?]\!]$ filter. The semantics of the **else** branch applies the semantics of t to the pre-states that may fail to satisfy condition c, which are selected using the $\mathbb{S}[\![\neg c?]\!]$ filter. Note that we use the symbol $\dot\cup$ to denote the pointwise lifting of $\cup$: $f \dot\cup f' \triangleq \sigma \in \mathbb{Z}^\omega \mapsto f(\sigma) \cup f'(\sigma)$.

Let us now describe the semantics of iteration statements **while** c **do** s. Starting with some pre-state, such statements iterate the body s of the loop, as long as the memory state satisfies the condition c.

Let X be a set of pre-states, and let $X' \triangleq \mathbb{S}[\![\textbf{while } c \textbf{ do } s]\!]_\iota X$ be the associated set of possible post-states. The evaluation of the condition c of the loop divides X into two subsets: the set of states which may satisfy c, and the set of states which may fail to satisfy c. Let $X'_{\geq 1} \triangleq \mathbb{S}[\![c?]\!]X$ be the former, and $X'_0 \triangleq \mathbb{S}[\![\neg c?]\!]X$ be the latter. The loop has no effect on the states in X'_0, hence $X'_0 \subseteq X'$. On the contrary, the body s of the loop is run on states in $X'_{\geq 1}$, resulting in states $\mathbb{S}[\![s]\!]_\iota X'_{\geq 1}$. Then, the condition is evaluated again for these states, dividing them again into two subsets: states in $X'_{\geq 2} \triangleq \mathbb{S}[\![c?]\!] \circ \mathbb{S}[\![s]\!]_\iota X'_{\geq 1}$ may satisfy c, while states in $X'_1 \triangleq \mathbb{S}[\![\neg c?]\!] \circ \mathbb{S}[\![s]\!]_\iota X'_{\geq 1}$ may fail to. States in X'_1 exit the loop at this point, hence $X'_1 \subseteq X'$. On the contrary, the body s of the loop is run again on states in $X'_{\geq 2}$, and the previous process is repeated, resulting in a infinite family of sets of states $(X'_0, X'_1, \ldots)$, such that $\forall n \in \mathbb{N} : X'_n \triangleq \mathbb{S}[\![\neg c?]\!] \circ (\mathbb{S}[\![s]\!]_\iota \circ \mathbb{S}[\![c?]\!])^n X \subseteq X'$. X'_n denotes the set of post-states that may be reached after exactly n iterations. Conversely, all states in X' are post-states that may be reached after finitely many iterations. Hence $X' = \cup_{n\in\mathbb{N}} X'_n$.

As consequence, $\mathbb{S}[\![\textbf{while } c \textbf{ do } s]\!]_\iota = \cup_{n\in\mathbb{N}} \mathbb{S}[\![\neg c?]\!] \circ (\mathbb{S}[\![s]\!]_\iota \circ \mathbb{S}[\![c?]\!])^n$. As $\mathbb{S}[\![c?]\!]$ is a complete $\cup$-morphism, this can be re-written as

$$\mathbb{S}[\![\textbf{while } c \textbf{ do } s]\!]_\iota = \mathbb{S}[\![\neg c?]\!] \circ (\mathbb{S}[\![s]\!]_\iota \circ \mathbb{S}[\![c?]\!])^*$$

where we note $f^* \triangleq \dot\cup_{n\in\mathbb{N}} f^n$ for any $f \in \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$.

Program semantics

We are now ready to define the formal semantics of Nimp programs. We choose to define it in terms of observable inputs and outputs, abstracting away the internal memory state. This is a standard choice when we are interested in properties of observable behaviors. For instance, [27, Sec 3.5] defines the semantics of the source language of the CompCert C compiler in terms of traces of observable input-output operations. We thus denote by $\mathbb{P}[\![p]\!] \in \mathbb{Z}^\omega \to \mathcal{P}(\mathbb{Z}^*)$ the semantics of programs $p \in$ *stat*. The semantics $\mathbb{P}[\![p]\!]_\iota$ of a program p reading from an input stream $\iota \in \mathbb{Z}^\omega$ is the set of sequences of values that terminating executions of p may output.

Recall, from Sec. 2.1.2, that program states are triples (ρ, n, o), where ρ is a memory state, n is a number of executed **input** statements, and o is a sequence of output values. Consider the initial memory state ρ_0 with all variables zero-initialized: $\forall V \in \mathcal{V} : \rho_0(V) =$

$I \leftarrow 0$;
$X \leftarrow \mathbf{input}(-100, 100)$;
while $X \geq 0$ **do**
  if $(X/2) \times 2 < X$ **then output**(X);
  $I \leftarrow I + 1$;
  $X \leftarrow \mathbf{input}(-100, 100)$
done

Figure 2.8: Filtering inputs

0. Consider the related initial program state, with zero-initialized input index, and empty output sequence $x_0 \triangleq (\rho_0, 0, \epsilon)$. Given some input stream $\iota \in \mathbb{Z}^\omega$, $\mathbb{S}[\![p]\!]_\iota \{ x_0 \}$ is the set of possible post-states of statement p.

Definition 1 (Semantics of Nimp programs)**.** We thus define the semantics of program p as:

$$\mathbb{P}[\![p]\!]_\iota \triangleq \{ o \mid (\rho, n, o) \in \mathbb{S}[\![p]\!]_\iota \{ x_0 \} \}$$

2.1.3 Properties

$\mathbb{P}$ is suitable to express a number of properties of terminating executions of Nimp programs, such as so-called "functional properties" expressing relations between inputs and outputs.

Example 4 (Filtering inputs)**.** For instance, a property such as "the program reads a sequence of integers in the range $[-100, 100]$ from the input stream, until some negative input value occurs, and writes the sequence of indexes of odd inputs to the output stream" may be expressed as:
$\forall \iota \in \mathbb{Z}^\omega : \exists N \in \mathbb{N} : (\iota_N \in [-100, 0] \wedge \forall n < N : \iota_n \in [0, 100]) \Rightarrow \forall o \in \mathbb{P}[\![p]\!]_\iota : |o| = |S_\iota(0, N)| \wedge o_0 = \min S_\iota(0, N) \wedge \forall n < |o| : o_{n+1} = \min S_\iota(o_n + 1, N)$,
where $S_\iota(p, q) \triangleq \{ n \in \mathbb{N} \mid p \leq n < q \wedge \iota_n \equiv 1 \mod 2 \}$.
Consider the program p on Fig. 2.8.
$\mathbb{S}[\![p]\!]_\iota \{ x_0 \} = \{ ([I \mapsto N, X \mapsto \iota_N], N + 1, o) \mid N \in \mathbb{N} \wedge \iota_N \in [-100, 0] \wedge \forall n < N : \iota_n \in [0, 100] \}$, where o satisfies the specification above.
Hence $\mathbb{P}[\![p]\!]_\iota \triangleq \{ o \mid (\rho, n, o) \in \mathbb{S}[\![p]\!]_\iota \{ x_0 \} \}$ can be proved to satisfy the specification.

Remark 4 (Properties of non-terminating executions). Though our definition of $\mathbb{P}$ focuses on terminating executions, some (functional) properties of non-terminating executions can be verified using **assert** statements. For instance, non-terminating executions of the program shown on Fig. 2.9 read only positive inputs. The semantic domain could be extended to express such properties formally (*e.g.* adding propagated error states), but we do not do it here for simplicity.

```
X ← input(−100, 100);
while true do
    assert(X ≥ 0);
    X ← input(−100, 100)
done
```

Figure 2.9: Non-terminating NIMP program

2.1.4 Proofs

As shown in previous sections, our formal semantics for NIMP statements and programs is able to express precise properties of program executions. It can thus be used as the mathematical reference for conducting a formal proof that a given NIMP program (or statement) meets a given specification. However, such proofs cannot be automated, because the semantics $\mathbb{S}$ and $\mathbb{P}$ are not computable.

Indeed, the elements of the semantic domain $\mathcal{D} \triangleq \mathbb{Z}^\omega \to \mathcal{P}(\Sigma)$ of $\mathbb{S}$ are infinite sets that cannot be represented in computer memory. Recall that $\Sigma = (\mathcal{V} \to \mathbb{Z}) \times \mathbb{N} \times \mathbb{Z}^\star$. As a consequence, the semantic transfer functions of atomic statements such as $\mathbb{S}[\![V \leftarrow e]\!]$ and $\mathbb{S}[\![c?]\!]$ are not computable. The same situation occurs for lattice operators such as $\cup$. Moreover, the semantic transfer function of loops may require an infinite number of iterations.

To side-step this computability issue, one could be tempted to make the set of memory states $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$ finite by restricting the values of variables to the range of machine integers. One could additionally restrict the maximum lengths of valid input and output sequences to some fixed numbers to handle finitely many variables. However, this would still result in a huge domain, so that a naive representation of sets in extension would not be practicable. For instance, even the set of memory states of a program with only 9 32-bit variables is larger than current estimates of the number of protons in the observable universe (10^{80}). Similarly, while semantic transfer functions of atomic statements and lattice operators would be computable in principle with these restrictions, evaluating them on every program state individually would not be tractable in practice. Finally, the transfer function of while loops would iterate in finite, but extremely long sequences.

We thus turn to another approach to overcome this computability issue. We will leverage the theory of abstract interpretation to replace the functions operating on the semantic domain by sound, computable approximations thereof, operating on an alternate semantic domain. The former are called *concrete*, the latter *abstract*.

2.2 Elements of abstract interpretation

Abstract interpretation is an invaluable tool for designing formal methods. In particular, it enables the design of static analyses that are sound by construction by formalizing the relations between a (concrete) program semantics of reference and (abstract) approximations thereof handled by a static analyzer. In this section, we propose a lightweight

introduction to the theory of Abstract Interpretation, limited to the elements used in this thesis. We nonetheless follow broadly the structuration of the more comprehensive [132, Chapter 2].

Abstract interpretation is a mathematical theory where:

1. Concrete and abstract semantic domains are formalized as sets of objects equipped with partial order relations. We thus review elements of order theory in Sec. 2.2.1.
2. Semantic functions such as our concrete semantics $\mathbb{S}[\![s]\!]$ of Sec. 2.1.2 are defined as fixpoints of operators over these partially ordered sets. We thus introduce relevant fixpoint theorems in Sec. 2.2.2.
3. These mathematical tools are used to construct abstract semantics as sound approximations of concrete ones. We thus introduce abstract domains in Sec. 2.2.3, and related operator approximation techniques in Sec. 2.2.4.

2.2.1 Order theory

Semantic domains such as our $\mathbb{Z}^\omega \to \mathcal{P}(\Sigma)$ of Sec. 2.1.2 represent properties of program computations. These properties are naturally ordered by logical implication. Such domains are thus modeled as partially ordered sets, or richer order structures, such as (complete) lattices. Most of the domains used in this thesis are complete lattices, but the framework is not limited to them. Let us introduce these order structures.

Posets

Definition 2 (Properties of binary relations)**.** A binary relation $\bowtie \in \mathcal{P}(X \times X)$ over a set X is called:

- reflexive if: $\forall x \in X : x \bowtie x$;
- transitive if: $\forall x, y, z \in X : x \bowtie y \wedge y \bowtie z \implies x \bowtie z$;
- symmetric if: $\forall x, y \in X : x \bowtie y \implies y \bowtie x$;
- anti-symmetric if: $\forall x, y \in X : x \bowtie y \wedge y \bowtie x \implies x = y$.

Definition 3 (Partial order, Poset)**.** A binary relation $\sqsubseteq$ over a set X is called a partial order if $\sqsubseteq$ is reflexive, transitive and anti-symmetric. $(X, \sqsubseteq)$ is then called a partially ordered set (*a.k.a. poset*).

Example 5 (Poset)**.** Given a set X, $(\mathcal{P}(X), \subseteq)$ is a poset. In particular, using the notations of Sec. 2.1.2, the set of memory states $(\mathcal{P}(\mathcal{V} \to \mathbb{Z}), \subseteq)$ and the set of program states $(\mathcal{P}(\Sigma), \subseteq)$ and are both posets.

Definition 4 (Lower and upper bounds of two elements)**.** Let $(X, \sqsubseteq)$ be a poset, and $a, b \in X$. $c \in X$ is called:

- a lower bound of a and b if $c \sqsubseteq a$ and $c \sqsubseteq b$;
- an upper bound of a and b if $a \sqsubseteq c$ and $b \sqsubseteq c$.

Definition 5 (glb and lub of two elements)**.** Let $(X, \sqsubseteq)$ be a poset, and $a, b \in X$. $c \in X$ is called:

- the greatest lower bound (*a.k.a. glb*) of a and b if $c' \sqsubseteq c$ for all lower bound c' of a and b;
- the least upper bound (*a.k.a. lub*) of a and b if $c \sqsubseteq c'$ for all upper bound c' of a and b.

Remark 5 (Unicity of glb and lub). If a and b have a glb (resp. lub) in $(X, \sqsubseteq)$ then it is unique, and denoted $a \sqcap b$ (resp. $a \sqcup b$).

Example 6 (glb and lub)**.** Let $S = \{ a, b, c, d \}$. Elements $X = \{ a, b \}$ and $Y = \{ b, c \}$ of the poset $(\mathcal{P}(S), \subseteq)$ have lub $X \cup Y = \{ a, b, c \}$ and glb $X \cap Y = \{ b \}$.

Definition 6 (Lower and upper bounds of a set)**.** Let $(X, \sqsubseteq)$ be a poset, and $A \in \mathcal{P}(X)$ be a subset. $x \in X$ is called:

- a lower bound of A if $x \sqsubseteq a$ for all $a \in A$;
- an upper bound of A if $a \sqsubseteq x$ for all $a \in A$.

Definition 7 (glb and lub of a set)**.** Let $(X, \sqsubseteq)$ be a poset, and $A \in \mathcal{P}(X)$ be a subset. $x \in X$ is called:

- the greatest lower bound (*a.k.a. glb*) of A if $x' \sqsubseteq x$ for all lower bound x' of A;
- the least upper bound (*a.k.a. lub*) of A if $x \sqsubseteq x'$ for all upper bound x' of A.

Remark 6 (glb and lub of a set). If A admits a glb (resp. lub) in $(X, \sqsubseteq)$ then it is unique, and denoted $\sqcap A$ (resp. $\sqcup A$).

Definition 8 (Chain)**.** Let $(X, \sqsubseteq)$ be a poset, and $C \in \mathcal{P}(X)$ be a subset. C is called a chain if all elements of C are comparable, *i.e.* $\forall a, b \in C : a \sqsubseteq b \vee b \sqsubseteq a$.

Hasse diagrams are directed graphs commonly used as graphic representations of posets. Vertices are elements of the set. The greater elements are placed above the smaller ones. Arcs depict the order relation between vertices. Arcs that can be inferred by transitivity are omitted for readability. In particular, Hasse diagrams help visualize glbs, lubs and chains.

Example 7 (Hasse diagram)**.** The Hasse diagrams for the poset $(\mathcal{P}(\{ a, b, c, d \}), \subseteq))$ is shown on Fig. 2.10.

Lattices

Definition 9 (Lattice)**.** $(X, \sqsubseteq, \sqcup, \sqcap)$ is called a lattice if $(X, \sqsubseteq)$ is a poset such that $a \sqcap b$ and $a \sqcup b$ exist for all $a, b \in X$.

Definition 10 (Complete lattice)**.** $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is called a complete lattice if $(X, \sqsubseteq)$ is a poset such that $\sqcap A$ and $\sqcup A$ exist for all $A \in \mathcal{P}(X)$. In particular, $\bot = \sqcup \emptyset$ denotes the least element, and $\top = \sqcup X$ denotes the greatest element.

Property 1 (Powerset lattice)**.** *For every set X, $(\mathcal{P}(X), \subseteq, \cup, \cap, \emptyset, X)$ is a complete lattice, called the powerset lattice of X.*

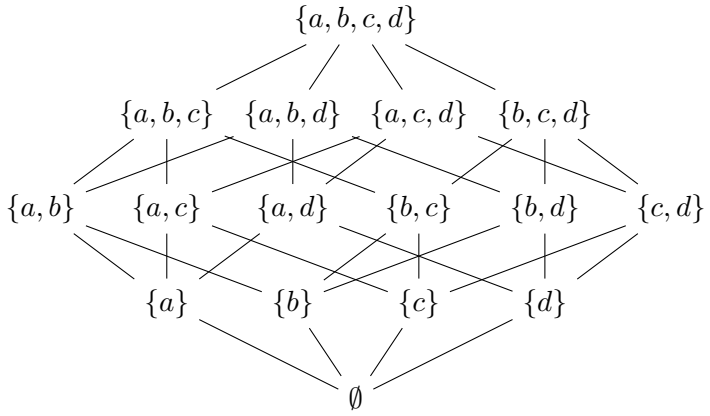


Figure 2.10: Hasse diagram for $(\mathcal{P}(\{\, a, b, c, d \,\}), \subseteq)$

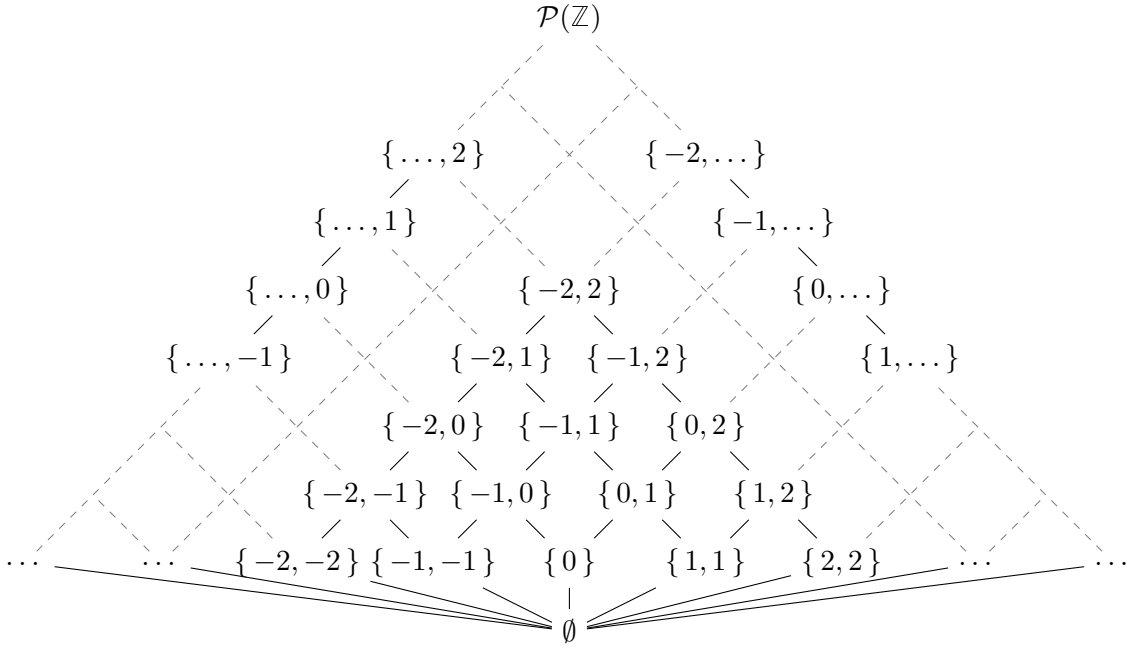


Figure 2.11: Hasse diagram for $(\mathcal{P}(\mathbb{Z}), \subseteq)$

Example 8 (Powerset lattices)**.** Fig. 2.10 shows the Hasse diagram for the powerset lattice of $\{\, a, b, c, d \,\}$. The Hasse diagram for the powerset lattice of $\mathbb{Z}$ is shown on Fig. 2.11.

Remark 7 (Maximal chains). Fig. 2.10 shows that all chains of $(\mathcal{P}(\{\, a, b, c, d \,\}), \subseteq)$ are finite. Maximal chains have length 5. In contrast, Fig. 2.11 shows that $(\mathcal{P}(\mathbb{Z}), \subseteq)$ has

infinite chains, such as $\bigcup_{n\in\mathbb{N}} \{ 0, \ldots, n \}$.

Property 2 (Pointwise lifting of a lattice)**.** *Let $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ be a complete lattice and S be a set. Then the derived order structure $((S \to X), \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top})$ defined by:*

$$\begin{aligned} f \mathrel{\dot{\sqsubseteq}} f' &\overset{\triangle}{\iff} \forall s \in S : f(s) \sqsubseteq f'(s) \\ \forall s \in S : (f \mathrel{\dot{\sqcup}} f)(s) &\triangleq f(s) \sqcup f'(s) \\ \forall s \in S : (f \mathrel{\dot{\sqcap}} f)(s) &\triangleq f(s) \sqcap f'(s) \\ \forall s \in S : \dot{\bot}(s) &\triangleq \bot \\ \forall s \in S : \dot{\top}(s) &\triangleq \top \end{aligned}$$

is a complete lattice.

Example 9 (Semantic domain as a pointwise lifted lattice)**.** The semantic domain $\mathcal{D} \triangleq \mathbb{Z}^\omega \to \mathcal{P}(\Sigma)$ introduced in Sec. 2.1.2 enjoys the structure of a complete lattice $(\mathcal{D}, \dot{\subseteq}, \dot{\cup}, \dot{\cap}, \dot{\emptyset}, \dot{\Sigma})$, as it lifts the powerset lattice of Σ pointwise.

Property 3 (Coalescent pointwise lifting of a lattice)**.** *Let $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ be a complete lattice and S be a set. Let $\bot_0 \notin S \to X$ be a new least element. Then the derived order structure $((S \to (X \setminus \{ \bot \})) \cup \{ \bot_0 \}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \bot_0, \dot{\top})$ defined by:*

$$\begin{aligned} f \mathrel{\dot{\sqsubseteq}} f' &\overset{\triangle}{\iff} \forall s \in S : f(s) \sqsubseteq f'(s) \\ \forall s \in S : (f \mathrel{\dot{\sqcup}} f)(s) &\triangleq f(s) \sqcup f'(s) \\ \forall s \in S : (f \mathrel{\dot{\sqcap}} f)(s) &\triangleq f(s) \sqcap f'(s) \\ \forall s \in S : \dot{\top}(s) &\triangleq \top \end{aligned}$$

is a complete lattice.

Remark 8 (Coalescent pointwise lifting of a lattice). Property 3 is often used to define program semantics based on invariant memory states, as liftings of non-relational numerical abstract domains. We will show the example of the Interval abstraction in Sec. 2.3.2.

2.2.2 Functions, operators and fixpoints

Functions and operators

Let us list some useful properties of functions over order structures.

Definition 11 (Monotonicity)**.** Given two posets $(X_1, \sqsubseteq_1)$ and $(X_2, \sqsubseteq_2)$, a function $f \in X_1 \to X_2$ is called monotone if:

$$\forall x, y \in X_1 : x \sqsubseteq_1 y \implies f(x) \sqsubseteq_2 f(y)$$

Remark 9 (Monotonicity). If we interpret the elements of posets as program properties and their partial orders as models of logical implication, monotonicity is a necessary property for a function to preserve logical implication.

Example 10 (Monotonicity)**.** Given a statement $s \in \mathit{stat}$ and a stream $\iota \in \mathbb{Z}^\omega$, the semantic transfer function $\mathbb{S}[\![s]\!]_\iota$ defined on Fig. 2.6 of Sec. 2.1.2 is monotone.

Definition 12 (Complete $\sqcup$-morphism)**.** Given two complete lattices $(X_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \bot_1, \top_1)$ and $(X_2, \sqsubseteq_2, \sqcup_2, \sqcap_2, \bot_2, \top_2)$ a function $f \in X_1 \to X_2$ is called a complete $\sqcup$-morphism if:

$$\forall C \in \mathcal{P}(X_1) : f(\sqcup_1 C) = \sqcup_2 f(C)$$

Concrete semantic transfer functions of programs are typically defined as complete $\sqcup$-morphisms. This property implies indeed that it suffices to observe the behavior of a program for each possible input separately, and then to join all behaviors to obtain the set of all possible program behaviors. For instance, we intentionally defined $\mathbb{S}[\![s]\!]_\iota$ as a complete $\cup$-morphism by Equation 2.1.

Definition 13 (operator)**.** We call operator on a set X any function $f \in X \to X$.

Semantic functions are operators of the semantic domain. By extension, two-variable functions over a set X are also called operators. For instance, $\sqcap$ and $\sqcup$ are called *lattice operators* of the lattice $(X, \sqsubseteq, \sqcup, \sqcap)$.

Definition 14 (Extensivity)**.** An operator f on a poset $(X, \sqsubseteq)$ is called extensive if

$$\forall x \in X : x \sqsubseteq f(x)$$

Example 11 (Extensivity)**.** $n \mapsto 2 \times n$ is extensive on $(\mathbb{N}, \leq)$.

Definition 15 (Reductivity)**.** An operator f on a poset $(X, \sqsubseteq)$ is called reductive if

$$\forall x \in X : f(x) \sqsubseteq x$$

Example 12 (Reductivity)**.** $n \mapsto \lfloor n/2 \rfloor$ is reductive on $(\mathbb{N}, \leq)$.

Fixpoints

Definition 16 (Fixpoint)**.** Let $f \in X \to X$ be an operator over a poset $(X, \sqsubseteq)$.

- $x \in X$ is called a fixpoint of f if $f(x) = x$. We denote as $\mathrm{fp}(f)$ the set of fixpoints of f;
- if $\mathrm{fp}(f)$ has a least element, it is called the the least fixed point of f, and denoted $\mathrm{lfp}\, f$.

Theorem 1 (Tarski's fixpoint theorem [167])**.** *Let $f \in X \to X$ be an operator over a complete lattice $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. If f is monotone, then $\mathrm{fp}(f)$ is complete lattice, and $\mathrm{lfp}\, f = \sqcap \{ x \in X \mid f(x) \sqsubseteq x \}$.*

Theorem 2 (Kleene's constructive fixpoint theorem (weakened))**.** *Let $f \in X \to X$ be an operator over a complete lattice $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. If f is a complete $\sqcup$-morphism, then* $\operatorname{lfp} f = \sqcup\{\, f^n(\bot) \mid n \in \mathbb{N} \,\}$.

Theorem 2 can be found in [48] with weaker assumptions. In particular, X need not be a lattice. We use the strong assumptions above to avoid introducing concepts that are not necessary in the context of this thesis.

This theorem is constructive, in that it expresses $\operatorname{lfp} f$ as the limit of an iteration starting from $\bot$. It also suggests that unbounded iterations of operators may be expressed as fixpoints. The semantic function of loops of Nimp programs, introduced in Sec. 2.1.2, features such an iteration:

$$\mathbb{S}[\![\textbf{while } c \textbf{ do } s]\!]_\iota = \mathbb{S}[\![\neg c?]\!] \circ \cup_{n\in\mathbb{N}} (\mathbb{S}[\![s]\!]_\iota \circ \mathbb{S}[\![c?]\!])^n$$

Using Theorem 2, we can indeed rewrite it as:

$$\mathbb{S}[\![\textbf{while } c \textbf{ do } s]\!]_\iota X = \mathbb{S}[\![\neg c?]\!](\operatorname{lfp} F^X_\iota) \tag{2.2}$$

where $F^X_\iota(Y) \triangleq X \cup \mathbb{S}[\![s]\!]_\iota(\mathbb{S}[\![c?]\!]Y)$.

2.2.3 Domain abstraction

We are now ready to formalize the relationships between concrete and abstract semantic domains, by giving a mathematical meaning to the notion of abstraction.

Concrete and abstract domains

The minimum structure for defining a concrete semantic domain is a poset $(C, \leq)$ (although we often have complete lattices in this thesis). Elements of C represent properties of programs executions, which are sets of program behaviors. $\leq$ is an information order compatible with implication: larger elements represent coarser properties, hence more behaviors.

Example 13 (Concrete semantic domain for Nimp)**.** The concrete semantic domain $\mathcal{D} \triangleq \mathbb{Z}^\omega \to \mathcal{P}(\Sigma)$ that we introduced in Sec. 2.1.2 is equipped with a complete lattice structure through pointwise lifting (see Example 9).

An abstract domain provides an alternate representation for a concrete domain, that may represent fewer elements, *i.e.* fewer properties. Abstract domains require thus the same minimal structure as concrete domains: a poset $(A, \sqsubseteq)$.

Example 14 (Abstracting away Nimp streams)**.** In the case of Nimp programs, one may choose to focus on numerical properties of variables, rather than on the relationships between output input and values. One will thus disregard input streams $\iota \in \mathbb{Z}^\omega$, indexes $n \in \mathbb{N}$ in these streams, and sequences of outputs $o \in \mathbb{Z}^*$, and keep only memory states $\rho \in \mathcal{E}$ as program states of interest. The resulting abstract "memory-only" domain is then $\hat{\mathcal{D}} \triangleq \mathcal{P}(\mathcal{E})$, a (powerset) complete lattice.

Remark 10 (Abstracting away NIMP streams). We will follow up on Example 14 in the rest of chapter, and construct step by step an abstract semantics expressing numerical invariants of reachable memory states of NIMP programs. The result will be classic semantics that ignores input and ouput streams, as in [132, Sec. 3.3]. While we make this choice to simplify the presentation of generic static analysis techniques in the current chapter, we will reintroduce streams in the concrete and abstract semantics in Chapter 3, in order to compare the semantics of two versions of a program.

We now describe the relation between concrete and abstract domains in the framework of abstract interpretation, and the related notion of soundness.

Concretization and soundness

The minimum connection between a concrete domain $(C, \leq)$ and an abstract domain $(A, \sqsubseteq)$ is a monotone function $\gamma \in A \to C$ that defines the meaning of abstract elements of A in terms of concrete elements of C. $\gamma(a)$ is the concrete property represented by $a \in A$. γ is called the concretization of A. Monotonicity ensures that abstract implication is compatible with concrete implication.

An abstract property $a \in A$ is said to be sound with respect to a concrete property $c \in C$ if c represents at least as many behaviours as c, *i.e.* $c \leq \gamma(a)$. Moreover a is called an exact abstraction of c if a and c represent the same set of behaviors, *i.e.* $c = \gamma(a)$.

Definition 17 (Sound and exact abstractions)**.** Let $(C, \leq)$ and $(A, \sqsubseteq)$ be two posets. Let $\gamma \in C \to A$ be a monotone function, and $c \in C$. $a \in A$ is called:

1. a sound abstraction of c for γ if $c \leq \gamma(a)$;
2. an exact abstraction of c for γ if $c = \gamma(a)$.

Definition 18 (Concretization)**.** Let $(C, \leq)$ and $(A, \sqsubseteq)$ be two posets. $\gamma \in A \to C$ is called a concretization function if it is monotone and every element of C has a sound abstraction for γ.

Example 15 (Concretization function for NIMP's memory-only domain)**.** Following up on Example 14, we propose the concretization $\hat{\gamma} \in \hat{\mathcal{D}} \to \mathcal{D}$ defined by $\hat{\gamma}(X)_\iota = X \times \mathbb{N} \times \mathbb{Z}^*$. It means that a set of (abstract) memory states represents the set of possible (concrete) program states that have these memory states (and arbitrary input and output sequences).

Remark 11 (Composition of abstractions). An immediate consequence of Definitions 18 and 17 is that abstractions compose. If $(A, \sqsubseteq)$ abstracts $(C, \leq)$ via $\gamma \in C \to A$, $(A', \sqsubseteq')$ abstracts $(A, \sqsubseteq)$ via $\gamma' \in A' \to A$, then $(A', \sqsubseteq')$ abstracts $(C, \leq)$ via $\gamma \circ \gamma'$. This allows incremental approaches to abstraction: abstract domains can be developed in layers. We will use this approach extensively in this thesis. We will illustrate it in the current chapter, by following up on Examples 14 and 15.

Some abstract domains rely solely on a concretization function to define their relation to a concrete domain. For instance, this it is the case of the polyhedra abstract domain [51], that represents sets of real numbers as affine inequalities. Most abstractions

used in this thesis will nonetheless enjoy a richer connection to concrete domains. They will feature an additional function $\alpha \in C \to A$, called abstraction function, such that the pair (α, γ) forms a Galois connection.

Galois connections

Definition 19 (Galois connection)**.** Let $(C, \leq)$ and $(A, \sqsubseteq)$ be two posets. A pair $(\alpha, \gamma) \in (C \to A) \times (A \to C)$ is called a Galois connection, and denoted $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, if:

1. γ is monotone;
2. α is monotone;
3. $\gamma \circ \alpha$ is extensive;
4. $\alpha \circ \gamma$ is reductive.

The monotonicity of γ and α ensure that concrete and abstract implications are compatible: more precise (concrete) properties are represented by more precise (abstract) properties. The extensivity of $\gamma \circ \alpha$ states that $\alpha(c)$ is a sound abstraction of c for all concrete element $c \in C$. The reductivity of $\alpha \circ \gamma$ additionally ensures that it is the least for the abstract order.

This last property is a key interest Galois connections: the abstraction function returns the so-called "best abstraction" of each concrete element.

Property 4 (Best abstraction)**.** *Let* $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ *be a Galois connection.*

$$\forall c \in C : \alpha(c) = \sqcap \{\, a \mid c \leq \gamma(a) \,\}$$

The following characterization is often used in practice to prove that two domains are related by a Galois connection.

Property 5 (Characteristic property of Galois connections)**.** *Let* $(C, \leq)$ *and* $(A, \sqsubseteq)$ *be two posets. A pair* $(\alpha, \gamma) \in (C \to A) \times (A \to C)$ *is a Galois connection if and only if:*

$$\forall (c, a) \in C \times A : c \leq \gamma(a) \iff a \sqsubseteq \alpha(c)$$

Example 16 (Galois connection for Nimp's memory-only domain)**.** Following up on Example 15, we propose the abstraction $\hat{\alpha} \in \mathcal{D} \to \hat{\mathcal{D}}$ defined by $\hat{\alpha}(f) = \bigcup_{\sigma \in \mathbb{Z}^\omega} \{\, \rho \mid (\rho, n, o) \in f(\sigma) \,\}$. We have $(\mathcal{D}, \dot{\subseteq}) \underset{\hat{\alpha}}{\overset{\hat{\gamma}}{\leftrightarrows}} (\hat{\mathcal{D}}, \subseteq)$.

Example 17 (Absence of Galois connection)**.** The polyhedra abstract domain [51] represents sets of real numbers as affine inequalities. The meaning of abstract elements is defined by a concretization function. However, no abstraction function can be proposed to form a Galois connection, as not all sets of real numbers have a best abstraction in this domain. Indeed, this domain can be seen, geometrically, as abstracting a set of points as a convex polyhedron. The best abstraction would be the smallest enclosing polyhedron. Unfortunately, such a polyhedron does not exist for some sets of points, such as discs. Fig. 2.12 illustrates this fact.

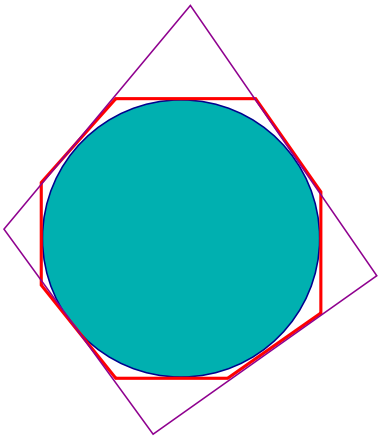


Figure 2.12: Absence of Galois connection (no polyhedral abstraction)

Although this is not necessary to have a Galois connection, several Galois connections introduced in this thesis will feature a unique abstract representation $a \in A$ for every concrete property $c \in C$. Such particular Galois connections are called *Galois embeddings*.

Definition 20 (Galois embedding)**.** A Galois connection $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ is called a Galois embedding, and denoted $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows\!\!\!\!\!\!\rightarrow}} (A, \sqsubseteq)$, if one of the following equivalent properties holds:

1. α is surjective;
2. γ is injective;
3. $\alpha \circ \gamma = Id$.

Example 18 (Galois embedding for Nimp's memory-only domain)**.** The Galois connection introduced in Example 16 is additionally a Galois embedding:

$$(\mathcal{D}, \dot{\subseteq}) \underset{\hat{\alpha}}{\overset{\hat{\gamma}}{\leftrightarrows\!\!\!\!\!\!\rightarrow}} (\hat{\mathcal{D}}, \subseteq)$$

In a Galois embedding, the abstract domain is isomorphic to a subset of the concrete domain (the image of $\hat{\gamma}$, *i.e.* $\hat{\gamma}(A)$). The abstraction of elements of $\hat{\gamma}(A)$ does not lose information, while the abstraction of other elements may lose information. In some cases, we introduce completely isomorphic abstractions, that do not lose information on any concrete elements. The interest of such abstractions is usually a change of representation, to make later abstractions easier to express. In those cases, we have Galois isomorphisms.

Definition 21 (Galois isomorphism)**.** A Galois connection $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ is called a Galois isomorphism, and denoted $(C, \leq) \underset{\alpha}{\overset{\gamma}{\twoheadleftarrow\!\!\!\!\!\!\rightarrow}} (A, \sqsubseteq)$, if one of the following equivalent properties holds:

1. α is bijective;

2. γ is bijective;
3. $\gamma \circ \alpha = \alpha \circ \gamma = Id$.

Example 19 (no Galois isomorphism for NIMP memory states)**.** The Galois connection introduced in Example 16 is not a Galois isomorphism.

Property 6 (Composing Galois connections)**.** *Let* $(X_1, \sqsubseteq_1) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} (X_2, \sqsubseteq_2)$ *and* $(X_2, \sqsubseteq_2) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} (X_3, \sqsubseteq_3)$ *be two Galois connections. Then* $(X_1, \sqsubseteq_1) \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} (X_3, \sqsubseteq_3)$ *is a Galois connection.*

Remark 12 (Composition of abstractions)*.* Remark 11 noted that abstractions can be developed in layers. Property 6 additionally ensures that optimal abstractions can be developed incrementally.

Property 7 (Lifting Galois connections pointwise)**.** *Let* $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ *be a Galois connection, and* S *be a set. Then* $(S \to C, \dot{\leq}) \underset{\dot{\alpha}}{\overset{\dot{\gamma}}{\leftrightarrows}} (S \to A, \dot{\sqsubseteq})$ *is a Galois connection, where*

$$
\begin{aligned}
f \dot{\leq} f' &\overset{\triangle}{\iff} \forall s \in S : f(s) \leq f'(s) \\
f \dot{\sqsubseteq} f' &\overset{\triangle}{\iff} \forall s \in S : f(s) \sqsubseteq f'(s) \\
\dot{\alpha}(f) &\triangleq \alpha \circ f \\
\dot{\gamma}(f) &\triangleq \gamma \circ f
\end{aligned}
$$

Property 8 (Coalescent pointwise lifting of a Galois connection)**.** *Let* $(C, \leq, \vee, \wedge, \bot_C, \top_C)$ *and* $(A, \sqsubseteq, \sqcup, \sqcap, \bot_A, \top_A)$ *be a two complete lattices such that* $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$*, and let* S *be a set. Let* $((S \to (C \setminus \{ \bot_C \})) \cup \{ \bot_C^0 \}, \dot{\leq}, \dot{\vee}, \dot{\wedge}, \bot_C^0, \dot{\top}_C)$ *and* $((S \to (A \setminus \{ \bot_A \})) \cup \{ \bot_A^0 \}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \bot_A^0, \dot{\top}_A)$ *be the coalescent pointwise liftings of* $(C, \leq, \vee, \wedge, \bot_C, \top_C)$ *and* $(A, \sqsubseteq, \sqcup, \sqcap, \bot_A, \top_A)$*, respectively, as defined by Property 3. Then*

$$
((S \to (C \setminus \{ \bot_C \})) \cup \{ \bot_C^0 \}, \dot{\leq}) \underset{\bar{\alpha}}{\overset{\bar{\gamma}}{\leftrightarrows}} ((S \to (A \setminus \{ \bot_A \})) \cup \{ \bot_A^0 \}, \dot{\sqsubseteq})
$$

is a Galois connection such that:

$$
\bar{\gamma}(f) \triangleq \begin{cases} \bot_C^0 & \textbf{if } f = \bot_A^0 \\ \gamma \circ f & \textbf{otherwise} \end{cases}
$$

$$
\bar{\alpha}(f) \triangleq \begin{cases} \bot_A^0 & \textbf{if } f = \bot_C^0 \\ \alpha \circ f & \textbf{otherwise} \end{cases}
$$

Remark 13 (Lifting Galois connections pointwise)*.* Property 8 is often used to define program semantics based on invariant memory states, as liftings of non-relational numerical abstract domains. We will show the example of the Interval abstraction in Sec. 2.3.2.

2.2.4 Operator and fixpoint approximation

A distinctive feature of abstract interpretation is to replace uncomputable semantic operators over a concrete domain by operators on an abstract domain. Abstract operators must be designed as sound approximations of the concrete ones. When abstract operators are computable, we obtain an effective static analysis, the results of which are guaranteed to be sound with respect to the concrete semantics. Now that we have established formal relations between concrete and abstract domains, (through concretizations or Galois connections), we are ready to discuss techniques for approximating operators. A critical subset thereof deal with approximating fixpoints.

Operator approximation

In Sec. 2.2.3, we have defined a notion of sound (resp. exact) abstraction in the context of domains. We extend these notions to domain operators, such as semantic transfer functions and lattice operators.

Definition 22 (Sound and exact operator abstraction)**.** Let $(C, \leq)$ and $(A, \sqsubseteq)$ be two posets, and let $f \in C \to C$ and $g \in A \to A$ be two operators. g is called:

1. a sound abstraction of f if $f \circ \gamma \dot{\leq} \gamma \circ g$;
2. an exact abstraction of f if $f \circ \gamma = \gamma \circ g$.

Remark 14 (operator abstraction)*.* Definition 22 generalizes to two-variable operators over a poset, such as lattice operators.

In the context of a Galois connection, we can additionally extend the notion of best abstraction to operators.

Definition 23 (Best operator abstraction)**.** Let $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ be a Galois connection, and let $f \in C \to C$ and $g \in A \to A$ be two operators. g is called the best abstraction of f if $g(a)$ is the best abstraction of $f(\gamma(a))$ for all $a \in A$.

Property 9 (Best operator abstraction)**.** *Let* $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ *be a Galois connection, and let* $f \in C \to C$ *be an operator. Then* $\alpha \circ f \circ \gamma$ *is the best abstraction of* f*.*

Example 20 (Operator abstraction for Nimp's memory-only domain)**.** The best abstraction of the operator $\mathbb{S}[\![V \leftarrow e]\!]$ over $\mathcal{D}$ defined in Fig. 2.6 is the operator $\hat{\mathbb{S}}[\![V \leftarrow e]\!]$ over $\hat{\mathcal{D}}$ defined as:

$$\hat{\mathbb{S}}[\![V \leftarrow e]\!] X \triangleq \{ \rho[V \mapsto v] \mid \rho \in X \wedge v \in \mathbb{E}[\![e]\!]\rho \}$$

More generally, Fig. 2.13 shows the best abstractions of the concrete semantics of atomic Nimp statements. As noted in Remark 10, this semantics abstracts away I/O streams: inputs are abstracted as nondeterministic choice, and outputs have no effect. As a consequence, this semantics cannot express some functional properties of interest of Nimp programs, such as the property stated in Example 4 for the program shown on Fig. 2.8.

$\hat{\mathbb{S}}[\![\, stat \,]\!] \in \hat{\mathcal{D}} \to \hat{\mathcal{D}}$

$$
\begin{array}{ll}
\hat{\mathbb{S}}[\![\, \mathbf{skip} \,]\!]\, X & \triangleq X \\
\hat{\mathbb{S}}[\![\, V \leftarrow e \,]\!]\, X & \triangleq \{\, \rho[V \mapsto v] \mid \rho \in X \wedge v \in \mathbb{E}[\![\, e \,]\!]\rho \,\} \\
\hat{\mathbb{S}}[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!] & \triangleq \hat{\mathbb{S}}[\![\, V \leftarrow \mathbf{rand}(a,b) \,]\!] \\
\hat{\mathbb{S}}[\![\, \mathbf{output}(V) \,]\!] & \triangleq \hat{\mathbb{S}}[\![\, \mathbf{skip} \,]\!] \\
\hat{\mathbb{S}}[\![\, \mathbf{assert}(c) \,]\!] & \triangleq \hat{\mathbb{S}}[\![\, c? \,]\!] \\
\quad \text{where} \quad \hat{\mathbb{S}}[\![\, c? \,]\!]\, X & \triangleq \{\, \rho \in X \mid \mathrm{true} \in \mathbb{C}[\![\, c \,]\!]\rho \,\}
\end{array}
$$

Figure 2.13: Abstract (memory-only) semantics of Nimp atomic statements

$\hat{\mathbb{S}}[\![\, stat \,]\!] \in \hat{\mathcal{D}} \to \hat{\mathcal{D}}$

$$
\begin{array}{ll}
\hat{\mathbb{S}}[\![\, s\,;\, t \,]\!] & \triangleq \hat{\mathbb{S}}[\![\, t \,]\!] \circ \hat{\mathbb{S}}[\![\, s \,]\!] \\
\hat{\mathbb{S}}[\![\, \mathbf{if}\ c\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!] & \triangleq \hat{\mathbb{S}}[\![\, s \,]\!] \circ \hat{\mathbb{S}}[\![\, c? \,]\!] \mathbin{\dot{\cup}} \hat{\mathbb{S}}[\![\, t \,]\!] \circ \hat{\mathbb{S}}[\![\, \neg c? \,]\!] \\
\hat{\mathbb{S}}[\![\, \mathbf{while}\ c\ \mathbf{do}\ s \,]\!]\, X & \triangleq \hat{\mathbb{S}}[\![\, \neg c? \,]\!]\, (\hat{\Pi}_X)
\end{array}
$$

where $\forall \iota : \mathrm{lfp}\, F_\iota^X \subseteq \hat{\gamma}(\hat{\Pi}_X)$

Figure 2.14: Abstract (memory-only) semantics of Nimp compound statements

Operator composition

Abstract operators can be composed, preserving the soundness of approximations.

Property 10 (Composition of sound abstract operators)**.** *Let $(C, \leq)$ be a concrete domain, and $(A, \sqsubseteq)$ be a related abstract domain. Let $f, f' \in C \to C$ be concrete operators, and $g, g' \in A \to A$ be abstract operators. Then:*

1. *if g and g' are sound abstractions of f and f', respectively, and f is monotonic, then $g \circ g'$ is a sound abstraction of $f \circ f'$.*
2. *if g and g' are exact abstractions of f and f', respectively, then $g \circ g'$ is an exact abstraction of $f \circ f'$.*

This property suggests an abstraction scheme for complex concrete operators defined as compositions of simpler ones. Such a scheme is well-suited to abstracting concrete semantic transfer functions defined by induction on the syntax of a programming language.

Example 21 (Composition of sound abstract operators)**.** The concrete semantics shown on Fig. 2.6 of Sec. 2.1.2 is defined by induction on the syntax of Nimp programs. For instance, $\mathbb{S}[\![\, s\,;\, t \,]\!]_\iota = \mathbb{S}[\![\, t \,]\!]_\iota \circ \mathbb{S}[\![\, s \,]\!]_\iota$ in the concrete domain $\mathcal{D}$. The semantics of the sequential composition may thus be abstracted as $\hat{\mathbb{S}}[\![\, s\,;\, t \,]\!] \triangleq \hat{\mathbb{S}}[\![\, t \,]\!] \circ \hat{\mathbb{S}}[\![\, s \,]\!]$ in the abstract domain $\hat{\mathcal{D}}$. More generally, Fig. 2.14 shows a possible abstraction of the semantics of compound Nimp statements, which follows the same structure, by induction on the syntax.

Remark 15 (Composition of sound abstract operators). Property 10 guarantees soundness, but not optimality: the composition of best abstractions may not be the best abstraction of the composition.

Figs. 2.13 and 2.14 illustrate how an abstract semantics can be soundly derived by approximating a concrete one. Yet, the construction is not complete. First, Fig. 2.14 requires a sound abstraction $\hat{\Pi}_X$ of the least fixpoint of the operator F_ι^X over the concrete domain. We will fill this first gap in the following of this section by fixpoint approximation. Second, the abstract semantics $\hat{\mathbb{S}}$ is not computable, as $\mathcal{D}$ is infinite. We will fill this second gap in Sec. 2.3.2 by numerical abstraction.

Fixpoint approximation

A first approach is to approximate a concrete least fixpoint by Kleenian iteration in the abstract.

Theorem 3 (Kleenian fixpoint approximation (weakened)). *Let $(C, \leq, \vee, \wedge, \bot_C, \top^C)$ be a complete lattice and $(A, \sqsubseteq)$ be a poset with a minimal element $\bot$.*

If $f \in C \to C$ is a complete $\vee$-morphism, $g \in A \to A$ is a sound abstraction of f, and the sequence $\{\, g^n(\bot) \,|\, n \in \mathbb{N} \,\}$ has a limit $l \in A$, then l is a sound abstraction of $\operatorname{lfp} f$.

This theorem can be found with weaker assumptions, *e.g.* in [132, Sec. 2.3]. In particular, C need not be a lattice. We use the strong assumptions above to avoid introducing concepts that are not necessary in the context of this thesis.

This approach is effective in practice if iterates of g from $\bot$ stabilize after a finite number of iterations, and if this number is reasonably small. The case occurs in some abstract domains, such as finite domains, or lattices featuring only finite chains. However, many abstract domains of interest feature infinite chains [54]. This is in particular the case of the widely used Interval Domain, as we will see in Sec. 2.3.2. To overcome this limitation, [48] introduced an iteration acceleration technique known as *widening*, that guarantees that a sound approximation of the concrete least fixpoint is computed in finite time.

Definition 24 (Widening operator). Let $(A, \sqsubseteq)$ be a poset and $\nabla \in A \times A \to A$ be a binary operator. ∇ is called a widening operator if:

1. $\forall x, y \in A : x \sqsubseteq x \nabla y \wedge y \sqsubseteq x \nabla y$
2. for all sequences $(x_n)_{n\in\mathbb{N}} \in A^{\mathbb{N}}$, the sequence $(y_n)_{n\in\mathbb{N}} \in A^{\mathbb{N}}$ defined by $y_0 \triangleq x_0$ and $y_{n+1} \triangleq y_n \nabla x_{n+1}$ is ultimately stationary, *i.e.* $\exists l \in \mathbb{N} : \forall n \geq l : y_n = y_l$.

Theorem 4 (Fixpoint approximation with widening). *Let $(C, \leq, \vee, \wedge, \bot_C, \top^C)$ be a complete lattice, and $(A, \sqsubseteq)$ be a poset with a minimal element $\bot$. Let $f \in C \to C$ be a monotonic operator, $g \in A \to A$ be a sound abstraction of f, and ∇ be a widening operator. Then the sequence $(y_n)_{n\in\mathbb{N}} \in A^{\mathbb{N}}$ defined by $y_0 \triangleq \bot$ and $y_{n+1} \triangleq y_n \nabla g(y_n)$ is ultimately stationary, and its limit is a sound abstraction of* $\operatorname{lfp} f$.

$\hat{\mathbb{S}}[\![\, stat \,]\!] \in \hat{\mathcal{D}} \to \hat{\mathcal{D}}$

$$
\begin{aligned}
\hat{\mathbb{S}}[\![\, s\,;\, t \,]\!] &\triangleq \hat{\mathbb{S}}[\![\, t \,]\!] \circ \hat{\mathbb{S}}[\![\, s \,]\!] \\
\hat{\mathbb{S}}[\![\, \textbf{if } c \textbf{ then } s \textbf{ else } t \,]\!] &\triangleq \hat{\mathbb{S}}[\![\, s \,]\!] \circ \hat{\mathbb{S}}[\![\, c? \,]\!] \dot{\cup} \hat{\mathbb{S}}[\![\, t \,]\!] \circ \hat{\mathbb{S}}[\![\, \neg c? \,]\!] \\
\hat{\mathbb{S}}[\![\, \textbf{while } c \textbf{ do } s \,]\!]\, X &\triangleq \hat{\mathbb{S}}[\![\, \neg c? \,]\!] \left((\hat{G}_X)^l(\bot) \right)
\end{aligned}
$$

where

$$
\begin{aligned}
l &\triangleq \min\{\, n \in \mathbb{N} \,|\, (\hat{G}_X)^n(\bot) = (\hat{G}_X)^{n+1}(\bot) \,\} \\
\hat{G}_X(Y) &\triangleq Y \,\nabla\, \hat{F}^X(Y) \\
\hat{F}^X(Y) &\triangleq X \cup \hat{\mathbb{S}}[\![\, s \,]\!] (\hat{\mathbb{S}}[\![\, c? \,]\!]\, Y)
\end{aligned}
$$

Figure 2.15: abstract (memory-only) semantics of NIMP compound statements with widening

Widening operators can be defined specifically in each abstract domain, as part of a trade-off between the cost of abstract iterations and the precision of the approximations. We show the naive widening of Example 22 as a witness that such operators exist.

Example 22 (Naive widening)**.**

$$
x \nabla y \triangleq \begin{cases} x & \textbf{if } \; y \sqsubseteq x \\ \top & \textbf{otherwise} \end{cases}
$$

The naive widening of Example 22 can be used in any abstract domain that contains a maximum element $\top$. It typically leads to very coarse approximations in practice, hence more refined widenings are usually defined in abstract domains. We will show an example in the case of the Interval abstract domain in Sec. 2.3.2.

Example 23 (Fixpoint approximation with widening)**.** Following up on Example 21 and assuming a widening operator ∇ for our abstract domain $\hat{\mathcal{D}}$, we can update Fig. 2.14 with a sound approximation of the least fixpoint $\hat{\Pi}_X$ of the concrete operator F^X_ι. The result is shown on Fig. 2.15. We perform iterations with widening of the sound approximation $\hat{F}^X$ of F^X_ι, relying on Theorem 4 for iterations to stabilize on a sound approximation of the concrete least fixpoint in finite time.

2.3 Static analysis

Static analysis aims at inferring properties of programs automatically, by computing with sound abstractions of their concrete semantics. We have introduced the (uncomputable) concrete semantics $\mathbb{S}$ of NIMP programs in Sec. 2.1. Then, we have presented in Sec. 2.2 how an abstract semantics can be designed as a sound approximation of a concrete one. We have illustrated this approach by deriving $\hat{\mathbb{S}}$, a simple abstraction of $\mathbb{S}$. However, $\hat{\mathbb{S}}$ is not yet suitable for an effective static analysis. Indeed, the elements of the abstract domain $\hat{\mathcal{D}} = \mathcal{P}(E)$ are not representable in computer memory, hence $\hat{\mathbb{S}}$, (as $\mathbb{S}$) is not

computable. We need to abstract $\hat{\mathbb{S}}$ further to obtain a computable abstraction $\mathbb{S}^\sharp$ of $\mathbb{S}$, resulting in an effective static analysis. Nonetheless, the modular approach to operator approximation illustrated in Sec. 2.2.4 allows clarifying the requirements for such a computable abstract semantics.

2.3.1 Generic computable abstract semantics

A computable abstraction of $\hat{\mathbb{S}}$ (and thus of $\mathbb{S}$) is a semantics of NIMP programs $\mathbb{S}^\sharp$ that meets the following requirements:

- $\mathbb{S}^\sharp[\![s]\!]$ is an operator over a poset $(\mathcal{D}^\sharp, \sqsubseteq^\sharp)$ such that:
  - $\mathcal{D}^\sharp$ is a set of computer-representable values with a least element $\bot^\sharp$ and a greatest element $\top^\sharp$, and
  - an effective algorithm is available to test whether $x^\sharp \sqsubseteq^\sharp y^\sharp$ holds for all $x^\sharp, y^\sharp \in \mathcal{D}^\sharp$;
- a monotonic (concretization) function $\gamma \in \mathcal{D}^\sharp \to \mathcal{D}$ exists such that $\gamma(\bot^\sharp) = \emptyset$ and $\gamma(\top^\sharp) = \mathcal{E}$;
- an effective algorithm is available to compute the operator $\mathbb{S}^\sharp[\![V \leftarrow e]\!]$ (resp. $\mathbb{S}^\sharp[\![e_1 \bowtie e_2]\!]$, resp. $\cup^\sharp$, resp. $\cap^\sharp$) over $\mathcal{D}^\sharp$, such that $\mathbb{S}^\sharp[\![V \leftarrow e]\!]$ (resp. $\mathbb{S}^\sharp[\![e_1 \bowtie e_2]\!]$, resp. $\cup^\sharp$, resp. $\cap^\sharp$) is a sound abstraction of $\hat{\mathbb{S}}[\![V \leftarrow e]\!]$ (resp. $\hat{\mathbb{S}}[\![e_1 \bowtie e_2]\!]$ resp. $\cup$, resp. $\cap$);
- an effective algorithm is available to compute a widening operator ∇.

In addition, a Galois connection $(\mathcal{D}, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{D}^\sharp, \sqsubseteq^\sharp)$ may optionally exist. In this case, best abstractions can be constructed for all operators.

Using the modular abstraction scheme introduced in in Sec. 2.2.4, we obtain the abstract semantics of NIMP programs $\mathbb{S}^\sharp$ shown on Fig. 2.16. This semantics is parameterized by a generic abstract domain $(\mathcal{D}^\sharp, \sqsubseteq^\sharp)$, approximations $\cup^\sharp$ and $\cap^\sharp$ of lattice operators, and transfer functions of assignments $V \leftarrow e$ and tests $e_1 \bowtie e_2$. This generic construction guarantees that $\mathbb{S}^\sharp[\![s]\!]$ is a sound approximation of $\hat{\mathbb{S}}[\![s]\!]$ (and of $\mathbb{S}[\![s]\!]$) for all statements s, and that the computation of $\mathbb{S}^\sharp[\![s]\!]X^\sharp$ terminates for all abstract properties $X^\sharp \in \mathcal{D}^\sharp$. It is therefore suitable as the design of a static analyzer that is parametric in the choice of an abstract domain.

To move on from a generic analyzer to a practical instance, we must provide an instance of the generic abstract domain $\mathcal{D}^\sharp$. $\mathcal{D}^\sharp$ abstracts sets of memory states in $\mathcal{D} = \mathcal{P}(\mathcal{V} \to \mathbb{Z})$, *i.e.* sets of points in a vector space of dimension $|\mathcal{V}|$. Such an abstract domain is called a *numerical abstract domain*.

2.3.2 Numerical abstract domains

Numerical domains are essential components of static analyzers. They express indeed invariants that are key to prove safety properties of programs, such as the range of individual variables, or algebraic relations between several variables. Numerical abstractions

$$
\left.\begin{array}{l} \cup^\sharp \in \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp \\ \cap^\sharp \in \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp \\ \nabla \in \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp \end{array}\right] \text{given}
$$

$$
\left.\begin{array}{l} \mathbb{S}^\sharp[\![V \leftarrow e]\!] \\ \mathbb{S}^\sharp[\![c?]\!] \end{array}\right] \text{given}
$$

$$\mathbb{S}^\sharp[\![\mathit{stat}]\!], \mathbb{S}^\sharp[\![\mathit{cond}\ ?]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$$

$$
\begin{array}{rl}
\mathbb{S}^\sharp[\![\textbf{skip}]\!] X^\sharp & \triangleq X^\sharp \\
\mathbb{S}^\sharp[\![V \leftarrow \textbf{input}(a,b)]\!] & \triangleq \mathbb{S}^\sharp[\![V \leftarrow \textbf{rand}(a,b)]\!] \\
\mathbb{S}^\sharp[\![\textbf{output}(V)]\!] & \triangleq \mathbb{S}^\sharp[\![\textbf{skip}]\!] \\
\mathbb{S}^\sharp[\![\textbf{assert}(c)]\!] & \triangleq \mathbb{S}^\sharp[\![c?]\!] \\
\mathbb{S}^\sharp[\![s\,;\, t]\!] & \triangleq \mathbb{S}^\sharp[\![t]\!] \circ \mathbb{S}^\sharp[\![s]\!] \\
\mathbb{S}^\sharp[\![\textbf{if } c \textbf{ then } s \textbf{ else } t]\!] & \triangleq \mathbb{S}^\sharp[\![s]\!] \circ \mathbb{S}^\sharp[\![c?]\!] \dot{\cup}^\sharp \mathbb{S}^\sharp[\![t]\!] \circ \mathbb{S}^\sharp[\![\neg c?]\!] \\
\mathbb{S}^\sharp[\![\textbf{while } c \textbf{ do } s]\!] X^\sharp & \triangleq \mathbb{S}^\sharp[\![\neg c?]\!] \left((G^\sharp_{X^\sharp})^l(\bot) \right) \text{ where} \\
l & \triangleq \min\{\, n \in \mathbb{N} \mid (G^\sharp_{X^\sharp})^n(\bot) = (G^\sharp_{X^\sharp})^{n+1}(\bot) \,\} \\
G^\sharp_{X^\sharp}(Y^\sharp) & \triangleq Y^\sharp \nabla F^\sharp_{X^\sharp}(Y^\sharp) \\
F^\sharp_{X^\sharp}(Y^\sharp) & \triangleq X^\sharp \cup^\sharp \mathbb{S}^\sharp[\![s]\!](\mathbb{S}^\sharp[\![c?]\!] Y^\sharp) \\
\mathbb{S}^\sharp[\![\text{true}\ ?]\!] X^\sharp & \triangleq X^\sharp \\
\mathbb{S}^\sharp[\![\text{false}\ ?]\!] X^\sharp & \triangleq \bot^\sharp \\
\mathbb{S}^\sharp[\![(c_1 \wedge c_2)\ ?]\!] & \triangleq \mathbb{S}^\sharp[\![c_1\ ?]\!] \dot{\cap}^\sharp \mathbb{S}^\sharp[\![c_2\ ?]\!] \\
\mathbb{S}^\sharp[\![(c_1 \vee c_2)\ ?]\!] & \triangleq \mathbb{S}^\sharp[\![c_1\ ?]\!] \dot{\cup}^\sharp \mathbb{S}^\sharp[\![c_2\ ?]\!]
\end{array}
$$

Figure 2.16: Abstract semantics of NIMP programs (parameterized by an abstract domain $\mathcal{D}^\sharp$)

have thus been widely studied, resulting in a variety of numerical domains in the literature. Popular examples include: Signs [47], Intervals [47], Congruences [82], Linear equalities [102], Polyhedra [51], Ellipses [73], Zones [135], Octagons [128], Exponentials [74], Gauges [173], and Zonotopes [124].

The goal of the present chapter is not to give an in-depth overview of available numerical abstract domains. Indeed, we mainly aim at demonstrating how a computable abstract semantics can be derived to enable a sound static analysis. We will thus focus on the simple Interval abstract domain. We will nonetheless provide a brief overview of other domains, featuring different trade-offs between expressiveness and cost.

Interval abstraction

Interval analysis aims at inferring numerical invariants of the form $a \leq \mathtt{x} \leq b$, where $\mathtt{x}$ is a program variable, and a and b are numerical constants discovered during the analysis. Such invariants are called non-relational, as they express numerical properties

$$(\hat{\mathcal{D}}, \subseteq) \underset{\alpha_C}{\overset{\gamma_C}{\leftrightarrows}} (\tilde{\mathcal{D}}, \tilde{\subseteq})$$

$$\alpha_C(\hat{X}) \triangleq \begin{cases} \tilde{\bot} & \textbf{if } X = \emptyset \\ V \mapsto \bigcup_{\rho \in \hat{X}} \{\, \rho(V) \,\} & \textbf{otherwise} \end{cases}$$

$$\gamma_C(\tilde{X}) \triangleq \begin{cases} \emptyset & \textbf{if } \tilde{X} = \tilde{\bot} \\ \bigcup \{\, \rho \mid \forall V \in \mathcal{V} : \rho(V) \in \tilde{X}(V) \,\} & \textbf{otherwise} \end{cases}$$

Figure 2.17: Cartesian abstraction

(ranges) of individual variables separately. In contrast, such non-relational invariants cannot express relations between multiple variables. The Interval domain is thus called a non-relational domain.

Remark 16 (non-relational domains). Other widely used non-relational domains include the Sign and Congruence domains. The Sign (resp. Congruence) domain expresses invariants of the form $\mathtt{x} \bowtie 0$ (resp. $\mathtt{x} \in a\mathbb{Z} + b$) where $\mathtt{x}$ is a program variable, $\bowtie \in \{\leq, \geq, =\}$, and a and b are numerical constants discovered during the analysis. The abstract operators of non-relational domains have typically linear worst-case cost, which makes them attractive. For instance, the Interval domain is used as a base domain in most static analyzers based on abstract interpretation, and combined with other domains in reduced products [49, 50] to contribute to the inference of precise invariants. The Congruence domain is used in static analyzers of C programs such as ASTRÉE [25], Frama-C [58] and Mopsa [97] to check that data structures are well-aligned on word boundaries in computer memory.

It is sufficient, in the case non-relational domains, to abstract the set of values of individual variables in each memory state separately, disregarding any relational information. Such a choice is an abstraction *per se*, called Cartesian abstraction.

Cartesian abstraction. We abstract the domain of memory-only states $\hat{\mathcal{D}} = \mathcal{P}(\mathcal{V} \to \mathbb{Z})$ further, into the domain $\tilde{\mathcal{D}} \triangleq \mathcal{V} \to \mathcal{P}(\mathbb{Z}) \cup \{\, \tilde{\bot} \,\}$, where $\tilde{\bot} \notin \mathcal{V} \to \mathcal{P}(\mathbb{Z})$ is the new least element of $(\tilde{\mathcal{D}}, \tilde{\subseteq})$ such that $\tilde{\subseteq} \triangleq \dot{\subseteq} \cup \bigcup_{X \in \tilde{\mathcal{D}}} \{\, (\tilde{\bot}, X) \,\}$. Abstract elements describe the sets of reachable values of individual program variables. This abstraction enjoys a Galois connection, as shown on Fig. 2.17. The semantic transfer functions resulting from the best abstraction are still uncomputable, because $\mathcal{P}(\mathbb{Z})$ contains infinite elements. The principle of interval analysis is to abstract the sets of integers representing the values of individual variables into intervals representing their ranges, to finally achieve computability.

Interval Domain. We start by abstracting sets of integers into intervals, which we will lift to abstract memory states in a second step. The definition of the Interval poset $(\mathcal{I}, \sqsubseteq_{\mathcal{I}})$ is shown on Fig. 2.19. Non-$\bot_{\mathcal{I}}$ abstract elements of $\mathcal{I}$ are defined by an ordered pair of integer or infinite bounds, extending the usual order $\leq$ on $\mathbb{Z}$ to

$$(\mathcal{P}(\mathbb{Z}), \subseteq) \underset{\alpha_{\mathcal{I}}}{\overset{\gamma_{\mathcal{I}}}{\leftrightarrows}} (\mathcal{I}, \sqsubseteq_{\mathcal{I}})$$

$$\alpha_{\mathcal{I}}(X) \triangleq \begin{cases} \bot_{\mathcal{I}} & \textbf{if } X = \emptyset \\ [\min X, \max X] & \textbf{otherwise} \end{cases}$$

$$\gamma_{\mathcal{I}}(i) \triangleq \begin{cases} \emptyset & \textbf{if } i = \bot_{\mathcal{I}} \\ \{\, x \mid l \leq x \leq u \wedge i = [l, u] \,\} & \textbf{otherwise} \end{cases}$$

Figure 2.18: Interval abstraction

$$\begin{array}{rcl} \mathcal{I} & \triangleq & \{\, [l, u] \mid l \in \mathbb{Z} \cup \{\, -\infty \,\}, u \in \mathbb{Z} \cup \{\, +\infty \,\}, l \leq u \,\} \quad \cup \quad \{\, \bot_I \,\} \\ \sqsubseteq_{\mathcal{I}} & \triangleq & \{\, ([l_1, u_1], [l_2, u_2]) \in (\mathcal{I} \setminus \{\, \bot_{\mathcal{I}} \,\})^2 \mid l_2 \leq l_1 \wedge u_1 \leq u_2 \,\} \quad \cup \quad \bigcup_{i \in \mathcal{I}} \{\, (\bot_{\mathcal{I}}, i) \,\} \end{array}$$

Figure 2.19: The Interval Poset

$$[a, b] \sqcup_{\mathcal{I}} [c, d] \triangleq [\min(a, c), \max(b, d)]$$
$$i \sqcup_{\mathcal{I}} \bot_{\mathcal{I}} \triangleq i \triangleq \bot_{\mathcal{I}} \sqcup_{\mathcal{I}} i$$
$$[a, b] \sqcap_{\mathcal{I}} [c, d] \triangleq \begin{cases} [\max(a, c), \min(b, d)] & \textbf{if } \max(a, c) \leq \min(b, d) \\ \bot_{\mathcal{I}} & \textbf{otherwise} \end{cases}$$
$$i \sqcap_{\mathcal{I}} \bot_{\mathcal{I}} \triangleq \bot_{\mathcal{I}} \triangleq \bot_{\mathcal{I}} \sqcap_{\mathcal{I}} i$$

Figure 2.20: Interval lattice operators

infinities. This representation is obviously representable in computer memory: at most two numbers must be represented. The order $\sqsubseteq_{\mathcal{I}}$ is also computable: it compares the bounds of intervals. Fig. 2.18 shows the relation between intervals and sets of integers: the Interval abstraction enjoys a Galois embedding with the complete powerset lattice of integers $(\mathcal{P}(\mathbb{Z}), \subseteq, \cup, \cap, \emptyset, \mathbb{Z})$. The lattice operators $\sqcup_{\mathcal{I}}$ and $\sqcap_{\mathcal{I}}$ can be derived as best abstractions of $\cup$ and $\cap$, as shown on Fig. 2.20. We can derive in a similar way the best abstractions of (arithmetic) operators over $\mathbb{Z}$. We only show the cases of operators $+_{\mathcal{I}}$, $-_{\mathcal{I}}$ and $\times_{\mathcal{I}}$ on Fig. 2.21 for conciseness.

The Interval domain $(\mathcal{I}, \sqsubseteq_{\mathcal{I}}, \sqcup_{\mathcal{I}}, \sqcap_{\mathcal{I}}, \bot_{\mathcal{I}}, [-\infty, +\infty])$ is a complete lattice that features infinite chains, as illustrated by Fig. 2.22. A widening is therefore required to enforce the convergence of abstract iterations. We could use the generic naive widening introduced in Example 22, but this would lead to very imprecise analyses of loops in practice. Instead, real-world analyzers such as ASTRÉE rely on widenings with thresholds. Let

$$\forall i \in \mathcal{I} : \bot_{\mathcal{I}} +_{\mathcal{I}} i \triangleq \bot_{\mathcal{I}} \triangleq i +_{\mathcal{I}} \bot_{\mathcal{I}} \qquad [a,b] +_{\mathcal{I}} [c,d] \triangleq [a+c, b+d]$$
$$\forall i \in \mathcal{I} : \bot_{\mathcal{I}} -_{\mathcal{I}} i \triangleq \bot_{\mathcal{I}} \triangleq i -_{\mathcal{I}} \bot_{\mathcal{I}} \qquad [a,b] -_{\mathcal{I}} [c,d] \triangleq [a-d, b-c]$$
$$\forall i \in \mathcal{I} : \bot_{\mathcal{I}} \times_{\mathcal{I}} i \triangleq \bot_{\mathcal{I}} \triangleq i \times_{\mathcal{I}} \bot_{\mathcal{I}} \qquad [a,b] \times_{\mathcal{I}} [c,d] \triangleq [\min T, \max T]$$
$$\text{where } T = \{\, ac, ad, bc, bd \,\}$$

Figure 2.21: Interval abstraction of arithmetic operators

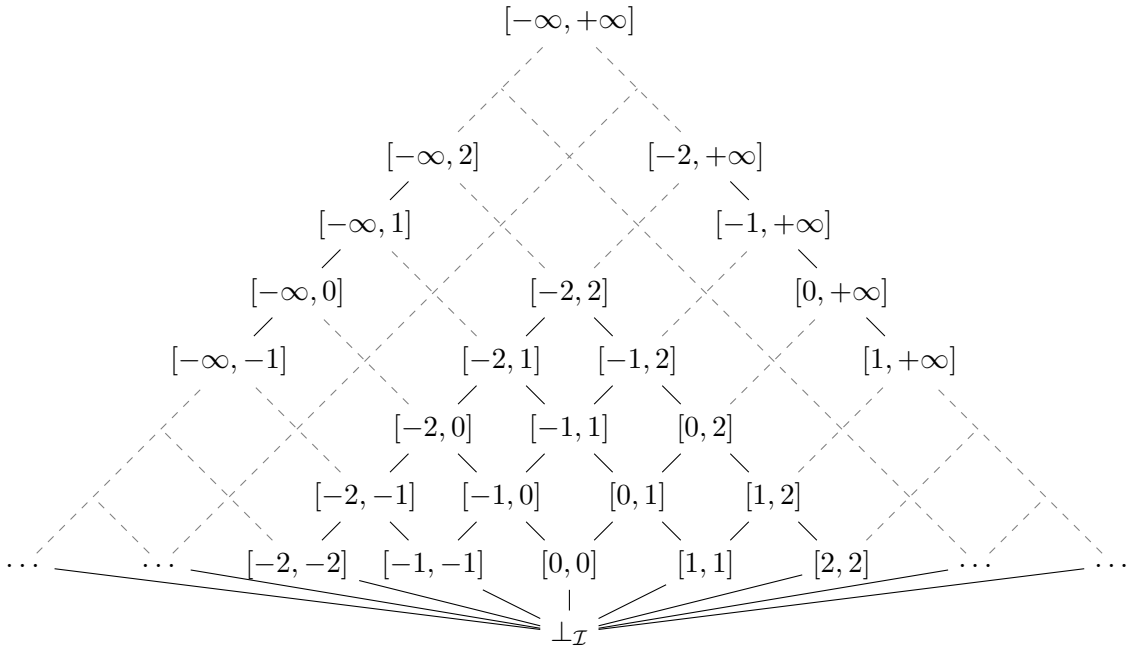


Figure 2.22: Hasse diagram of the Interval poset $(\mathcal{I}, \sqsubseteq_{\mathcal{I}})$

$T \in \mathcal{P}(\mathbb{Z})$ be a finite set of bounds (thresholds). The widening operator ∇_I^T is defined as $[a,b]\nabla_I^T[c,d] \triangleq [x,y]$ where:

- $x = a$ if $a \leq c$, otherwise $x = \max\{\, n \in T \cup \{\, -\infty, +\infty \,\} \mid n \leq c \,\}$;
- $y = b$ if $b \geq d$, otherwise $y = \min\{\, n \in T \cup \{\, -\infty, +\infty \,\} \mid n \geq d \,\}$.

Interval based abstract program semantics. We are now ready to define an interval-based abstract domain $\mathcal{D}^\sharp$ that can be used to parameterize the generic abstract program semantics of Fig. 2.16. Leveraging Property 8, we derive $\mathcal{D}^\sharp$ by coalescent pointwise lifting:

$$\mathcal{D}^\sharp \triangleq (\mathcal{V} \to (\mathcal{I} \setminus \{\, \bot_I \,\})) \cup \{\, \bot^\sharp \,\}$$

$$X^\sharp \sqsubseteq^\sharp Y^\sharp \iff (X^\sharp = \bot^\sharp) \vee (X^\sharp, Y^\sharp \neq \bot^\sharp \wedge \forall V \in \mathcal{V} : X^\sharp(V) \sqsubseteq_{\mathcal{I}} Y^\sharp(V))$$

$$\bar{\gamma}_{\mathcal{I}}(X^\sharp) = \begin{cases} \tilde{\bot} & \textbf{if } X^\sharp = \bot^\sharp \\ \gamma_{\mathcal{I}} \circ X^\sharp & \textbf{otherwise} \end{cases}$$

$$\bar{\alpha}_{\mathcal{I}}(\tilde{X}) = \begin{cases} \bot^\sharp & \textbf{if } \tilde{X} = \tilde{\bot} \\ \alpha_{\mathcal{I}} \circ \tilde{X} & \textbf{otherwise} \end{cases}$$

$$\top^\sharp(V) = [-\infty, +\infty]$$

$$X^\sharp \cup^\sharp Y^\sharp = \begin{cases} Y^\sharp & \textbf{if } X^\sharp = \bot^\sharp \\ X^\sharp & \textbf{if } Y^\sharp = \bot^\sharp \\ V \mapsto X^\sharp(V) \sqcup_{\mathcal{I}} Y^\sharp(V) & \textbf{otherwise} \end{cases}$$

$$X^\sharp \nabla_T^\sharp Y^\sharp = \begin{cases} Y^\sharp & \textbf{if } X^\sharp = \bot^\sharp \\ X^\sharp & \textbf{if } Y^\sharp = \bot^\sharp \\ V \mapsto X^\sharp(V) \nabla_{\mathcal{I}}^T Y^\sharp(V) & \textbf{otherwise} \end{cases}$$

$$X^\sharp \cap^\sharp Y^\sharp = \begin{cases} \bot^\sharp & \textbf{if } X^\sharp = \bot^\sharp \vee Y^\sharp = \bot^\sharp \\ \bot^\sharp & \textbf{if } \exists V \in \mathcal{V} : X^\sharp(V) \sqcap_{\mathcal{I}} Y^\sharp(V) = \bot_{\mathcal{I}}^\sharp \\ V \mapsto X^\sharp(V) \sqcap_{\mathcal{I}} Y^\sharp(V) & \textbf{otherwise} \end{cases}$$

Figure 2.23: Interval abstract operators

such that all elements where $\bot_I$ appears are coalesced into a unique element $\bot^\sharp$. Property 8 guarantees optimal abstraction: $(\tilde{\mathcal{D}}, \tilde{\subseteq}) \underset{\bar{\alpha}_{\mathcal{I}}}{\overset{\bar{\gamma}_{\mathcal{I}}}{\leftrightarrows}} (\mathcal{D}^\sharp, \sqsubseteq^\sharp)$, with $\sqsubseteq^\sharp = \dot{\tilde{\sqsubseteq}}_{\mathcal{I}} \cup \bigcup_{X^\sharp \in \mathcal{D}^\sharp} \{ (\bot^\sharp, X^\sharp) \}$. This allows deriving most abstract operators on $\mathcal{D}^\sharp$ by lifting operators on $\mathcal{I}$ pointwise. Fig. 2.23 shows these operators. They are sound, optimal and computable.

The only missing operators to complete the requirements of Sec. 2.3.1 for a computable abstraction are the transfer functions for assignments and tests $\mathbb{S}^\sharp[\![V \leftarrow e]\!]$ and $\mathbb{S}^\sharp[\![e_1 \bowtie e_2 ?]\!]$. The standard practice is to propose algorithms that define the semantics, rather than deriving optimal abstract operators through $\alpha_{\mathcal{I}}$. The soundness of these algorithms must then be proved using $\gamma_{\mathcal{I}}$. We will not do it here, as such algorithms and proofs are standard. A standard abstract semantics for assignments is:

$$\mathbb{S}^\sharp[\![V \leftarrow e]\!] X^\sharp \triangleq \begin{cases} \bot^\sharp & \textbf{if } X = \bot^\sharp \vee \mathbb{E}^\sharp[\![e]\!] X^\sharp = \bot_{\mathcal{I}} \\ X^\sharp[V \mapsto \mathbb{E}^\sharp[\![e]\!]] & \textbf{otherwise} \end{cases}$$

where the abstract semantics $\mathbb{E}^\sharp[\![e]\!]$ of expression $e \in \mathit{expr}$ is evaluated by induction on the syntax, propagating the abstract, interval-based, representation for sets of values. The definition of $\mathbb{E}^\sharp[\![e]\!]$ is shown on Fig. 2.24. The abstract semantics for tests is shown on Fig. 2.25: it handles precisely simple cases, safely defaulting to the identity for other cases. Note that much more precise abstract semantics can be defined for tests. Standard

$$
\begin{array}{ll}
\mathbb{E}^\sharp[\![\, V \in \mathcal{V} \,]\!] X^\sharp & \triangleq X^\sharp(V) \\
\mathbb{E}^\sharp[\![\, c \in \mathbb{Z} \,]\!] X^\sharp & \triangleq [c, c] \\
\mathbb{E}^\sharp[\![\, \mathbf{rand}(a, b) \,]\!] X^\sharp & \triangleq [a, b] \\
\mathbb{E}^\sharp[\![\, - e \,]\!] X^\sharp & \triangleq -^\sharp\, \mathbb{E}^\sharp[\![\, e \,]\!] X^\sharp \\
\mathbb{E}^\sharp[\![\, e_1 \,+\, e_2 \,]\!] X^\sharp & \triangleq \mathbb{E}^\sharp[\![\, e_1 \,]\!] X^\sharp \;+_{\mathcal{I}}\; \mathbb{E}^\sharp[\![\, e_2 \,]\!] X^\sharp \\
\mathbb{E}^\sharp[\![\, e_1 \,-\, e_2 \,]\!] X^\sharp & \triangleq \mathbb{E}^\sharp[\![\, e_1 \,]\!] X^\sharp \;-_{\mathcal{I}}\; \mathbb{E}^\sharp[\![\, e_2 \,]\!] X^\sharp \\
\mathbb{E}^\sharp[\![\, e_1 \,\times\, e_2 \,]\!] X^\sharp & \triangleq \mathbb{E}^\sharp[\![\, e_1 \,]\!] X^\sharp \;\times_{\mathcal{I}}\; \mathbb{E}^\sharp[\![\, e_2 \,]\!] X^\sharp
\end{array}
$$

Figure 2.24: Abstract semantics of expressions

$$
\mathbb{S}^\sharp[\![\, V \leq n \;?\,]\!] X^\sharp \triangleq \begin{cases} X^\sharp[V \mapsto [a, \min\{\, b, n \,\}]] & \textbf{if } \; a \leq n \\ \bot^\sharp & \textbf{otherwise} \end{cases}
$$

$$
\mathbb{S}^\sharp[\![\, V \leq W \;?\,]\!] X^\sharp \triangleq \begin{cases} X^\sharp[V \mapsto [a, \min\{\, b, d \,\}], W \mapsto [\max\{\, a, c \,\}, d]] & \textbf{if } \; a \leq d \\ \bot^\sharp & \textbf{otherwise} \end{cases}
$$

$$
\mathbb{S}^\sharp[\![\, e_1 \bowtie e_2 \;?\,]\!] X^\sharp \triangleq X^\sharp \;\text{ in all other cases.}
$$

where $[a, b] = X^\sharp(V)$ and $[c, d] = X^\sharp(W)$

Figure 2.25: Abstract semantics of tests

versions rely on bottom-up and top-down traversals of the abstract syntax tree of the condition $e_1 \bowtie e_2$, known the HC4 [23] algorithm in the constraint solving community. A presentation in the abstract interpretation framework can be found in [132, Sec. 4.6].

Interval analysis. In the previous sections, starting with Example 15 of Sec. 2.2.3, we have used a gradual abstraction scheme to abstract the uncomputable semantics $\mathbb{S}$ of Nimp programs into a computable numerical abstract semantics $\mathbb{S}^\sharp$. Abstraction steps can be summarized as a sequence of Galois connections:

$$
(\mathcal{D}, \subseteq) \underset{\hat{\alpha}}{\overset{\hat{\gamma}}{\leftrightarrows}} (\hat{\mathcal{D}}, \subseteq) \underset{\alpha_C}{\overset{\gamma_C}{\leftrightarrows}} (\tilde{\mathcal{D}}, \tilde{\subseteq}) \underset{\bar{\alpha}_{\mathcal{I}}}{\overset{\bar{\gamma}_{\mathcal{I}}}{\leftrightarrows}} (\mathcal{D}^\sharp, \sqsubseteq^\sharp)
$$

The two first abstraction steps are done directly, while the last one results from the coalescent pointwise lifting of the Galois connection $(\mathcal{P}(\mathbb{Z}), \subseteq) \underset{\alpha_{\mathcal{I}}}{\overset{\gamma_{\mathcal{I}}}{\leftrightarrows}} (\mathcal{I}, \sqsubseteq_{\mathcal{I}})$.

Our construction on now complete: $\mathbb{S}^\sharp$ is a computable abstract semantics based on the interval domain, which can be implemented as part of a static analyzer, and used for the analysis of real, if simple, Nimp programs.

Example 24. Fig. 2.26 shows a simple Nimp program that reads an input in the range $[0, 5]$, stores it into variable X, and increments X by steps of 2 until it is larger than 42. An assertion line 5 expresses the expected property that $X < 50$ after the loop. Analyzing this program with our semantics $\mathbb{S}^\sharp$, we enter the loop with the abstract

```
1 :  X ← input(0, 5);
2 :  while (X < 42) do
3 :    X ← X + 2
4 :  done;
5 :  assert(X < 50)
```

Figure 2.26: Simple loop

invariant $X_2^\sharp = X \mapsto [0, 5]$, and iterate $X^\sharp \mapsto X^\sharp \, \nabla_T^\sharp \, (X_2^\sharp \cup^\sharp \mathbb{S}^\sharp[\![X \leftarrow X + 2]\!] \circ \mathbb{S}^\sharp[\![X < 42]\!] X^\sharp)$ from $\bot^\sharp$. Equivalently in the interval domain, we iterate $I \mapsto I \, \nabla_{\mathcal{I}}^T \, (I_2 \sqcup_I (I \sqcap_{\mathcal{I}} [-\infty, 41]) +_{\mathcal{I}} [2, 2])$ from $I_2 = [0, 5]$. Assuming $T = \emptyset$ (no widening thresholds), we reach a fixpoint $I^* = [0, +\infty]$ in 2 iterations. We obtain the invariant $X \in [42, +\infty]$ line 5, which is too coarse to prove the assertion. In contrast, if we add a widening threshold $T = \{ 45 \}$, we reach the fixpoint $I^* = [0, 45]$ in 2 iterations. We obtain the invariant $X \in [42, 45]$ line 5, which is precise enough to prove the assertion. Note that the most precise invariant is $[42, 43]$. The best invariant we can obtain with our interval abstraction alone is $[0, 43]$, which is computed if 43 is a widening threshold ($43 \in T$).

The precision of the Interval abstraction can be improved by a number of techniques, such as loop unrolling, delayed widening and narrowing. We do not present these techniques in the current background chapter, and refer the interested reader to [132]. The Interval domain is widely used as a base domain in most static analyzers based on abstract interpretation, because variable bounds are needed in most static analyses, and because the linear cost of its abstract operators makes it attractive. The Interval domain infers coarse invariants when used in isolation. It is typically combined with other domains in reduced products [49, 50] to infer precise invariants.

Linear relational domains

In contrast to the Interval domain and other non-relational domain presented earlier, a variety of domains have been developed to infer relational numerical invariants, such as algebraic relations between program variables. Such domains are called relational domains. The majority focus on linear (or affine) invariants, as such invariants are of critical importance for the analysis of general-purpose software. Multiple domains have been proposed to infer linear relations between program variables. The most expressive one is the the Polyhedra domain [51]. More restricted domains include zones [135], octagons [128], linear equalities [102] pentagons [119], parallelotopes [10], octahedra [40], two variables per inequality [161], 4-octahedron [149, 150], Logahedra [86] and gauges [173]. They all consist in considering specific templates of polyhedra.

Polyhedra. The Polyhedra domain is one of the most used relational abstract domains. It expresses general linear inequalities of the form:

$$\bigwedge_{j=1}^{m} \sum_{i=1}^{n} a_{ij}\mathtt{x}_i \geq b_j$$

where $\mathtt{x}_i$ are program variables, $n = |\mathcal{V}|$ is the number of program variables, m is the number of inequalities, and a_{ij} and b_i are numerical constants discovered by the analysis. The set of abstract elements is subject to a double representation: a constraint-based representation, and a generator-based one. Some operators are indeed more efficient on the former, while others are more efficient on the latter. The domain uses Chernikova's algorithm [35, 116] to convert between the two representations. The constraint representation relies directly on the matrix $(a_{ij})_{(i,j)\in[1,n]\times[1,m]}$ and vector $(b_j)_{j\in[1,m]}$. The generator representation is a set of vectors representing vertices or rays. Every point inside a bounded polyhedron is an affine combination of vertices. For an unbounded polyhedron, combinations of rays defining the directions in which it is unbounded are added to the combination of vertices. The meaning of each representation is defined by a concretization function, as there is no Galois connection between $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$ and the polyhedra domain. Recall indeed Example 17: not all sets of numbers enjoy a best abstraction in this domain. Assignments and tests of linear expressions are represented exactly (no loss of precision). Non-linear expressions are handled by default, unprecise transfer functions. The abstract join computes the convex hull of two polyhedra. Implementations of the Polyhedra domain can be found in publicly available abstract domain libraries such as Apron [93], PPL [14] and ELINA [163]. Abstract operators exhibit exponential cost in practice. Some implementations [162] rely on a constraint-only representation, as an important part of the cost of operators in the classic double description method is due to the necessary conversions between the two representations.

Octagons. Polyhedra allow precise analyses of linear invariants, but do not scale to large programs. On the contrary, Intervals scale, but allow very limited precision. The quest for an intermediate option motivated the introduction of so-called weakly relational domains [126]. Such domains express restricted linear invariants, relying on operators with polynomial cost. The Octagon domain [128] is one of the most widely used weakly relational domains. It expresses linear inequalities of the form

$$\bigwedge_{(i,j)\in[1,n]\times[1,m]} \pm\mathtt{x}_j \pm \mathtt{x}_i \leq c_{ij}$$
$$\bigwedge_{i\in[1,n]} a_i \leq \mathtt{x}_i \leq b_i$$

where $\mathtt{x}_i$ are program variables, $n = |\mathcal{V}|$ is the number of program variables, m is the number of two-variable inequalities, and a_i, b_i and c_{ij} are numerical constants discovered by the analysis. The abstract representation relies on differences bound matrices (DBM, or equivalently, potential graphs). The domain enjoys a Galois connection (for integer

or real-valued programs). The same concrete elements may have several abstract representations as DBM, hence operators rely on strong normalization. Tests of the form $\pm\mathtt{x}_j \pm \mathtt{x}_i \leq c$ and assignments of the form $\mathtt{x}_j \leftarrow \pm\mathtt{x}_i + c$ are abstracted exactly. Abstract operators have quadratic space (resp. cubic time) complexity in the worst case, which is much better than polyhedra. Yet this is still costly for programs with a large number of variables, hence variable packing [25, Sec. III.H.5] strategies are used in practice. A widely used implementation of Octagons ships with the Apron [93] abstract domain library. A related industrial-strength implementation ships with the ASTRÉE [25] static analyzer. It is important, for instance, to analyze precisely some tests and some loops that are frequently used as part of basic operators of control-command programs. A certified implementation is available in VERASCO [95, 96].

Non-linear relational domains

Domains expressing non-linear invariants are less widely used. Abstract domains expressing general polynomial inequalities are currently not scalable [15, 159]. In contrast, some specialized domains are able to infer efficiently non-linear invariants that are necessary to analyze some domain-specific programs, such as programs modeling or controlling physical systems. Non-linear numerical abstract domains include ellipses [73, 160, 151] for analyzing digital filters [155], and exponentials [74], for bounding the drift of floating-point numbers due to the slow accumulation of rand-off errors over time. For instance, ellipses and exponentials were implemented into the ASTRÉE [25] static analyzer to analyze precisely control-command programs embedded in some Airbus fly-by-wire control systems. ASTRÉE implements a large set of relational domains, and not only domain-specific ones. This set includes intervals, congruences, octagons, ellipses, exponentials and gauges. In constrast, polyhedra are not used because of floating-point implementation shorcomings [46, Sec. 53.2.11]. Geometric representations of the concretizations of the invariants inferred by intervals, congruences, octagons, ellipses, exponentials and polyhedra are shown on Fig. 2.27 (adapted from [25]).

2.4 Conclusion

In this chapter, we have introduced a standard approach to static analysis by abstract interpretation that will be used extensively in the rest of the thesis. In particular, we have demonstrated a classic construction of a static analysis of numerical programs by induction on the syntax, that is parametric in the choice of a numerical abstract domain. We have illustrated this construction on the simple imperative language NIMP.

We will follow the same approach in Chapter 3, where we turn to the joint analysis of pairs of NIMP programs, with a view to inferring their functional equivalence. This goal requires comparing their semantics at an abstract level. We will thus design a joint concrete semantics for pairs of NIMP programs, defined by induction on the syntax of so-called double programs. We will then abstract this joint semantics in numerical domains, and introduce a novel numerical domain to bound the differences between the values of

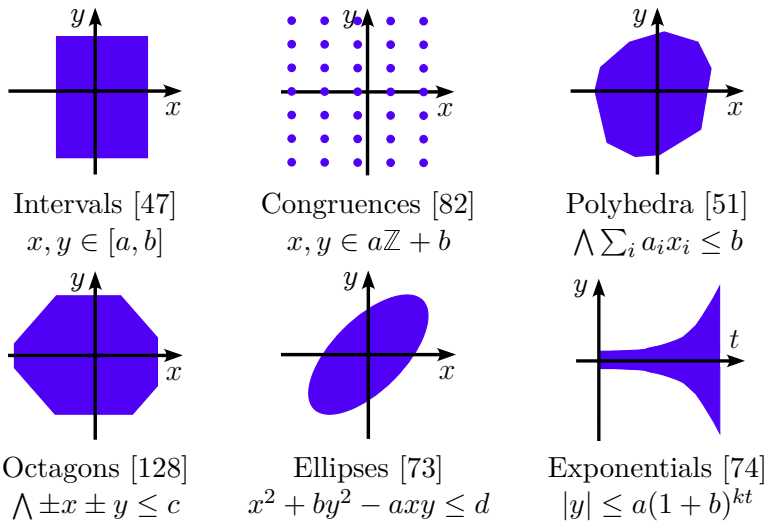


Figure 2.27: Examples of numerical abstract domains [25]

the variables in the two programs. We will introduce another novel numerical domain in Chapter 7. This domain infers relations between individual bytes of C variables that are useful for inferring the endian portability of C programs.

Chapter 3

Patch Analysis

In this chapter, we present a static analysis for software patches. Given two syntactically close versions of a program, our analysis can infer a semantic difference, and prove that both versions compute the same outputs when run on the same inputs.

3.1 Motivation

The main application of this analysis is regression verification [79]: prove that a program change does not add any undesirable behavior. Recall, for instance, the motivating Example 1 from Sec. 1.1.4. This example features a change in a library used by the POSIX `rm` command, which we reproduce on Fig. 3.1. The `cache_fstatat` function is designed to cache the results of the POSIX `fstatat` function. In revision v6.10 of Coreutils, the former used the `st_size` field of the stat structure `*st` to cache the `errno` error value returned by the latter. The implementation worked for operating systems featuring only positive `errno` values. The patch displayed on Fig. 1.5 aims at accommodating operating systems allowing for negative `errno` values, such as BeOS [1] and Haiku [2].

On this example, regression verification amounts to proving that the behavior of the main function of the library is unchanged on systems with only positive error values. This is, indeed, validated by our analysis. The analyzed source code includes a stub variable for `errno`, and stub code for the `fstatat` function. The stub for `fstatat` updates `errno` with a non-deterministic value, ranging over positive integers. Note that a separate analysis of the `cache_fstatat` function, as opposed to an analysis of the whole library, makes it necessary to model its possible execution environments with an unbounded loop, calling `cache_fstatat` an arbitrary number of times, with parameters taken from an arbitrary sequence of file names and stat structures. This unbounded sequence is modeled, in practice, using an unbounded number of reads from an input stream. Fig. 3.2 shows an example for n files, where n may be unbounded.

More generally, we are interested in analyzing patches of programs reading an unbounded number of input values, e.g. programs reading from file or I/O streams, and embedded reactive software with internal state, which no related work addresses. Our goal is to prove that the original and patched versions of such programs compute equal

```
172 172   /* Like fstatat, but cache the result.  If ST->st_size is -1, the
173 173      status has not been gotten yet.  If less than -1, fstatat failed
174     -    with errno == -1 - ST->st_size.  Otherwise, the status has already
    174 +    with errno == ST->st_ino.  Otherwise, the status has already
175 175      been gotten, so return 0.  */
176 176   static int
177 177   cache_fstatat (int fd, char const *file, struct stat *st, int flag)
178 178   {
179 179     if (st->st_size == -1 && fstatat (fd, file, st, flag) != 0)
180     -     st->st_size = -1 - errno;
    180 +     {
    181 +       st->st_size = -2;
    182 +       st->st_ino = errno;
    183 +     }
181 184     if (0 <= st->st_size)
182 185       return 0;
183     -   errno = -1 - st->st_size;
    186 +   errno = (int) st->st_ino;
184 187     return -1;
185 188   }
```

Figure 3.1: Patch on `remove.c` of Coreutils (between v6.10 and v6.11)

```
for (c=0; c<n; c++) cache_stat_init (&file[c].st);

while ((c=getchar()) >= 0 && c < n)
  r = cache_fstatat (AT_FDCWD, file[c].name, &file[c].st, AT_SYMLINK_NOFOLLOW);
```

Figure 3.2: Execution environments for `cache_fstatat`

outputs, when run with the same sequence of inputs. Like in Sec. 2.1.2, we therefore model streams directly in the concrete semantics on which our analysis is based.

3.2 Running example

Let us sketch our approach to the analysis of semantic differences between two syntactically similar programs P_1 and P_2. We are interested in proving that P_1 and P_2 compute equal outputs when run on equal inputs.

Example 25 (running example)**.** As a running example, let us start with two syntactically close versions of the `Unchloop` benchmark, taken from [172], and displayed on Fig. 3.3(c). Figs. 3.3(a) and 3.3(b) show two versions of the `Unchloop` program, where lines 3 and 9 differ. Let P_1 denote the first (or left) version, and P_2 the second (or right) version. We call *double program*, and denote by P shown in Fig. 3.3(c), a joint syntactic representation of P_1 and P_2 that highlights their common parts and their differences. P_1 and P_2 are called simple programs, and referred to as the left (or first) and right (or

```
1 :  a ← input(−1000, 1000);
2 :  b ← input(−1000, 1000);
3 :  c ← 1;
4 :  i ← 0;
5 :  while (i < a) {
6 :    c ← c + b;
7 :    i ← i + 1;
8 :  }
9 :  r ← c;
10 : output(r);
```

(a) Left version P_1

```
1 :  a ← input(−1000, 1000);
2 :  b ← input(−1000, 1000);
3 :  c ← 0;
4 :  i ← 0;
5 :  while (i < a) {
6 :    c ← c + b;
7 :    i ← i + 1;
8 :  }
9 :  r ← c + 1;
10 : output(r);
```

(b) Right version P_2

```
1 :  a ← input(−1000, 1000);
2 :  b ← input(−1000, 1000);
3 :  c ← 1 ‖ 0;
4 :  i ← 0;
5 :  while (i < a) {
6 :    c ← c + b;
7 :    i ← i + 1;
8 :  }
9 :  r ← c ‖ c + 1;
10 : output(r);
```

(c) Double program P

Figure 3.3: Two versions of the `Unchloop` example

second) versions of P. The $\|$ symbol is used to represent syntactic differences in the syntax of P. It is available at expression, condition, and statement levels in our syntax for double programs. For instance at line 3, $c \leftarrow 1 \,\|\, 0$ means $c \leftarrow 1$ for P_1, and $c \leftarrow 0$ for P_2. In contrast, line 4 means $i \leftarrow 0$ for both P_1 and P_2.

Let us describe this running example. Both versions P_1 and P_2 read inputs in the range $[-1000, 1000]$ into `a` and `b` at lines 1 and 2. At line 3, the counter `c` is being initialised with value `1` for program P_1, and value `0` for program P_2. Then, both programs add `a` times the value of `b` to `c` in a loop. Finally, they both store the result into `r` at line 9: `c` for P_1, `c+1` for P_2. Finally, both program versions write the value of `r` to some output stream at line 10.

The property we would like to check is the following: if both program versions P_1 and P_2 read the same input values at lines 1 and 2, and if P_1 and P_2 both reach line 10, then P_1 and P_2 should both write the same output value.

More generally, the semantics of P is parameterized by a (possibly infinite) sequence of input values, and we wish to prove that, given the same sequence of input values, P_1 and P_2 write the same sequence of output values. Following the standard approach

$$
\begin{array}{lcll}
\textit{dstat} & ::= & \textit{stat} & \\
& | & \textit{stat} \parallel \textit{stat} & \\
& | & V \leftarrow \textit{dexpr} & V \in \mathcal{V} \\
& | & \textit{dstat}; \textit{dstat} & \\
& | & \textbf{if } \textit{dcond} \textbf{ then } \textit{dstat} \textbf{ else } \textit{dstat} & \\
& | & \textbf{while } \textit{dcond} \textbf{ do } \textit{dstat} &
\end{array}
$$

Figure 3.4: Statements of double programs

to abstract interpretation [48], we develop a concrete collecting semantics for a toy imperative language for double programs, which we call NIMP$_2$. NIMP$_2$ is constructed as an extension of NIMP in the current chapter. Our approach will be further developed in Chapters 5 and 6, to enable the analysis of real-world C programs. In this chapter, we assume the double program P given. We will present in Chapter 4 an algorithm for automating its construction from P_1 and P_2.

The rest of the chapter is organized as follows. We first describe the syntax and semantics of double programs reading from infinite input streams in Sec. 3.3 and 3.4. Then, we abstract this semantics in numerical domains in Sec. 3.5. We present an experimental evaluation in Sec. 3.6 and related works in Sec. 3.7. Finally, Sec. 3.8 concludes.

3.3 Syntax

The syntax of double statements *dstat* is shown in Fig. 3.4. It is built on top of double expressions *dexpr* and double conditions *dcond*, defined in Fig. 3.5. The distinctive trait of NIMP$_2$ syntax is the $\parallel$ operator, which may occur anywhere in the parse tree to denote syntactic differences between then left and right versions of a double program. However, $\parallel$ operators cannot be nested: a double program only describes a pair of programs.

Remark 17 ($\parallel$ operators in expressions and conditions). $\parallel$ operators can occur in expressions and Boolean conditions, but only at the top level of their parse tree.

Given a double program P, we may extract its left (resp. right) version $P_1 = \pi_1(P)$ (resp. $P_2 = \pi_2(P)$), using the π_1 (resp. π_2) version extraction function, keeping only the left (resp. right) side of $\parallel$ symbols. The definition for π_k is given on Fig. 3.6 and 3.7, by induction on the syntax of double statements, expressions and conditions. For instance, $\pi_1(x \leftarrow 1 \parallel y \leftarrow 0) = x \leftarrow 1$, and $\pi_2(x \leftarrow 1 \parallel y \leftarrow 0) = y \leftarrow 0$, while $\pi_1(z \leftarrow 0) = z \leftarrow 0 = \pi_2(z \leftarrow 0)$.

Simple programs P_1 and P_2 enjoy the NIMP syntax for simple imperative programs, presented in Fig. 2.1 of Sec. 2.1.1.

$$\begin{array}{lcl} \mathit{dexpr} & ::= & \mathit{expr} \\ & | & \mathit{expr} \parallel \mathit{expr} \end{array} \qquad \begin{array}{lcl} \mathit{dcond} & ::= & \mathit{cond} \\ & | & \mathit{cond} \parallel \mathit{cond} \end{array}$$

Figure 3.5: Double expressions and conditions of double programs

$$\underline{\pi_k \in \mathit{dstat} \to \mathit{stat}} \qquad k \in \{1, 2\}$$

$$\begin{array}{lcl} \pi_k(s) & = & s \textbf{ if } s \in \mathit{stat} \\ \pi_k(s_1 \parallel s_2) & = & s_k \\ \pi_k(V \leftarrow e) & = & V \leftarrow \pi_k^{\mathit{expr}}(e) \\ \pi_k(s; t) & = & \pi_k(s); \pi_k(t) \\ \pi_k(\textbf{if } c \textbf{ then } s \textbf{ else } t) & = & \textbf{if } \pi_k^{\mathit{cond}}(c) \textbf{ then } \pi_k(s) \textbf{ else } \pi_k(t) \\ \pi_k(\textbf{while } c \textbf{ do } s) & = & \textbf{while } \pi_k^{\mathit{cond}}(c) \textbf{ do } \pi_k(s) \end{array}$$

Figure 3.6: Version extractor for statements of double programs

$$\underline{\pi_k^{\mathit{expr}} \in \mathit{dexpr} \to \mathit{expr}} \qquad \underline{\pi_k^{\mathit{cond}} \in \mathit{dcond} \to \mathit{cond}} \qquad k \in \{1, 2\}$$

$$\begin{array}{lcl} \pi_k^{\mathit{expr}}(e) & = & e \textbf{ if } e \in \mathit{expr} \\ \pi_k^{\mathit{expr}}(e_1 \parallel e_2) & = & e_k \end{array} \qquad \begin{array}{lcl} \pi_k^{\mathit{cond}}(c) & = & c \textbf{ if } c \in \mathit{cond} \\ \pi_k^{\mathit{cond}}(c_1 \parallel c_2) & = & c_k \end{array}$$

Figure 3.7: Version extractor for expressions and conditions

3.4 Concrete semantics

In this section, we define the concrete semantics of the NIMP$_2$ language. To this aim, we first lift the semantics $\mathbb{S}$ of terminating simple statements defined in Sec. 2.1.2 to a semantics for double statements, denoted $\mathbb{D}$. Then, we present the double semantics $\mathbb{P}_2[\![\cdot]\!]$ of NIMP$_2$ double programs, and the program equivalence property of interest. Finally, we describe an extension of NIMP$_2$ designed to express equivalence properties of non-terminating reactive programs running in lockstep.

3.4.1 From simple statements to double statements

A double program represents a pair of simple programs, reading from the same stream of inputs. As simple program versions $P_k = \pi_k(P)$ have concrete states in $\Sigma = (\mathcal{V} \to \mathbb{Z}) \times \mathbb{N} \times \mathbb{R}^\star$, double program P has concrete states in $\mathcal{D} \triangleq \Sigma \times \Sigma$. In addition, $\mathbb{D}$ is parameterized by infinite sequences of values, representing the input stream shared by P_1 and P_2.

Thus the semantics of a double statement $s \in \mathit{dstat}$, denoted $\mathbb{D}[\![s]\!] \in \mathbb{Z}^\omega \to \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$, describes the relation between pre- and post-states of s, which are pairs of states of simple programs, for a given shared sequence of input values. Given an input stream $\iota \in \mathbb{Z}^\omega$ and a set of double program pre-states $X \in \mathcal{P}(\mathcal{D})$, the set of reachable double post-states $\mathbb{D}[\![s]\!]_\iota X$ is shown on Fig. 3.8. It is defined by induction on the syntax, so as to allow for modular, joint analyses of double programs that maintain input-output

$\mathbb{D}[\![\, dstat \,]\!] \in \mathbb{Z}^\omega \to \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

$$
\begin{array}{ll}
\mathbb{D}[\![\, \textbf{skip} \,]\!]_\iota X & \triangleq X \\
\mathbb{D}[\![\, s_1 \parallel s_2 \,]\!]_\iota X & \triangleq \bigcup_{(\sigma_1,\sigma_2)\in X} \{\, (\sigma_1', \sigma_2') \mid \sigma_1' \in \mathbb{S}[\![\, s_1 \,]\!]_\iota \{\, \sigma_1 \,\} \wedge \sigma_2' \in \mathbb{S}[\![\, s_2 \,]\!]_\iota \{\, \sigma_2 \,\} \,\} \\
\mathbb{D}[\![\, V \leftarrow e_1 \parallel e_2 \,]\!] & \triangleq \mathbb{D}[\![\, V \leftarrow e_1 \parallel V \leftarrow e_2 \,]\!] \\
\mathbb{D}[\![\, V \leftarrow e \,]\!] & \triangleq \mathbb{D}[\![\, V \leftarrow e \parallel V \leftarrow e \,]\!] \\
\mathbb{D}[\![\, \textbf{assert}(c) \,]\!] & \triangleq \mathbb{D}[\![\, \textbf{assert}(c) \parallel \textbf{assert}(c) \,]\!] \\
\mathbb{D}[\![\, V \leftarrow \textbf{input}(a,b) \,]\!] & \triangleq \mathbb{D}[\![\, V \leftarrow \textbf{input}(a,b) \parallel V \leftarrow \textbf{input}(a,b) \,]\!] \\
\mathbb{D}[\![\, \textbf{output}(V) \,]\!] & \triangleq \mathbb{D}[\![\, \textbf{output}(V) \parallel \textbf{output}(V) \,]\!] \\
\mathbb{D}[\![\, s_1;\ s_2 \,]\!]_\iota & \triangleq \mathbb{D}[\![\, s_2 \,]\!]_\iota \circ \mathbb{D}[\![\, s_1 \,]\!]_\iota \\
\mathbb{D}[\![\, \textbf{if } c_1 \parallel c_2 \textbf{ then } s \textbf{ else } t \,]\!]_\iota & \triangleq \mathbb{D}[\![\, s \,]\!]_\iota \circ \mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] \\
& \dot\cup\ \mathbb{D}[\![\, \pi_1(s) \parallel \pi_2(t) \,]\!]_\iota \circ \mathbb{F}[\![\, c_1 \parallel \neg c_2 \,]\!] \\
& \dot\cup\ \mathbb{D}[\![\, \pi_1(t) \parallel \pi_2(s) \,]\!]_\iota \circ \mathbb{F}[\![\, \neg c_1 \parallel c_2 \,]\!] \\
& \dot\cup\ \mathbb{D}[\![\, t \,]\!]_\iota \circ \mathbb{F}[\![\, \neg c_1 \parallel \neg c_2 \,]\!] \\
\mathbb{D}[\![\, \textbf{if } c \textbf{ then } s \textbf{ else } t \,]\!] & \triangleq \mathbb{D}[\![\, \textbf{if } c \parallel c \textbf{ then } s \textbf{ else } t \,]\!] \\
\mathbb{D}[\![\, \textbf{while } c_1 \parallel c_2 \textbf{ do } s \,]\!]_\iota X & \triangleq \mathbb{F}[\![\, \neg c_1 \parallel \neg c_2 \,]\!](\mathrm{lfp}\ H_\iota^X) \\
\mathbb{D}[\![\, \textbf{while } c \textbf{ do } s \,]\!] & \triangleq \mathbb{D}[\![\, \textbf{while } c \parallel c \textbf{ do } s \,]\!]
\end{array}
$$

$$
\begin{array}{lll}
\text{where} & \mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] X & \triangleq \{\, ((\rho_1, n_1, o_1), (\rho_2, n_2, o_2)) \in X \mid \mathrm{true} \in \mathbb{C}[\![\, c_1 \,]\!]\rho_1 \cap \mathbb{C}[\![\, c_2 \,]\!]\rho_2 \,\} \\
\text{and} & H_\iota^X(Y) & \triangleq X \\
& & \cup\ \mathbb{D}[\![\, s \,]\!]_\iota \circ \mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] Y \\
& & \cup\ \mathbb{D}[\![\, \pi_1(s) \parallel \textbf{skip} \,]\!]_\iota \circ \mathbb{F}[\![\, c_1 \parallel \neg c_2 \,]\!] Y \\
& & \cup\ \mathbb{D}[\![\, \textbf{skip} \parallel \pi_2(s) \,]\!]_\iota \circ \mathbb{F}[\![\, \neg c_1 \parallel c_2 \,]\!] Y
\end{array}
$$

Figure 3.8: Denotational concrete semantics of double programs

relations on the variables. Note that $\mathbb{D}$ is parametric in $\mathbb{S}$.

The semantics for the empty program is the identity function. The semantics $\mathbb{D}[\![\, s_1 \parallel s_2 \,]\!]$ for the composition of two syntactically different statements reverts to the pairing of the simple program semantics of individual simple statements s_1 and s_2. Note that $\mathbb{D}[\![\, s_1 \parallel s_2 \,]\!]_\iota = \mathbb{D}[\![\, \textbf{skip} \parallel s_2 \,]\!]_\iota \circ \mathbb{D}[\![\, s_1 \parallel \textbf{skip} \,]\!]_\iota$. The semantics for atomic statements, such as assignments, assert, input and output statements, are defined using this construct. In particular, the semantics defines the assignments of double expressions $V \leftarrow e_1 \parallel e_2$ (different expressions to the same variable) as shortands for $V \leftarrow e_1 \parallel V \leftarrow e_2$. The interest of double expressions in the syntax is to allow for simple symbolic simplifications in later abstraction steps, when computing differences between expressions assigned to a variable.

The semantics for the sequential composition of statements boils down to the composition of the semantics of individual statements. The semantics for selection statements relies on the filter $\mathbb{F}[\![\, c_1 \parallel c_2 \,]\!]$ to distinguish between cases where both program versions agree on the value of the controlling expression, and cases where they do not (*a.k.a.*

unstable tests). There are two stable and two unstable test cases, according to the evaluations of the two conditions. The semantics for stable test cases is standard. The semantics for the first unstable test case is defined by composing the left version of the **then** branch, $\pi_1(s)$, filtered by the condition c_1, and of the right version of the **else** branch, $\pi_2(t)$, filtered by the negation of the condition $\neg c_2$. Intuitively, $\pi_1(s) \parallel \pi_2(t)$ means that the left version of the double program executes the left version of s, while the right version of the double program executes the right version of t. The semantics for the second unstable test case is the dual. The semantics for (possibly unbounded) iteration statements is defined using the least fixpoint of a function defined similarly. Note that $\mathbb{F}[\![c_1 \parallel c_2]\!] = \mathbb{F}[\![c_1 \parallel \text{true}]\!] \cap \mathbb{F}[\![\text{true} \parallel c_2]\!]$, $\mathbb{F}[\![c_1 \wedge c_1' \parallel c_2]\!] = \mathbb{F}[\![c_1 \parallel c_2]\!] \cap \mathbb{F}[\![c_1' \parallel \text{true}]\!]$, and $\mathbb{F}[\![\vee_{i \in I} c_i \parallel c']\!] = \cup_{i \in I} \mathbb{F}[\![c_i \parallel c']\!]$.

Remark 18 (**input** versus **rand**). The semantics $\mathbb{D}[\![V \leftarrow \textbf{input}(a, b)]\!]_\iota$ of input statements is different from the semantics $\mathbb{D}[\![V \leftarrow \textbf{rand}(a, b)]\!]_\iota$ of non-deterministic assignments. The latter entails no relationship between the values read by the two versions of a double program, besides the fact that they range in the same interval. On the contrary, the former reads from a shared input stream ι, hence the left and right versions P_1 and P_2 read equal values if their input indexes n_1 and n_2 are equal. This is the case when P_1 and P_2 have called **input** equal numbers of times. On the contrary, if one version, say P_1, has called **input** more often than the other, then P_1 is ahead of P_2 in the stream, and the two versions are desynchronized. Nonetheless, they may resynchronize later if P_2 catches up with P_1, hence read equal values again. Also, owing to the semantics $\mathbb{S}[\![V \leftarrow \textbf{input}(a, b)]\!]$ of simple input statements, $\textbf{input}(a, b)$ returns only if the input value at the current index is in the range $[a, b]$. Therefore, it should be considered a semantic error if P_1 and P_2 use different ranges $[a_1, b_1] \neq [a_2, b_2]$ to read the input at the same index. For the sake of simplicity, we do not check this in our semantics (altough our implementation performs this check).

The presence of both **input** and **rand** primitives makes the semantics very expressive, and useful for modeling many practical problems. Non-determinism allows to abstract unknown parts of a program: for instance, $\textbf{rand}(0, 10)$ is a sound stub for $f()$, when function f is only known to return values between 0 and 10. Also, combining **input** and **rand** allows to model information flow problems, as we will see in Sec 3.7.2.

3.4.2 Semantics of double programs

We are now ready to define the formal semantics of NIMP$_2$ programs. Our goal is to develop patch and portability analyses, able to compare the behaviors of two versions P_1 and P_2 of a double program $P \in \textit{dstat}$. Hence, our semantics $\mathbb{P}_2[\![P]\!]$ must enable the comparison of the input-output relationships of P_1 and P_2. We thus let $\mathbb{P}_2[\![P]\!]_\iota$ denote the set of pairs of output sequences (o_1, o_2) that may be written by terminating executions of P_1 and P_2, when reading from the same input stream $\iota \in \mathbb{Z}^\omega$.

Consider the initial (simple) memory state ρ_0 with all variables zero-initialized: $\forall V \in \mathcal{V} : \rho_0(V) = 0$. Consider the related initial simple program state, with zero-initialized input index, and empty output sequence $x_0 \triangleq (\rho_0, 0, \epsilon)$. The related initial double

program state is (x_0, x_0). Given some input stream $\iota \in \mathbb{Z}^\omega$, $\mathbb{D}[\![P]\!]_\iota \{ (x_0, x_0) \}$ is the set of possible double post-states of statement P.

Definition 25 (Semantics of double NIMP$_2$ programs)**.** We thus define the semantics of double program P as:

$$\mathbb{P}_2[\![P]\!]_\iota \triangleq \{ (o_1, o_2) \mid ((\rho_1, n_1, o_1), (\rho_2, n_2, o_2)) \in \mathbb{D}[\![P]\!]_\iota \{ (x_0, x_0) \} \}$$

3.4.3 Properties of interest

We wish to prove the functional equivalence of the left and right versions $P_1 = \pi_1(P)$ and $P_2 = \pi_2(P)$ of a given double program $P \in \mathit{dstat}$. P_1 and P_2 are considered functionally equivalent if they write the same sequence of outputs when reading the same sequence of inputs.

Definition 26 (Equivalence property)**.** Therefore the property of interest may be formalized as:

$$\forall \iota \in \mathbb{Z}^\omega : \forall (o_1, o_2) \in \mathbb{P}_2[\![P]\!]_\iota : o_1 = o_2$$

Remark 19 (Equivalence property versus certified compilation)*.* This definition of equivalence is reminiscent of the notion of semantic preservation used in the proof of the CompCert [115] compiler, where the source and transformed programs should exhibit the same traces of input-output operations.

Coming back to the motivating Example 25 from Fig. 3.3(c) of Sec. 3.2, the set of double program states reachable from (x_0, x_0) with input stream $\iota \in \mathbb{Z}^\omega$ can be computed, by hand, from the concrete semantics:

$$\begin{array}{rcl} \mathbb{D}[\![P]\!]_\iota \{ (x_0, x_0) \} & = & \{ \langle \langle R_1\iota, 2, (1) \rangle, \langle R_2\iota, 2, (1) \rangle \rangle \mid \iota_0 < 0 \} \\ & \cup & \{ \langle \langle R'_1\iota, 2, (1 + \iota_0 \times \iota_1) \rangle, \langle R'_2\iota, 2, (1 + \iota_0 \times \iota_1) \rangle \rangle \mid \iota_0 \geq 0 \} \end{array}$$

where $R_1, R_2, R'_1, R'_2 \in \mathbb{Z}^\omega \to \mathcal{E}$ are defined as:

$$\begin{array}{rcl} R_1\sigma & \triangleq & [\ a \mapsto \sigma_0,\ b \mapsto \sigma_1,\ c \mapsto 1,\ i \mapsto 0,\ r \mapsto 1\ ] \\ R_2\sigma & \triangleq & [\ a \mapsto \sigma_0,\ b \mapsto \sigma_1,\ c \mapsto 0,\ i \mapsto 0,\ r \mapsto 1\ ] \\ R'_1\sigma & \triangleq & [\ a \mapsto \sigma_0,\ b \mapsto \sigma_1,\ c \mapsto 1 + \sigma_0 \times \sigma_1,\ i \mapsto \sigma_0,\ r \mapsto 1 + \sigma_0 \times \sigma_1\ ] \\ R'_2\sigma & \triangleq & [\ a \mapsto \sigma_0,\ b \mapsto \sigma_1,\ c \mapsto \sigma_0 \times \sigma_1,\ i \mapsto \sigma_0,\ r \mapsto 1 + \sigma_0 \times \sigma_1\ ] \end{array}$$

The double program semantics is given by:

$$\begin{array}{rcl} \mathbb{P}_2[\![P]\!]_\iota & = & \{ \langle \epsilon, \epsilon \rangle, \langle (1), (1) \rangle \mid \iota_0 < 0 \} \\ & \cup & \{ \langle \epsilon, \epsilon \rangle, \langle (1 + \iota_0 \times \iota_1), (1 + \iota_0 \times \iota_1) \rangle \mid \iota_0 \geq 0 \} \end{array}$$

The semantics $\mathbb{P}_2$ is thus suitable to enable a formal proof that both program versions output the same value, whatever the input sequence ι.

$W \leftarrow \textbf{input}(0,1);$
$V \leftarrow \textbf{input}(-10,10) \parallel \textbf{skip};$
while $W = 1$ **do**
  skip $\parallel V \leftarrow \textbf{input}(-10,10);$
  output(V);
  $W \leftarrow \textbf{input}(0,1);$
  $V \leftarrow \textbf{input}(-10,10) \parallel \textbf{skip}$
done

Figure 3.9: Reording input statements

Unfortunately, our concrete collecting semantics $\mathbb{P}_2$ is not computable in general. A particular difficulty of Example 25 is that the input-ouptut relation is non linear: $(a \leq 0 \Rightarrow r = 1) \wedge (a \geq 0 \Rightarrow r = 1 + a \times b)$. Hence, inferring such information is beyond classic numerical domains, such as polyhedra. We will provide a new analysis method which works using intervals and avoids resorting to more complex, non-linear numerical domains.

Example 26 (Reordering reads from input stream)**.** An additional difficulty, is that some programs may read an unbounded number of values from their input stream. For instance, Fig. 3.9 shows a program reordering reads from the input stream across the body of an unbounded, possibly non-terminating loop. Inputs of even index are Booleans, used to decide whether the program should continue reading from the input stream. Inputs of odd index are copied to the output stream, as long as Boolean inputs are true. P_1 executes one **input** statement more than P_2 in all terminating executions. This program satisfies the equivalence property of Definition 26, as:

$$\mathbb{P}_2[\![P]\!]_\iota = \left\{ \langle o, o \rangle \;\middle|\; \begin{array}{l} |o| \in \mathbb{N} \wedge \iota_{2|o|} = 0 \wedge \iota_{2|o|+1} \in [-10,10] \wedge \\ \forall\, 0 \leq k \leq |o| - 1 :\ \iota_{2k} = 1 \wedge o_k = \iota_{2k+1} \in [-10,10] \end{array} \right\}$$

Our notion of program equivalence is thus robust to reorderings of input statements between program versions.

3.4.4 Non-terminating executions

Our concrete collecting semantics $\mathbb{P}_2$ relates pairs of terminating executions of two versions of a program. It is suitable to prove a number of properties, including that two terminating programs starting from equal initial states will produce equal outputs, a notion called partial equivalence in [79]. In contrast, $\mathbb{P}_2$ does not express any differences between pairs of executions where at least one of the program versions does not terminate.

Example 27 (P_2 may not terminate)**.** For instance, consider the program

$$P \triangleq X \leftarrow \textbf{input}(0,5);\ \textbf{skip} \parallel \textbf{while}\ X = 2\ \textbf{do skip done};\ \textbf{output}(X)$$

while true **do**
  $R \leftarrow X \parallel X + 1$;
  $X \leftarrow$ **input**$(-10, 10)$;
  output(R)
done

(a) Non equivalent versions

while true **do**
  $R \leftarrow X \parallel X + 1$;
  $X \leftarrow$ **input**$(-10, 10)$;
  $Y \leftarrow R + 1 \parallel R$;
  output(Y)
done

(b) Equivalent versions

Figure 3.10: Lockstep composition of versions of a reactive program

P_2 does not terminate for the input value 2, whereas P_1 does. The semantics of P can be computed by hand:

$$\mathbb{D}[\![P]\!]_\iota \{ (x_0, x_0) \} = \{ ((\{ X \mapsto \iota_0 \}, 1, (\iota_0)), (\{ X \mapsto \iota_0 \}, 1, (\iota_0))) \mid \iota_0 \in [0, 1] \cup [3, 5]] \}$$

Hence $\mathbb{P}_2[\![P]\!] = \{ ((\iota_0), (\iota_0)) \mid \iota_0 \in [0, 1] \cup [3, 5]] \}$ satisfies the equivalence property of Definition 26, which only considers outputs of pairs of executions where both P_1 and P_2 terminate.

Moreover, $\mathbb{P}_2$ conveys no information on non-terminating executions. For instance, $\mathbb{P}_2[\![$ **while** true **do** $s]\!]_\iota = \emptyset$ holds for all double statements s and input sequences ι. This is inconvenient, as we would like to be able to compare the behaviors of multiple versions of some reactive software, such as embedded control-command programs, which typically run in infinite loops [61]. We are especially interested in reactive loops where the two program versions are composed in lockstep, like the double program on Fig. 3.10(a). P_1 outputs the infinite sequence $(0, \iota_0, \ldots, \iota_n, \ldots)$, whereas P_2 outputs the infinite sequence $(1, 1+\iota_0, \ldots, 1+\iota_n, \ldots)$. In contrast, both versions of the double program on Fig. 3.10(b) output the infinite sequence $(1, 1 + \iota_0, \ldots, 1 + \iota_n, \ldots)$. Yet, $\mathbb{P}_2$ does not distinguish these two double programs.

To overcome this issue, we extend the language with an additional primitive, used for specifications. We add the **assert_sync** statement to the language of double statements, as a means to assert that both program versions should have written the same sequence of values to the output stream when they reach the current control point. This extension is shown on Fig. 3.11. The semantics of **assert_sync** filters away any double program state with unequal output sequences. In practice, it also reports a semantic error if any such state may be reachable, although we do not represent this in the formalism for conciseness. Note that program versions are only allowed to execute **assert_sync** in lockstep. Indeed, executing **assert_sync** is an error in the simple program semantics.

dstat ::= ⋯
| **assert_sync**

(a) Extension of the syntax

$$\pi_k(\textbf{assert\_sync}) \triangleq \textbf{assert}(\text{false}); \quad k \in \{1,2\}$$
$$\mathbb{D}[\![\,\textbf{assert\_sync}\,]\!]_\iota X \triangleq \{\, ((\rho_1,n_1,o_1),(\rho_2,n_2,o_2)) \in X \mid o_1 = o_2 \,\}$$

(b) Extension of the semantics

Figure 3.11: Extension of *dstat* with **assert_sync**

We may now rewrite the double program of Fig. 3.10(b) as

$$\begin{array}{l} P \triangleq \textbf{while}\ \text{true}\ \textbf{do} \\ \quad R \leftarrow X \parallel X+1; \\ \quad X \leftarrow \textbf{input}(-10,10); \\ \quad Y \leftarrow R+1 \parallel R; \\ \quad \textbf{output}(Y); \\ \quad \textbf{assert\_sync} \\ \textbf{done} \end{array}$$

The calculation of $\mathbb{P}_2[\![\,P\,]\!]_\iota$ encounters no semantic error, as it only evaluates $\mathbb{D}[\![\,\textbf{assert\_sync}\,]\!]_\iota X_n$, for the family of sets of double states $(X_n)_{n\in\mathbb{N}}$ defined by

$$X_0 = \{\, \langle [R\mapsto 0, X\mapsto \iota_0, Y\mapsto 1], 1, (1), [R\mapsto 1, X\mapsto \iota_0, Y\mapsto 1], 1, (1)\rangle \mid \iota_0 \in [-10,10] \,\}$$

and $\forall n \geq 1 : X_n = \{\, \langle [R \mapsto \iota_{n-1}, X \mapsto \iota_n, Y \mapsto 1+\iota_{n-1}], 1+n, (1, 1+\iota_0, \ldots, 1+\iota_{n-1}), [R\mapsto 1+\iota_{n-1}, X\mapsto \iota_n, Y\mapsto 1+\iota_{n-1}], 1+n, (1, 1+\iota_0, \ldots, 1+\iota_{n-1})\rangle \mid \forall k \leq n : \iota_k \in [-10,10] \,\}$.

Remark 20 (**assert_sync**(V) notation). Note that we are especially interested in reactive programs where output statements are executed in lockstep. The encoding of such programs in the NIMP$_2$ syntax typically features sequences
output(V_1); …; **output**(V_n); **assert_sync**,
for variables $V_1, \ldots, V_n \in \mathcal{V}$. We thus abuse notations to simplify the encoding in examples, writing **assert_sync**$(V_1, \ldots, V_n)$ as a shorthad for
output(V_1); …; **output**(V_n); **assert_sync**.
For instance, we may rewrite the double program of Fig. 3.10(b) as:

$$\begin{array}{l} P \triangleq \textbf{while}\ \text{true}\ \textbf{do} \\ \quad R \leftarrow X \parallel X+1; \\ \quad X \leftarrow \textbf{input}(-10,10); \\ \quad Y \leftarrow R+1 \parallel R; \\ \quad \textbf{assert\_sync}(Y) \\ \textbf{done} \end{array}$$

3.5 Abstract semantics

Our concrete collecting semantics $\mathbb{D}$ and $\mathbb{P}_2$ are not computable in general. We therefore tailor an abstract semantics suitable for the analysis of program differences.

Semantics	Sec.	Input abstraction	Output abstraction	Environment abstraction
$\mathbb{D}$	3.4	Infinite sequence	Unbounded sequence	Concrete maps
$\hat{\mathbb{D}}$	3.5.1	Unbounded FIFO	Unbounded sequence	Concrete maps
$\hat{\mathbb{D}}^p$	3.5.2	Queue of length p	Unbounded sequence	Concrete maps
$\tilde{\mathbb{D}}^p$	3.5.3	Queue of length p	Lockstep	Concrete maps
$\tilde{\mathbb{D}}^0$	3.5.4	Lockstep	Lockstep	Concrete maps
$\tilde{\mathbb{D}}^{\sharp p}$	3.5.5	Queue of length p	Lockstep	Standard numerical domains
$\triangle^{\sharp p}$	3.5.6	Queue of length p	Lockstep	Dedicated numerical domain

Figure 3.12: Abstractions of $\mathbb{D}$.

This abstraction is constructed in several steps, summarized on Fig. 3.12. Each row of the table refers to a section that introduces an additional abstraction step, on top of previous ones. Each step abstracts either inputs, or outputs, or memory states (environments). More specifically, sections 3.5.1 to 3.5.4 start with designing uncomputable abstractions of $\mathbb{D}$. Sec. 3.5.1 first abstracts infinite input sequences into unbounded FIFO queues. Then Sec. 3.5.2 abstracts them further into bounded queues, featuring a finite number of variables. Furthermore, Sec. 3.5.3 abstracts away output sequences. In addition, Sec. 3.5.4 introduces a simplified version of the semantics, that is sufficient for the special case of programs reading inputs and writing outputs in lockstep (no desynchronizations). Finally, sections 3.5.5 and 3.5.6 derive computable numerical abstractions of $\mathbb{D}$, resulting in effective static analyses. Section 3.5.5 leverages standard numerical abstractions, while section 3.5.6 introduces a dedicated numerical domain.

3.5.1 Wrapping up infinite input sequences

A first observation is that we do not need to recall the whole input sequence $\iota \in \mathbb{Z}^\omega$ shared by the left and right versions P_1 and P_2 of a double program P. Indeed, we only aim at exploiting equalities between the input values read by P_1 and P_2. We therefore only need to record, at any point in the analysis, the input subsequence that has been read by one program, but not the other one yet. This ensures that, when a program that has read less values than the other catches up with it, it reads the same values. Values read by both programs can be discarded, and values not read by any program do not need to be known in advance, as they can be chosen non-deterministically. This subsequence of input values read by one program only forms an (unbounded) FIFO queue, as inputs are read in order. We therefore abstract the input sequence ι, and indexes n_1 and n_2 of P_1 and P_2 in this sequence, defined in $\mathcal{D}$, as the difference $\delta \triangleq n_2 - n_1$, and a FIFO queue of length $|\delta|$. The new semantic domain is thus $\hat{\mathcal{D}} \triangleq \hat{\Sigma} \times \hat{\Sigma} \times \mathbb{Z} \times \mathbb{Z}^\star$, where $\hat{\Sigma} \triangleq \mathcal{E} \times \mathbb{Z}^\star$ records the memory states and output streams of individual program versions. A formalization of this abstraction is shown on Fig. 3.13. Note that we use the symbol $\dot{\subseteq}$ to denote the pointwise lifting of $\subseteq$: $f \dot{\subseteq} f' \equiv \forall \sigma \in \mathbb{Z}^\omega : f(\sigma) \subseteq f'(\sigma)$.

Proposition 1. *The pair $(\alpha_{\mathfrak{F}}, \gamma_{\mathfrak{F}})$ defined in Fig. 3.13 is a Galois embedding.*

$$(\mathbb{Z}^\omega \to \mathcal{P}(\mathcal{D}), \dot{\subseteq}) \underset{\alpha_{\mathfrak{F}}}{\overset{\gamma_{\mathfrak{F}}}{\leftrightarrows}} (\mathcal{P}(\hat{\mathcal{D}}), \subseteq)$$

$$\begin{array}{rl} \alpha_{\mathfrak{F}}(f) & \triangleq \{\, \beta_\sigma(s) \,|\, s \in f(\sigma) \wedge \sigma \in \mathbb{Z}^\omega \,\} = \bigcup_{\sigma\in\mathbb{Z}^\omega} \{\, \beta_\sigma(f(\sigma)) \,\} \\ (\gamma_{\mathfrak{F}}(\hat{R}))(\sigma) & \triangleq \{\, s \,|\, \beta_\sigma(s) \in \hat{R} \,\} = \beta_\sigma^{-1}(\hat{R}) \end{array}$$

$$\begin{array}{rl} \text{where } \forall \sigma \in \mathbb{Z}^\omega : & \beta_\sigma \in \mathcal{D} \to \hat{\mathcal{D}} \\ & \beta_\sigma\,(((\rho_1, n_1, o_1), (\rho_2, n_2, o_2))) \triangleq ((\rho_1, o_1), (\rho_2, o_2), \delta, q) \\ \text{with} & \quad \delta = n_2 - n_1 \\ & \wedge\ |q| = |\delta| \\ & \wedge\ \forall\, 0 \le n < |\delta| : q_n = \sigma_{\max\{n_1,n_2\}-n-1} \end{array}$$

Figure 3.13: Abstraction of shared input sequences with unbounded FIFO queues

Note that this abstraction includes some redundancy: indeed, it would be enough to record only the sign of δ, instead of its value, as its absolute value is given by the length of the queue. However, keeping the value simplifies subsequent abstraction steps.

Simple programs

Starting from the concrete semantics $\mathbb{D}$, let us now formalize the semantics resulting from this first abstraction step. To start with, we first define the simple program semantics. The behaviors of the left and right versions P_1 and P_2 of a double program P depend on which is ahead in the input sequence, and which is behind. P_1 is ahead if $\delta < 0$, and P_2 is ahead if $\delta > 0$. Therefore, we need to particularize the simple program semantics $\hat{\mathbb{S}}_k[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{E}}) \to \mathcal{P}(\hat{\mathcal{E}})$, where $\hat{\mathcal{E}} \triangleq \hat{\Sigma} \times \mathbb{Z} \times \mathbb{Z}^\star$, and $k \in \{1, 2\}$. Fig. 3.14 shows the semantics for $\hat{\mathbb{S}}_k[\![\, V \leftarrow \mathbf{input}(a, b) \,]\!]$. Note that we write $q{\cdot}q'$ to denote the concatenation of queues q and q'. Intuitively, this semantics distinguishes between two cases:

1. If program P_k is ahead of the other program in the input sequence, or at the same point, then a new successful input read operation produces a fresh input value, and adds it at the head of the queue.

2. If program P_k is behind the other program in the input sequence, then a new successful input read operation retrieves the value at the tail of the queue.

In both cases, an input read operation is only successful if the value read matches the bounds specified for the **input** statement.

We do not display the semantics for other commands, as the semantics for assignments, tests and outputs are unchanged for memory environments and output sequences, and leave input index differences and queues unchanged. For instance:

$$\begin{array}{lcl} \hat{\mathbb{S}}_k[\![\, V \leftarrow e \,]\!]\, \hat{X} & \triangleq & \{\, ((\rho[V \mapsto v], o), \delta, q) \,|\, ((\rho, o), \delta, q) \in \hat{X} \wedge v \in \mathbb{E}[\![\, e \,]\!]\rho \,\} \\ \hat{\mathbb{S}}_k[\![\, \mathbf{output}(V) \,]\!]\, \hat{X} & \triangleq & \{\, ((\rho, o \cdot \rho(V)), \delta, q) \,|\, ((\rho, o), \delta, q) \in \hat{X} \,\} \end{array}$$

The complete semantics is available on Fig. A.1 of appendix A.1.

$$\hat{\mathbb{S}}_k[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{E}}) \to \mathcal{P}(\hat{\mathcal{E}}) \quad ; \quad k \in \{1,2\}$$

$$\hat{\mathbb{S}}_1[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!]\, \hat{X} \triangleq \left\{ ((\rho[V \mapsto \nu], o), \delta - 1, \nu \cdot q) \;\middle|\; \begin{array}{l} ((\rho, o), \delta, q) \in \hat{X} \\ \delta \leq 0 \\ a \leq \nu \leq b \end{array} \right\}$$
$$\cup \left\{ ((\rho[V \mapsto v], o), \delta - 1, q) \;\middle|\; \begin{array}{l} ((\rho, o), \delta, q \cdot v) \in \hat{X} \\ \delta > 0 \\ a \leq \nu \leq b \end{array} \right\}$$

$$\hat{\mathbb{S}}_2[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!]\, \hat{X} \triangleq \left\{ ((\rho[V \mapsto \nu], o), \delta + 1, \nu \cdot q) \;\middle|\; \begin{array}{l} ((\rho, o), \delta, q) \in \hat{X} \\ \delta \geq 0 \\ a \leq \nu \leq b \end{array} \right\}$$
$$\cup \left\{ ((\rho[V \mapsto v], o), \delta + 1, q) \;\middle|\; \begin{array}{l} ((\rho, o), \delta, q \cdot v) \in \hat{X} \\ \delta < 0 \\ a \leq \nu \leq b \end{array} \right\}$$

Figure 3.14: Abstract semantics of simple programs P_1 and P_2 with unbounded queues

$$\hat{\mathbb{D}}[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{D}}) \to \mathcal{P}(\hat{\mathcal{D}})$$

$$\hat{\mathbb{D}}[\![\, s_1 \parallel s_2 \,]\!] \triangleq \hat{\mathbb{D}}_2[\![\, s_2 \,]\!] \circ \hat{\mathbb{D}}_1[\![\, s_1 \,]\!]$$
$$\hat{\mathbb{D}}_1[\![\, s \,]\!]\, \hat{X} \triangleq \{\, (r_1', r_2, \delta', q') \mid (r_1', \delta', q') \in \hat{\mathbb{S}}_1[\![\, s \,]\!]\, \{\, (r_1, \delta, q) \,\} \wedge (r_1, r_2, \delta, q) \in \hat{X} \,\}$$
$$\hat{\mathbb{D}}_2[\![\, s \,]\!]\, \hat{X} \triangleq \{\, (r_1, r_2', \delta', q') \mid (r_2', \delta', q') \in \hat{\mathbb{S}}_2[\![\, s \,]\!]\, \{\, (r_2, \delta, q) \,\} \wedge (r_1, r_2, \delta, q) \in \hat{X} \,\}$$

$$\hat{\mathbb{F}}[\![\, c_1 \parallel c_2 \,]\!] \triangleq \hat{\mathbb{F}}_2[\![\, c_2 \,]\!] \circ \hat{\mathbb{F}}_1[\![\, c_1 \,]\!]$$
$$\hat{\mathbb{F}}_k[\![\, c \,]\!]\, \hat{X} \triangleq \{\, ((\rho_1, o_1), (\rho_2, o_2), \delta, q) \in \hat{X} \mid \mathrm{true} \in \mathbb{C}[\![\, c \,]\!]\rho_k \,\} \;;\; k \in \{1;2\}$$

Figure 3.15: Abstract semantics of double programs with unbounded queues

Double programs

We then lift the semantics $\hat{\mathbb{S}}_1[\![\, s \,]\!]$ and $\hat{\mathbb{S}}_2[\![\, s \,]\!]$ to double programs. The definition of $\hat{\mathbb{D}}[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{D}}) \to \mathcal{P}(\hat{\mathcal{D}})$ is very similar to that of $\mathbb{D}[\![\, s \,]\!]$. It can be obtained by removing ι parameters from Fig. 3.8, except for the composition of syntactically different statements $\hat{\mathbb{D}}[\![\, s_1 \parallel s_2 \,]\!]$ and conditions $\hat{\mathbb{F}}[\![\, c_1 \parallel c_2 \,]\!]$. We thus only show the definitions of these relations on Fig. 3.15, and the remainder on Fig. A.2 of Appendix A.1.

Following the particularization of simple statement semantics, the semantics for double statements and conditions compose the semantics of their left and right versions $\hat{\mathbb{D}}_k[\![\, s_k \,]\!]$ and $\hat{\mathbb{F}}_k[\![\, c_k \,]\!]$, where $\hat{\mathbb{D}}_k$ and $\hat{\mathbb{F}}_k$ operate on simple statements and conditions only. Note that the order of the composition is arbitrary, and not significant, as $\hat{\mathbb{D}}_2[\![\, s \,]\!] \circ \hat{\mathbb{D}}_1[\![\, t \,]\!] = \hat{\mathbb{D}}_1[\![\, t \,]\!] \circ \hat{\mathbb{D}}_2[\![\, s \,]\!]$, and likewise for $\hat{\mathbb{F}}_1[\![\, c \,]\!]$ and $\hat{\mathbb{F}}_2[\![\, d \,]\!]$. Note also that $\hat{\mathbb{D}}_1[\![\, \pi_1(s) \,]\!] = \hat{\mathbb{D}}[\![\, \pi_1(s) \parallel \mathbf{skip} \,]\!]$, and $\hat{\mathbb{D}}_2[\![\, \pi_2(s) \,]\!] = \hat{\mathbb{D}}[\![\, \mathbf{skip} \parallel \pi_2(s) \,]\!]$.

Finally, we formalize the relation between the abstract semantics $\hat{\mathbb{D}}$ and the concrete collecting semantics $\mathbb{D}$.

$$(\mathcal{P}(\hat{\mathcal{D}},\subseteq) \underset{\alpha_p}{\overset{\gamma_p}{\leftrightarrows}} (\mathcal{P}(\hat{\mathcal{D}}_p),\subseteq)$$

$$\begin{aligned}\alpha_p(\hat{R}) &\triangleq \{\, \beta_p(s) \mid s \in \hat{R} \,\} = \beta_p(\hat{R})\\ \gamma_p(\hat{R}_p) &\triangleq \{\, s \mid \beta_p(s) \in \hat{R}_p \,\} = \beta_p^{-1}(\hat{R}_p)\\ \text{where } \beta_p &\in \hat{\mathcal{D}} \to \hat{\mathcal{D}}_p\end{aligned}$$

$$\beta_p(((\rho_1,o_1),(\rho_2,o_2),\delta,q)) \triangleq ((\rho_1,o_1),(\rho_2,o_2),\delta,\tilde{q}) \text{ with } \tilde{q}_n = \begin{cases} q_n \textbf{ if } 0 \le n < |\delta| \\ 0 \quad \textbf{if } |\delta| \le n < p \end{cases}$$

Figure 3.16: Abstraction of FIFO queues to fixed length $p \geq 1$

Proposition 2. *$\hat{\mathbb{D}}$ is the best abstraction of $\mathbb{D}$: $\hat{\mathbb{D}} = \alpha_{\mathfrak{F}} \circ \mathbb{D} \circ \gamma_{\mathfrak{F}}$.*

3.5.2 Bounding input queues

The abstract semantics $\hat{\mathbb{D}}$ features unbounded queues of input values. We aim at abstracting the concrete collecting semantics $\mathbb{D}$ in numerical domains, so we need to deal with a bounded number of variables. As it is also simpler to deal with a fixed number of variables, we parameterize our abstract semantics with some predetermined integer $p \geq 1$, used to define the lengths of abstract FIFO queues in domain $\hat{\mathcal{D}}_p \triangleq \Sigma \times \Sigma \times \mathbb{Z} \times \mathbb{Z}^p$. Queues from $\hat{\mathcal{D}}$ are truncated whenever $|\delta| > p$, and padded with zeros whenever $|\delta| < p$. A formalization of this abstraction is shown on Fig. 3.16.

Proposition 3. *For all $p \geq 1$, the pair (α_p, γ_p) defined in Fig. 3.16 is a Galois embedding.*

Let $p \geq 1$. Starting from semantics $\hat{\mathbb{D}}$, we now give a formal definition for the abstract double program semantics $\hat{\mathbb{D}}^p$ resulting from this second abstraction step.

Simple programs

To this aim, we first define the semantics $\hat{\mathbb{S}}_k^p[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{E}}_p) \to \mathcal{P}(\hat{\mathcal{E}}_p)$ of simple programs, where $\hat{\mathcal{E}}_p \triangleq \Sigma \times \mathbb{Z} \times \mathbb{Z}^p$, and $k \in \{1,2\}$. Fig. 3.17 shows the semantics of $\hat{\mathbb{S}}_k^p[\![\, V \leftarrow \textbf{input}(a,b) \,]\!]$. Intuitively, this semantics distinguishes between three cases:

1. If program P_k is ahead of the other program in the input sequence, or at the same point, then a new successful input read operation produces a fresh input value, and adds it on top of the queue, discarding the value at the bottom at the queue.

2. If program P_k is behind the other program in the input sequence, and the delay is less than the size of the input queue, then a new successful input read operation retrieves the value in the queue indexed by this delay, and resets this value to zero.

3. If program P_k is behind the other program in the input sequence, and the delay is more than the size of the input queue, then a new successful input read operation produces a fresh input value, and leaves the queue unchanged.

$$\hat{\mathbb{S}}^p_k[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{E}}_p) \to \mathcal{P}(\hat{\mathcal{E}}_p) \quad ; \quad k \in \{\, 1, 2 \,\}$$

$$\hat{\mathbb{S}}^p_1[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!]\, \hat{X}_p \triangleq$$
$$\left\{ ((\rho[V \mapsto \nu], o), \delta - 1, \nu \cdot q) \;\middle|\; \delta \leq 0 \wedge \nu \in [a,b] \wedge ((\rho, o), \delta, q \cdot v) \in \hat{X}_p \wedge v \in \mathbb{Z} \right\}$$
$$\cup \left\{ ((\rho[V \mapsto v], o), \delta - 1, q \cdot 0 \cdot r) \;\middle|\; \begin{array}{l} \delta \in (0, p] \wedge v \in [a,b] \\ ((\rho, o), \delta, q \cdot v \cdot r) \in \hat{X}_p \wedge \forall n < p - \delta - 1 : r_n = 0 \end{array} \right\}$$
$$\cup \left\{ ((\rho[V \mapsto \nu], o), \delta - 1, q) \;\middle|\; \delta > p \wedge \nu \in [a,b] \wedge ((\rho, o), \delta, q) \in \hat{X}_p \right\}$$

$$\hat{\mathbb{S}}^p_2[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!]\, \hat{X}_p \triangleq$$
$$\left\{ ((\rho[V \mapsto \nu], o), \delta + 1, \nu \cdot q) \;\middle|\; \delta \geq 0 \wedge \nu \in [a,b] \wedge ((\rho, o), \delta, q \cdot v) \in \hat{X}_p \wedge v \in \mathbb{Z} \right\}$$
$$\cup \left\{ ((\rho[V \mapsto v], o), \delta + 1, q \cdot 0 \cdot r) \;\middle|\; \begin{array}{l} \delta \in [-p, 0) \wedge v \in [a,b] \\ ((\rho, o), \delta, q \cdot v \cdot r) \in \hat{X}_p \wedge \forall n < p - \delta - 1 : r_n = 0 \end{array} \right\}$$
$$\cup \left\{ ((\rho[V \mapsto \nu], o), \delta + 1, q) \;\middle|\; \delta < -p \wedge \nu \in [a,b] \wedge ((\rho, o), \delta, q) \in \hat{X}_p \right\}$$

Figure 3.17: Abstract semantics of simple program P_1 and P_2 with queues of length $p \geq 1$.

In any case, an input read operation is only successful if the value read matches the bounds specified for the **input** statement. We do not display the semantics for other commands, as the semantics for assignments and tests are standard for memory environments, and leave input index differences and queues unchanged. For instance, $\hat{\mathbb{S}}^p_k[\![\, V \leftarrow e \,]\!]\, \hat{X}_p \triangleq \{\, ((\rho[V \mapsto v], o), \delta, q) \,|\, ((\rho, o), \delta, q) \in \hat{X}_p \wedge v \in \mathbb{E}[\![\, e \,]\!]\rho \,\}$. The complete semantics is available on Fig. A.3 of App. A.2.

Double programs

We then lift the semantics $\hat{\mathbb{S}}^p_1[\![\, s \,]\!]$ and $\hat{\mathbb{S}}^p_2[\![\, s \,]\!]$ to double programs. The definition of $\hat{\mathbb{D}}^p[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{D}}_p) \to \mathcal{P}(\hat{\mathcal{D}}_p)$ is very similar to that of $\hat{\mathbb{D}}[\![\, s \,]\!]$. The main change is that $\hat{\mathbb{D}}^p_k[\![\, s \,]\!]$ is defined with $\hat{\mathbb{S}}^p_k[\![\, s \,]\!]$, whereas $\hat{\mathbb{D}}_k[\![\, s \,]\!]$ is defined with $\hat{\mathbb{S}}_k[\![\, s \,]\!]$. We thus only show the definitions of some transfer functions on Fig. 3.18, and the remainder on Fig. A.4 of Appendix A.2. These definitions are very similar to those of $\hat{\mathbb{D}}[\![\, s \,]\!]$ on Fig. 3.15. The semantics for double statements and conditions compose the semantics of their left and right versions. The order of the composition is arbitrary, but significant for statements, as $\hat{\mathbb{D}}^p_2[\![\, s \,]\!] \circ \hat{\mathbb{D}}^p_1[\![\, t \,]\!] \neq \hat{\mathbb{D}}^p_1[\![\, t \,]\!] \circ \hat{\mathbb{D}}^p_2[\![\, s \,]\!]$ due to input statements that can lose precision upon overflowing the bounded queue. Both composition orders, however, are sound. A way to make the analyse precise and independent from the order would be to compute the intersection of the compositions with the two orders. The order is in constrast not significant for conditions, as $\hat{\mathbb{F}}^p_2[\![\, c \,]\!] \circ \hat{\mathbb{F}}^p_1[\![\, d \,]\!] = \hat{\mathbb{F}}^p_1[\![\, d \,]\!] \circ \hat{\mathbb{F}}^p_2[\![\, c \,]\!]$. Note also that $\hat{\mathbb{D}}^p_1[\![\, \pi_1(s) \,]\!] = \hat{\mathbb{D}}^p[\![\, \pi_1(s) \parallel \mathbf{skip} \,]\!]$, and $\hat{\mathbb{D}}^p_2[\![\, \pi_2(s) \,]\!] = \hat{\mathbb{D}}^p[\![\, \mathbf{skip} \parallel \pi_2(s) \,]\!]$.

Finally, we formalize the relation between the abstract semantics $\hat{\mathbb{D}}^p$ and the previous abstraction $\hat{\mathbb{D}}$ of the concrete collecting semantics.

$$\hat{\mathbb{D}}^p[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{D}}_p) \to \mathcal{P}(\hat{\mathcal{D}}_p)$$

$$\begin{aligned}
\hat{\mathbb{D}}^p[\![\, s_1 \parallel s_2 \,]\!] &\triangleq \hat{\mathbb{D}}^p_2[\![\, s_2 \,]\!] \circ \hat{\mathbb{D}}^p_1[\![\, s_1 \,]\!] \\
\hat{\mathbb{D}}^p_1[\![\, s \,]\!]\, \hat{R}_p &\triangleq \{\, (r'_1, r_2, \delta', q') \mid (r'_1, \delta', q') \in \hat{\mathbb{S}}^p_1[\![\, s \,]\!]\ \{\, (r_1, \delta, q) \,\} \wedge (r_1, r_2, \delta, q) \in \hat{R}_p \,\} \\
\hat{\mathbb{D}}^p_2[\![\, s \,]\!]\, \hat{R}_p &\triangleq \{\, (r_1, r'_2, \delta', q') \mid (r'_2, \delta', q') \in \hat{\mathbb{S}}^p_2[\![\, s \,]\!]\ \{\, (r_2, \delta, q) \,\} \wedge (r_1, r_2, \delta, q) \in \hat{R}_p \,\} \\
\hat{\mathbb{F}}^p[\![\, c_1 \parallel c_2 \,]\!] &\triangleq \hat{\mathbb{F}}^p_2[\![\, c_2 \,]\!] \circ \hat{\mathbb{F}}^p_1[\![\, c_1 \,]\!] \\
\hat{\mathbb{F}}^p_k[\![\, c \,]\!]\, \hat{R}_p &\triangleq \{\, ((\rho_1, o_1), (\rho_2, o_2), \delta, q) \in \hat{R}_p \mid \mathrm{true} \in \mathbb{C}[\![\, c \,]\!]\rho_k \,\}\ ;\ k \in \{1; 2\}
\end{aligned}$$

Figure 3.18: Abstract semantics of double programs with queues of length $p \geq 1$

dstat ::= $\cdots$
| **assert_sync**(V)

(a) Extension of the syntax

$$\pi_k(\textbf{assert\_sync}(V)) \triangleq \textbf{assert}(\mathrm{false}); \quad k \in \{1, 2\}$$

$$\mathbb{D}[\![\, \textbf{assert\_sync}(V) \,]\!]_\iota X \triangleq \left\{ ((\rho_1, n_1, o \cdot r), (\rho_2, n_2, o \cdot r)) \;\middle|\; \begin{array}{l} ((\rho_1, n_1, o), (\rho_2, n_2, o)) \in X \\ \wedge\ \rho_1(V) = \rho_2(V) = r \end{array} \right\}$$

(b) Extension of the semantics

Figure 3.19: Extension of *dstat* with **assert_sync**(V)

Proposition 4. *For all $p \geq 1$, $\hat{\mathbb{D}}^p$ is a sound and optimal abstraction of $\hat{\mathbb{D}}$:*

$$\hat{\mathbb{D}}^p = \alpha_p \circ \hat{\mathbb{D}} \circ \gamma_p$$

3.5.3 Obliviating output sequences

The abstract semantics $\hat{\mathbb{D}}^p$ features bounded queues of input values, but still unbounded sequences of output values. We aim at proving the equality of the output sequences o_1 and o_2 of P_1 and P_2, using numerical domains on a finite number of variables.

We could thus abstract outputs the same way we abstracted inputs: using bounded FIFO queues to retain precise information on the differences between the outputs of P_1 and P_2. This approach would indeed enable us to infer the property of interest, and would be robust to bounded desynchronizations between output statements of P_1 and P_2. Yet, the examples we encountered, both in open source software and the embedded reactive software we target, did not raise the need for such precision. Instead, these examples execute output statements in lockstep. Output sequences are thus always equal, or always unequal after some difference arises: there is no resynchronization after bounded desynchronizations.

Nimp$_2^-$, a reduced dialect of Nimp$_2$

There is no need to distinguish **assert_sync** specifications from outputs in the syntax of such examples. The equality of output sequences can be checked at every output.

We therefore introduce a dedicated specialized version of the NIMP$_2$ language. We extend NIMP$_2$ with an additional primitive **assert_sync**(V), used for specifications. In the following, we first formalize the concrete semantics $\mathbb{D}[\![\,\textbf{assert\_sync}(V)\,]\!]$ of this additional statement. Then, we show its semantics $\hat{\mathbb{D}}^p[\![\,\textbf{assert\_sync}(V)\,]\!]$ with the abstraction introduced in Sec. 3.5.2. Both semantics convey information on output streams. Finally, we introduce a new semantics $\tilde{\mathbb{D}}^p[\![\,\textbf{assert\_sync}(V)\,]\!]$ for the specialized version of the NIMP$_2$ language, which abstracts away output streams.

We start with adding the **assert_sync**(V) statement to the language of double statements, as a means to assert that, if both program versions reach the current control point, then they should:

1. have written the same sequence of values to the output stream;
2. agree on the value of V, which is the next value to be written to the output stream.

This extension is shown on Fig. 3.19. The semantics of **assert_sync**(V) filters away any double program states with unequal output sequences, or environments disagreeing on the value of V. In practice, it also reports a semantic error if any such state may be reachable, although we do not represent this in the formalism for conciseness. Note that $\mathbb{D}[\![\,\textbf{assert\_sync}(V)\,]\!] = \mathbb{D}[\![\,\textbf{assert\_sync}\,]\!] \circ \mathbb{D}[\![\,\textbf{output}(V)\,]\!]$, which is compatible with the notation introduced by Remark 20: **assert_sync**(V) may be seen as a shorthand for **output**(V); **assert_sync**. Also note that program versions are only allowed to execute **assert_sync**(V) in lockstep. Indeed, executing **assert_sync**(V) is an error in the simple program semantics.

Consistently, we remove the statements **output**(V) and **assert_sync**() from the syntax of NIMP$_2$, and call NIMP$_2^-$ the resulting language. NIMP$_2^-$ programs can only execute **assert_sync**(V) statements to write to the output stream (in lockstep, otherwise an error is propagated). NIMP$_2^-$ programs cannot resynchronize after bounded desynchronizations of **assert_sync**(V) statements. Therefore it not necessary to keep output sequences in the abstract state to prove the property of interest. The abstract semantics $\hat{\mathbb{D}}^p$ of NIMP$_2^-$ programs is defined as that of NIMP$_2$ programs. This abstraction introduced in Sec. 3.5.2 conveys information on output streams. In particular,

$$\hat{\mathbb{D}}^p[\![\,\textbf{assert\_sync}(V)\,]\!]\,\hat{R}_p \triangleq \left\{ ((\rho_1, o\cdot r),(\rho_2, o\cdot r),\delta,q) \;\middle|\; \begin{array}{l}((\rho_1,o),(\rho_2,o),\delta,q)\in\hat{R}_p\\ \wedge\ \rho_1(V)=\rho_2(V)=r\end{array} \right\}$$

and $\hat{\mathbb{D}}^p_k[\![\,\pi_k(\textbf{assert\_sync}(V))\,]\!] = \hat{\mathbb{S}}^p_k[\![\,\pi_k(\textbf{assert\_sync}(V))\,]\!] = \emptyset$ raise propagated semantic errors. The rest of the transfer functions propagate output sequences unchanged across statements.

Output sequences of program versions are initially empty: $o_1 = o_2 = \epsilon$. In addition, $\hat{\mathbb{D}}^p[\![\,\textbf{assert\_sync}(V)\,]\!]$ is the only transfer function of NIMP$_2^-$ that constructs or filters outputs sequences. It appends equal values to equal sequences. In absence of propagated semantic errors, output sequences of the versions P_1 and P_2 of a NIMP$_2^-$ program can be proved to be always equal in all reachable states of terminating executions, and finite prefixes of non-terminating executions. As a consequence, it is not necessary to keep output sequences in the abstract state of NIMP$_2^-$ programs. We thus remove them from

$$(\mathcal{P}(\hat{\mathcal{D}}_p), \subseteq) \underset{\tilde{\alpha}_p}{\overset{\tilde{\gamma}_p}{\leftrightarrows}} (\mathcal{P}(\tilde{\mathcal{D}}_p), \subseteq)$$

$$\begin{aligned}
\tilde{\alpha}_p(\hat{R}_p) &\triangleq \{\, \varphi_p(s) \mid s \in \hat{R}_p \,\} = \varphi_p(\hat{R}_p) \\
\tilde{\gamma}_p(\tilde{R}_p) &\triangleq \{\, s \mid \varphi_p(s) \in \tilde{R}_p \,\} = \varphi_p^{-1}(\tilde{R}_p) \\
\text{where } \varphi_p &\in \hat{\mathcal{D}}_p \to \tilde{\mathcal{D}}_p \\
\varphi_p(((\rho_1, o_1), (\rho_2, o_2), \delta, q)) &\triangleq (\rho_1, \rho_2, \delta, q)
\end{aligned}$$

Figure 3.20: Simple abstraction of output sequences for NIMP$_2^-$.

the simpler abstract domain $\tilde{\mathcal{D}}_p \triangleq \mathcal{E} \times \mathcal{E} \times \mathbb{Z} \times \mathbb{Z}^p$. A formalization of this abstraction is shown on Fig. 3.20.

Proposition 5. *For all $p \geq 1$, the pair $(\tilde{\alpha}_p, \tilde{\gamma}_p)$ defined in Fig. 3.20 is a Galois embedding.*

Let $p \geq 1$. Starting from semantics $\hat{\mathbb{D}}^p$, we now define the semantics $\tilde{\mathbb{D}}^p$ resulting from this third abstraction step.

Simple programs Nimp without ouputs

We first define the semantics $\tilde{\mathbb{S}}_k^p[\![\, s \,]\!] \in \mathcal{P}(\tilde{\mathcal{E}}_p) \to \mathcal{P}(\tilde{\mathcal{E}}_p)$ of simple programs, where $\tilde{\mathcal{E}}_p \triangleq \mathcal{E} \times \mathbb{Z} \times \mathbb{Z}^p$, and $k \in \{\, 1, 2 \,\}$.

The semantics $\tilde{\mathbb{S}}_k^p[\![\, s \,]\!]$ of statements $s \in \textit{stat}$ is very similar to $\hat{\mathbb{S}}_k^p[\![\, s \,]\!]$ for NIMP statements. The only differences are that the transfer function for **output** is removed, and that output sequences o are no longer propagated. For instance:

$$\tilde{\mathbb{S}}_k^p[\![\, V \leftarrow e \,]\!]\, \tilde{X}_p \triangleq \{\, (\rho[V \mapsto v], \delta, q) \mid (\rho, \delta, q) \in \tilde{X}_p \wedge v \in \mathbb{E}[\![\, e \,]\!]\rho \,\}$$

The complete semantics is available on Fig. A.5 of Appendix A.3.

Double Nimp$_2^-$ programs

We now lift the semantics $\tilde{\mathbb{S}}_1^p[\![\, s \,]\!]$ and $\tilde{\mathbb{S}}_2^p[\![\, s \,]\!]$ to double programs. The definition of $\tilde{\mathbb{D}}^p[\![\, s \,]\!] \in \mathcal{P}(\tilde{\mathcal{D}}_p) \to \mathcal{P}(\tilde{\mathcal{D}}_p)$ for NIMP$_2^-$ statements is very similar to that of $\hat{\mathbb{D}}^p[\![\, s \,]\!]$ for NIMP$_2$ statements. The main changes are that:

1. the transfer functions for **output** and **assert_sync** are removed;
2. the output sequences o are no longer propagated;
3. $\tilde{\mathbb{D}}^p[\![\, \textbf{assert\_sync}(V) \,]\!]$ filters away any abstract state such that P_1 and P_2 disagree on the value of V. In practice, an alarm is raised and propagated whenever any such state may be reached. We do not represent alarms in the semantics for conciseness.
4. $\tilde{\mathbb{D}}_k^p[\![\, s \,]\!]$ is defined with $\tilde{\mathbb{S}}_k^p[\![\, s \,]\!]$, whereas $\hat{\mathbb{D}}_k^p[\![\, s \,]\!]$ is defined with $\hat{\mathbb{S}}_k^p[\![\, s \,]\!]$.

$$\tilde{\mathbb{D}}^p[\![\, s \,]\!] \in \mathcal{P}(\tilde{\mathcal{D}}_p) \to \mathcal{P}(\tilde{\mathcal{D}}_p)$$

$$\begin{aligned}
\tilde{\mathbb{D}}^p[\![\, \textbf{assert_sync}(V) \,]\!]\, \tilde{R}_p &\triangleq \{\, (\rho_1, \rho_2, \delta, q) \in \tilde{R}_p \mid \rho_1(V) = \rho_2(V) \,\} \\
\tilde{\mathbb{D}}^p[\![\, s_1 \parallel s_2 \,]\!] &\triangleq \tilde{\mathbb{D}}^p_2[\![\, s_2 \,]\!] \circ \tilde{\mathbb{D}}^p_1[\![\, s_1 \,]\!] \\
\tilde{\mathbb{D}}^p_1[\![\, s \,]\!]\, \tilde{R}_p &\triangleq \left\{ (\rho_1', \rho_2, \delta', q') \;\middle|\; \begin{array}{l} (\rho_1', \delta', q') \in \tilde{\mathbb{S}}^p_1[\![\, s \,]\!]\, \{\, (\rho_1, \delta, q) \,\} \\ (\rho_1, \rho_2, \delta, q) \in \tilde{R}_p \end{array} \right\} \\
\tilde{\mathbb{D}}^p_2[\![\, s \,]\!]\, \tilde{R}_p &\triangleq \left\{ (\rho_1, \rho_2', \delta', q') \;\middle|\; \begin{array}{l} (\rho_2', \delta', q') \in \tilde{\mathbb{S}}^p_2[\![\, s \,]\!]\, \{\, (\rho_2, \delta, q) \,\} \\ (\rho_1, \rho_2, \delta, q) \in \tilde{R}_p \end{array} \right\} \\
\tilde{\mathbb{F}}^p[\![\, c_1 \parallel c_2 \,]\!] &\triangleq \tilde{\mathbb{F}}^p_2[\![\, c_2 \,]\!] \circ \tilde{\mathbb{F}}^p_1[\![\, c_1 \,]\!] \\
\tilde{\mathbb{F}}^p_k[\![\, c \,]\!]\, \tilde{R}_p &\triangleq \{\, (\rho_1, \rho_2, \delta, q) \in \tilde{R}_p \mid \text{true} \in \mathbb{C}[\![\, c \,]\!]\rho_k \,\} \;;\; k \in \{1; 2\}
\end{aligned}$$

Figure 3.21: Abstract semantics of double NIMP$_2^-$ programs with input queues of length $p \geq 1$.

We thus only show the definitions of some transfer functions on Fig. 3.21. These definitions are very similar to those of $\hat{\mathbb{D}}^p[\![\, s \,]\!]$ on Fig. 3.18. The complete semantics is available on Fig. A.6 of Appendix A.3.

Finally, we formalize the relation between the abstract semantics $\tilde{\mathbb{D}}^p$ and the previous abstraction $\hat{\mathbb{D}}^p$ of the concrete collecting semantics.

Proposition 6 (sound and optimal abstraction for NIMP$_2^-$)**.** *For all $p \geq 1$, $\tilde{\mathbb{D}}^p$ is a sound and optimal abstraction of $\hat{\mathbb{D}}^p$ for NIMP$_2^-$ programs:*

$$\tilde{\mathbb{D}}^p = \tilde{\alpha}_p \circ \hat{\mathbb{D}}^p \circ \tilde{\gamma}_p$$

3.5.4 Special case: inputs and outputs in lockstep

NIMP$_2^-$ programs no longer feature output queues, but still feature bounded queues of input values. The abstract semantics $\tilde{\mathbb{D}}^p$ expresses relational information on the values read by P_1 and P_2 from some input stream, with desynchronizations up to p between **input** statements by P_1 and P_2. It retains only non relational information for longer desynchronizations. We will come back to this semantics in Sec. 3.5.5, and abstract it further into a computable semantics, using numerical domains.

Nimp$_2^\star$, a reduced dialect of Nimp$_2^-$

Nonetheless, we introduce here a simplified version of NIMP$_2^-$, called NIMP$_2^\star$, that is sufficient to handle programs without any desynchronization of inputs or outputs. NIMP$_2^\star$ will allow us to simplify further $\tilde{\mathbb{D}}^p$ into a non computable abstraction $\tilde{\mathbb{D}}^0$. $\tilde{\mathbb{D}}^0$ will be used in later chapters, dedicated to analyses of double C programs: Chapters 5, 6, and 7. Indeed, these chapters will focus on double C programs that read input values in lockstep (and also write output values in lockstep). For instance, the reactive programs of Fig. 3.10(b) execute both **input** and **output** statements in lockstep, while the programs of Fig. 3.9 execute **output** statements in lockstep, but not **input** statements. The programs of

$$\begin{array}{l} \mathit{dstat} ::= \cdots \\ \quad \mid\ V \leftarrow \mathbf{input\_sync}(a,b) \end{array}$$

(a) Extension of the syntax

$$\pi_k(V \leftarrow \mathbf{input\_sync}(a,b)) \triangleq \mathbf{assert}(\text{false}); \quad k \in \{1,2\}$$

$$\mathbb{D}[\![V \leftarrow \mathbf{input\_sync}(a,b)]\!]_\iota X \triangleq \left\{ ((\rho_1[V \mapsto \iota_n], n+1, o_1), (\rho_2[V \mapsto \iota_n], n+1, o_2)) \;\middle|\; \begin{array}{l} ((\rho_1,n,o_2),(\rho_2,n,o_2)) \in X \\ \wedge\ a \leq \iota_n \leq b \end{array} \right\}$$

(b) Extension of the semantics

Figure 3.22: Extension of *dstat*

Fig. 3.10(b) can thus be encoded in NIMP$^\star_2$, and modeled precisely with $\tilde{\mathbb{D}}^0$, while the programs of Fig. 3.9 cannot, and require the more expressive semantics $\tilde{\mathbb{D}}^p$ introduced in Sec. 3.5.3. To this aim, we first extend NIMP$_2$ and NIMP$^-_2$ with an additional primitive $V \leftarrow \mathbf{input\_sync}(a,b)$, used for lockstep input. In the following, we first formalize the concrete semantics $\mathbb{D}[\![V \leftarrow \mathbf{input\_sync}(a,b)]\!]$ of this additional statement. Then, we show its semantics $\tilde{\mathbb{D}}^p[\![V \leftarrow \mathbf{input\_sync}(a,b)]\!]$ with the abstraction introduced in Sec. 3.5.3. Both semantics convey information on input streams. Finally, we introduce a new semantics $\tilde{\mathbb{D}}^0[\![V \leftarrow \mathbf{input\_sync}(a,b)]\!]$ for the simplified NIMP$^\star_2$ language, which abstracts away input streams.

We start with adding the $V \leftarrow \mathbf{input\_sync}(a,b)$ statement to the language of double statements, as a means to assert that, if both program versions reach the current control point, then they should:

1. have read the same number of values from the shared input stream;
2. read a new input value in the range $[a,b]$, and store it into V.

This extension is shown on Fig. 3.22. The semantics of $V \leftarrow \mathbf{input\_sync}(a,b)$ filters away any double program states with unequal input indexes, or input value outside the range $[a,b]$. In practice, it also reports a semantic error if any such state may be reachable, although we do not represent this in the formalism for conciseness. Then, it assigns the input value to V in both program versions. Note that program versions are only allowed to execute $V \leftarrow \mathbf{input\_sync}(a,b)$ in lockstep. Indeed, executing $V \leftarrow \mathbf{input\_sync}(a,b)$ is an error in the simple program semantics.

Consistently, we remove the statement $V \leftarrow \mathbf{input}(a,b)$ from the syntax of NIMP$^-_2$, and call NIMP$^\star_2$ the resulting language. NIMP$^\star_2$ programs can only execute $V \leftarrow \mathbf{input\_sync}(a,b)$ statements to read from the shared input stream (in lockstep, otherwise an error is propagated). NIMP$^\star_2$ programs cannot resynchronize after bounded desynchronizations of $V \leftarrow \mathbf{input\_sync}(a,b)$ statements. Therefore it not necessary to keep information on input sequences in the abstract state to prove the property of interest. The abstract semantics $\tilde{\mathbb{D}}^p$ of NIMP$^\star_2$ programs is defined as that of NIMP$^-_2$ programs. This abstraction introduced in Sec. 3.5.3 conveys information on input streams. In particular,

$$\tilde{\mathbb{D}}^p[\![V \leftarrow \mathbf{input\_sync}(a,b)]\!]\, \tilde{R}_p \triangleq \left\{ (\rho_1[V \mapsto q'_0], \rho_2[V \mapsto q'_0], \delta, q') \;\middle|\; \begin{array}{l} (\rho_1,\rho_2,\delta,q) \in \tilde{R}_p \\ \wedge\ q'_0 \in [a,b] \\ \wedge\ \forall 0 < n < p :\ q'_n = q_{n-1} \end{array} \right\}$$

$$(\mathcal{P}(\tilde{\mathcal{D}}_p), \subseteq) \underset{\tilde{\alpha}_p^0}{\overset{\tilde{\gamma}_p^0}{\leftrightarrows}} (\mathcal{P}(\tilde{\mathcal{D}}_0), \subseteq)$$

$$\begin{array}{rcl}
\tilde{\alpha}_p^0(\tilde{R}_p) & \triangleq & \bigcup_{s \in \tilde{R}_p} \psi_p(s) = \bigcup \psi_p(\tilde{R}_p) \\
\tilde{\gamma}_p^0(\tilde{R}_0) & \triangleq & \{\, s \mid \psi_p(s) \subseteq \tilde{R}_0 \,\} \\
\text{where } \psi_p & \in & \tilde{\mathcal{D}}_p \to \mathcal{P}(\tilde{\mathcal{D}}_0) \\
\psi_p(\rho_1, \rho_2, \delta, q) & \triangleq & \begin{cases} \{\, (\rho_1, \rho_2) \,\} & \textbf{if } \mathcal{R}_p(\delta, q) \\ \tilde{\mathcal{D}}_0 & \textbf{otherwise} \end{cases} \\
\mathcal{R}_p(\delta, q) & \overset{\triangle}{\iff} & \forall |\delta| \leq n < p : \; q_n = 0
\end{array}$$

Figure 3.23: Simple abstraction of input sequences for NIMP$^\star_2$.

and $\tilde{\mathbb{D}}^p_k[\![V \leftarrow \textbf{input\_sync}(a,b)]\!] = \tilde{\mathbb{S}}^p_k[\![\pi_k(V \leftarrow \textbf{input\_sync}(a,b))]\!] = \emptyset$ raise propagated semantic errors. The rest of the transfer functions propagate input delays δ and bounded input sequences q unchanged across statements.

Initially $\delta = 0$, and δ is changed by no transfer function of a NIMP$^\star_2$ program. In absence of propagated semantic errors, $\delta = 0$ can be proved to hold in all reachable states of terminating executions, and finite prefixes of non-terminating executions of NIMP$^\star_2$ programs. As a consequence, it is not necessary to keep input delays δ and bounded input sequences q in the abstract state of NIMP$^\star_2$ programs. We thus remove them from the simpler abstract domain $\tilde{\mathcal{D}}_0 \triangleq \mathcal{E} \times \mathcal{E}$. A formalization of this abstraction is shown on Fig. 3.23.

Proposition 7. *For all $p \geq 1$, the pair $(\tilde{\alpha}^0_p, \tilde{\gamma}^0_p)$ defined in Fig. 3.23 is a Galois embedding.*

Remark 21. The precise value of p is not significant. The same $\tilde{\mathcal{D}}_0$ abstracts $\tilde{\mathcal{D}}_p$ for any $p \geq 1$. Moreover, one could abstract the semantics $\hat{\mathbb{D}}$ of 3.5.3 into $\tilde{\mathcal{D}}_0$ directly. We choose to start from $\tilde{\mathcal{D}}_p$ in order to reuse the abstraction of outputs streams introduced in 3.5.3.

Let $p \geq 1$. Starting from semantics $\tilde{\mathbb{D}}^p$, we now define the semantics $\tilde{\mathbb{D}}^0$ resulting from this additional abstraction step.

Simple programs Nimp without inputs nor ouputs

We first define the semantics $\tilde{\mathbb{S}}^0_k[\![s]\!] \in \mathcal{P}(\tilde{\mathcal{E}}_0) \to \mathcal{P}(\tilde{\mathcal{E}}_0)$ of simple programs, where $\tilde{\mathcal{E}}_0 \triangleq \mathcal{E}$, and $k \in \{\, 1, 2 \,\}$.

The semantics $\tilde{\mathbb{S}}^0_k[\![s]\!]$ of statements $s \in \textit{stat}$ is similar to $\hat{\mathbb{S}}^p_k[\![s]\!]$ for NIMP statements without outputs. The main changes are that:

1. we no longer distinguish $\tilde{\mathbb{S}}^0_1[\![s]\!]$ and $\tilde{\mathbb{S}}^0_2[\![s]\!]$, as this distinction was necessary to account for desynchronized input statements; we thus define a single simple semantics $\tilde{\mathbb{S}}^0[\![s]\!]$;
2. the transfer function for **input** is removed;

$\tilde{\mathbb{D}}^0[\![\, s \,]\!] \in \mathcal{P}(\tilde{\mathcal{D}}_0) \to \mathcal{P}(\tilde{\mathcal{D}}_0)$

$$
\begin{array}{ll}
\tilde{\mathbb{D}}^0[\![\, V \leftarrow \textbf{input\_sync}(a,b) \,]\!]\, \tilde{R}_0 & \triangleq \{\, (\rho_1[V \mapsto v], \rho_2[V \mapsto v]) \mid v \in [a,b] \wedge (\rho_1,\rho_2) \in \tilde{R}_0 \,\} \\
\tilde{\mathbb{D}}^0[\![\, s_1 \parallel s_2 \,]\!] & \triangleq \tilde{\mathbb{D}}^0_2[\![\, s_2 \,]\!] \circ \tilde{\mathbb{D}}^0_1[\![\, s_1 \,]\!] \\
\tilde{\mathbb{D}}^0_1[\![\, s \,]\!]\, \tilde{R}_0 & \triangleq \{\, (\rho_1', \rho_2) \mid \rho_1' \in \tilde{\mathbb{S}}^0[\![\, s \,]\!]\, \{\, \rho_1 \,\} \wedge (\rho_1,\rho_2) \in \tilde{R}_0 \,\} \\
\tilde{\mathbb{D}}^0_2[\![\, s \,]\!]\, \tilde{R}_0 & \triangleq \{\, (\rho_1, \rho_2') \mid \rho_2' \in \tilde{\mathbb{S}}^0[\![\, s \,]\!]\, \{\, \rho_2 \,\} \wedge (\rho_1,\rho_2) \in \tilde{R}_0 \,\} \\
\tilde{\mathbb{F}}^0[\![\, c_1 \parallel c_2 \,]\!] & \triangleq \tilde{\mathbb{F}}^0_2[\![\, c_2 \,]\!] \circ \tilde{\mathbb{F}}^0_1[\![\, c_1 \,]\!] \\
\tilde{\mathbb{F}}^0_k[\![\, c \,]\!]\, \tilde{R}_0 & \triangleq \{\, (\rho_1,\rho_2) \in \tilde{R}_0 \mid \mathrm{true} \in \mathbb{C}[\![\, c \,]\!]\rho_k \,\}\ ;\ k \in \{1;2\}
\end{array}
$$

Figure 3.24: Abstract semantics of double NIMP$^\star_2$ programs.

3. input delays δ and bounded input sequences q are no longer propagated.

For instance:

$$\tilde{\mathbb{S}}^0[\![\, V \leftarrow e \,]\!]\, \tilde{X}_0 \triangleq \{\, \rho[V \mapsto v] \mid \rho \in \tilde{X}_0 \wedge v \in \mathbb{E}[\![\, e \,]\!]\rho \,\}$$

The complete semantics is available on Fig. A.7 of Appendix A.4.

Double Nimp$^\star_2$ programs

We now lift the simple semantics $\tilde{\mathbb{S}}^0[\![\, s \,]\!]$ to double programs. The definition of $\tilde{\mathbb{D}}^0[\![\, s \,]\!] \in \mathcal{P}(\tilde{\mathcal{D}}_0) \to \mathcal{P}(\tilde{\mathcal{D}}_0)$ for NIMP$^\star_2$ statements is similar to that of $\tilde{\mathbb{D}}^p[\![\, s \,]\!]$ for NIMP$^-_2$ statements. The main changes are that:

1. the transfer function for **input** is removed;
2. input delays δ and bounded input sequences q are no longer propagated;
3. $V \leftarrow \textbf{input\_sync}(a,b)$ assigns equal values in the range $[a,b]$ to V in both program versions;
4. $\tilde{\mathbb{D}}^0_k[\![\, s \,]\!]$ is defined with $\tilde{\mathbb{S}}^0[\![\, s \,]\!]$, whereas $\tilde{\mathbb{D}}^p_k[\![\, s \,]\!]$ is defined with $\tilde{\mathbb{S}}^p_k[\![\, s \,]\!]$.

We thus only show the definitions of some transfer functions on Fig. 3.24. These definitions are very similar to those of $\tilde{\mathbb{D}}^p[\![\, s \,]\!]$ on Fig. 3.21. The complete semantics is available on Fig. A.8 of Appendix A.4.

Finally, we formalize the relation between the abstract semantics $\tilde{\mathbb{D}}^0$ and the previous abstraction $\tilde{\mathbb{D}}^p$ of the concrete collecting semantics.

Proposition 8 (sound and optimal abstraction for NIMP$^\star_2$)**.** *For all $p \geq 1$, $\tilde{\mathbb{D}}^0$ is a sound and optimal abstraction of $\tilde{\mathbb{D}}^p$ for NIMP$^\star_2$ programs:*

$$\tilde{\mathbb{D}}^0 = \tilde{\alpha}^0_p \circ \tilde{\mathbb{D}}^p \circ \tilde{\gamma}^0_p$$

3.5.5 Numerical abstraction

In this section, we have constructed so far several non computable abstractions of the concrete collecting semantics $\mathbb{D}$. Fig. 3.25 shows a summary. Let us come back to the semantics $\tilde{\mathbb{D}}^p[\![\, s \,]\!]$ of Sec. 3.5.3, that features bounded inputs sequences of length $p \geq 1$, and no output sequences. We rely now on numerical abstractions to abstract it further

	Sec.	Input abstraction	Output abstraction
$\hat{\mathbb{D}}$	3.5.1	Unbounded FIFO	Unbounded sequence
$\hat{\mathbb{D}}^p$	3.5.2	Queue of length p	Unbounded sequence
$\tilde{\mathbb{D}}^p$	3.5.3	Queue of length p	Lockstep
$\tilde{\mathbb{D}}^0$	3.5.4	Lockstep	Lockstep

Figure 3.25: Uncomputable abstractions of $\mathbb{D}$.

into a computable abstract semantics $\tilde{\mathbb{D}}^{\sharp p}[\![s]\!]$, resulting in an effective static analysis of NIMP_2^- programs.

Connecting to numerical domains

As $\tilde{\mathcal{D}}_p \approx \mathbb{Z}^{2|\mathcal{V}|+p+1}$, any numerical abstract domain with $2|\mathcal{V}|+p+1$ dimensions may be used, such as polyhedra [51].

Let $\mathcal{N}$ be such an abstract domain, with values in $\mathcal{D}^\sharp$, order $\sqsubseteq^\sharp$, concretization $\gamma_{\mathcal{N}} \in \mathcal{D}^\sharp \to \mathcal{P}(\mathbb{Z}^{2|\mathcal{V}|+p+1})$, and operators $\tilde{\mathbb{S}}^{\sharp p}[\![s]\!], \tilde{\mathbb{C}}^{\sharp p}[\![c]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ for assignments and tests of simple programs over variables in $\mathcal{V}_1 \cup \mathcal{V}_2 \cup \mathcal{Q}$, where $\mathcal{V}_k \triangleq \{ x_k \mid x \in \mathcal{V} \}$, and $\mathcal{Q} \triangleq \{ \delta \} \cup \{ q_n \mid 0 \leq n < p \}$. Let $\cup^\sharp$ and $\cap^\sharp$ be the abstractions of set union and intersection of domain $\mathcal{N}$, and $\triangledown$ be its widening operator.

We abstract $\tilde{\mathbb{D}}^p[\![s]\!] \in \mathcal{P}(\tilde{\mathcal{D}}_p) \to \mathcal{P}(\tilde{\mathcal{D}}_p)$ by $\tilde{\mathbb{D}}^{\sharp p}[\![s]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$, with the soundness condition $\forall X^\sharp \in \mathcal{D}^\sharp : \tilde{\mathbb{D}}^p[\![s]\!](\gamma_{\mathcal{N}}(X^\sharp)) \subseteq \gamma_{\mathcal{N}}(\tilde{\mathbb{D}}^{\sharp p}[\![s]\!](X^\sharp))$. As $\tilde{\mathbb{D}}^p[\![s]\!]$ is defined by induction on the syntax, the definition for $\tilde{\mathbb{D}}^{\sharp p}[\![s]\!]$ is straightforward: the abstract semantics needs only be defined for the composition of syntactically different statements $s_1 \parallel s_2$ and conditions $c_1 \parallel c_2$. We let $\tilde{\mathbb{D}}^{\sharp p}[\![s_1 \parallel s_2]\!] \triangleq \tilde{\mathbb{S}}^{\sharp p}[\![\tau_2(s_2)]\!] \circ \tilde{\mathbb{S}}^{\sharp p}[\![\tau_1(s_1)]\!]$, and $\tilde{\mathbb{F}}^{\sharp p}[\![c_1 \parallel c_2]\!] \triangleq \tilde{\mathbb{C}}^{\sharp p}[\![\tau_2(c_2)]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\tau_1(c_1)]\!]$, where we use the syntactic renaming operator τ_1 (resp. τ_2), defined by induction on the syntax, to distinguish the variables of the left (resp. right) version of a double program, with suffix 1 (resp. 2). For instance, $\tilde{\mathbb{D}}^{\sharp p}[\![c \leftarrow 1 \parallel 0]\!] = \tilde{\mathbb{S}}^{\sharp p}[\![c_2 \leftarrow 0]\!] \circ \tilde{\mathbb{S}}^{\sharp p}[\![c_1 \leftarrow 1]\!]$.

The definitions of all transfer functions are shown on Fig. 3.26.

Leveraging standard numerical domains

Coming back to the motivating Example 25 from Fig. 3.3(c) of Sec. 3.2, recall that the relation between `c` and `i` is non linear: $c_1 = i_1 \times b_1 + 1$ and $c_2 = i_2 \times b_2$ from line 4 to line 9. Thus, a separate analysis of programs P_1 and P_2 would require a non linear abstract domain to compare r_1 and r_2. In contrast, our joint analysis of P_1 and P_2 will be sufficiently precise, even when using linear numerical domains, because the difference between the values of the variables in P_1 and in P_2 remains linear. For instance, the polyhedra domain [51] is able to infer that the invariant $-c_1 + c_2 + 1 = 0$ holds from line 3 to 9, hence $r_1 = r_2$ at line 9, although it is not able to discover any interval for r_1 or r_2. The octagon domain [128] is also able to express these invariants, but its transfer function for assignment is not precise enough to infer them. Indeed, $x \leftarrow a - b$ cannot be exactly abstracted by the domain, and currently proposed transfer functions fall back

$$\left.\begin{array}{l}\tilde{\mathbb{S}}^{\sharp p}[\![\, V \leftarrow e \,]\!] \\ \tilde{\mathbb{C}}^{\sharp p}[\![\, e \bowtie e' \,]\!]\end{array}\right] \text{given}$$

$\tilde{\mathbb{D}}^{\sharp p}[\![\, s \,]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$

$$\begin{array}{ll}
\tilde{\mathbb{D}}^{\sharp p}[\![\, \textbf{skip} \,]\!]\, R^\sharp & \triangleq R^\sharp \\
\tilde{\mathbb{D}}^{\sharp p}[\![\, s_1 \parallel s_2 \,]\!] & \triangleq \tilde{\mathbb{D}}_2^{\sharp p}[\![\, s_2 \,]\!] \circ \tilde{\mathbb{D}}_1^{\sharp p}[\![\, s_1 \,]\!] \\
\tilde{\mathbb{D}}_k^{\sharp p}[\![\, s \,]\!] & \triangleq \tilde{\mathbb{S}}^{\sharp p}[\![\, \tau_k(s) \,]\!] \;;\; k \in \{1;2\} \\
\tilde{\mathbb{D}}^{\sharp p}[\![\, V \leftarrow e_1 \parallel e_2 \,]\!] & \triangleq \tilde{\mathbb{S}}^{\sharp p}[\![\, V_2 \leftarrow \tau_2(e_2) \,]\!] \circ \tilde{\mathbb{S}}^{\sharp p}[\![\, V_1 \leftarrow \tau_1(e_1) \,]\!] \\
\tilde{\mathbb{D}}^{\sharp p}[\![\, V \leftarrow e \,]\!] & \triangleq \tilde{\mathbb{S}}^{\sharp p}[\![\, V_2 \leftarrow \tau_2(e) \,]\!] \circ \tilde{\mathbb{S}}^{\sharp p}[\![\, V_1 \leftarrow \tau_1(e) \,]\!] \\
\tilde{\mathbb{D}}^{\sharp p}[\![\, \textbf{assert_sync}(V) \,]\!] & \triangleq \tilde{\mathbb{C}}^{\sharp p}[\![\, V_1 = V_2 \,]\!] \\
\tilde{\mathbb{D}}^{\sharp p}[\![\, \textbf{assert}(c) \,]\!] & \triangleq \tilde{\mathbb{D}}_2^{\sharp p}[\![\, \textbf{assert}(c) \,]\!] \circ \tilde{\mathbb{D}}_1^{\sharp p}[\![\, \textbf{assert}(c) \,]\!] \\
\tilde{\mathbb{D}}^{\sharp p}[\![\, V \leftarrow \textbf{input}(a,b) \,]\!] & \triangleq \tilde{\mathbb{D}}_2^{\sharp p}[\![\, V \leftarrow \textbf{input}(a,b) \,]\!] \circ \tilde{\mathbb{D}}_1^{\sharp p}[\![\, V \leftarrow \textbf{input}(a,b) \,]\!]
\end{array}$$

$$\tilde{\mathbb{D}}_1^{\sharp p}[\![\, V \leftarrow \textbf{input}(a,b) \,]\!] \triangleq \tilde{\mathbb{S}}^{\sharp p}[\![\, \delta \leftarrow \delta - 1 \,]\!] \circ \left(\begin{array}{l} \tilde{\mathbb{S}}^{\sharp p}[\![\, V_1 \leftarrow q_0 \,]\!] \circ \tilde{\mathbb{S}}^{\sharp p}[\![\, q_0 \leftarrow \textbf{rand}(a,b) \,]\!] \circ \tilde{\mathbb{S}}^{\sharp p}[\![\, q_1 \leftarrow q_0 \,]\!] \circ \cdots \circ \tilde{\mathbb{S}}^{\sharp p}[\![\, q_{p-1} \leftarrow q_{p-2} \,]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, \delta \leq 0 \,]\!] \quad \dot{\cup}^\sharp \\ \tilde{\mathbb{S}}^{\sharp p}[\![\, q_{\delta-1} \leftarrow 0 \,]\!] \circ \tilde{\mathbb{S}}^{\sharp p}[\![\, V_1 \leftarrow q_{\delta-1} \,]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, q_{\delta-1} \leq b \,]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, q_{\delta-1} \geq a \,]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, \delta \leq p \,]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, \delta > 0 \,]\!] \quad \dot{\cup}^\sharp \\ \tilde{\mathbb{S}}^{\sharp p}[\![\, V_1 \leftarrow \textbf{rand}(a,b) \,]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, \delta > p \,]\!] \end{array} \right)$$

$$\tilde{\mathbb{D}}_2^{\sharp p}[\![\, V \leftarrow \textbf{input}(a,b) \,]\!] \triangleq \tilde{\mathbb{S}}^{\sharp p}[\![\, \delta \leftarrow \delta + 1 \,]\!] \circ \left(\begin{array}{l} \tilde{\mathbb{S}}^{\sharp p}[\![\, V_2 \leftarrow q_0 \,]\!] \circ \tilde{\mathbb{S}}^{\sharp p}[\![\, q_0 \leftarrow \textbf{rand}(a,b) \,]\!] \circ \tilde{\mathbb{S}}^{\sharp p}[\![\, q_1 \leftarrow q_0 \,]\!] \circ \cdots \circ \tilde{\mathbb{S}}^{\sharp p}[\![\, q_{p-1} \leftarrow q_{p-2} \,]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, \delta \geq 0 \,]\!] \quad \dot{\cup}^\sharp \\ \tilde{\mathbb{S}}^{\sharp p}[\![\, q_{-\delta-1} \leftarrow 0 \,]\!] \circ \tilde{\mathbb{S}}^{\sharp p}[\![\, V_2 \leftarrow q_{-\delta-1} \,]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, q_{-\delta-1} \leq b \,]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, q_{-\delta-1} \geq a \,]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, \delta \geq -p \,]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, \delta < 0 \,]\!] \quad \dot{\cup}^\sharp \\ \tilde{\mathbb{S}}^{\sharp p}[\![\, V_2 \leftarrow \textbf{rand}(a,b) \,]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, \delta < -p \,]\!] \end{array} \right)$$

$$\begin{array}{ll}
\tilde{\mathbb{D}}^{\sharp p}[\![\, s \,;\, t \,]\!] & \triangleq \tilde{\mathbb{D}}^{\sharp p}[\![\, t \,]\!] \circ \tilde{\mathbb{D}}^{\sharp p}[\![\, s \,]\!] \\
\tilde{\mathbb{D}}^{\sharp p}[\![\, \textbf{if } c_1 \parallel c_2 \textbf{ then } s \textbf{ else } t \,]\!] & \triangleq \tilde{\mathbb{D}}^{\sharp p}[\![\, s \,]\!] \circ \tilde{\mathbb{F}}_2^{\sharp p}[\![\, c_2 \,]\!] \quad \circ \tilde{\mathbb{F}}_1^{\sharp p}[\![\, c_1 \,]\!] \\
& \quad \dot{\cup}^\sharp \tilde{\mathbb{D}}^{\sharp p}[\![\, t \,]\!] \quad \circ \tilde{\mathbb{F}}_2^{\sharp p}[\![\, \neg c_2 \,]\!] \circ \tilde{\mathbb{F}}_1^{\sharp p}[\![\, \neg c_1 \,]\!] \\
& \quad \dot{\cup}^\sharp \tilde{\mathbb{D}}_2^{\sharp p}[\![\, \pi_2(t) \,]\!] \circ \tilde{\mathbb{D}}_1^{\sharp p}[\![\, \pi_1(s) \,]\!] \circ \tilde{\mathbb{F}}_2^{\sharp p}[\![\, \neg c_2 \,]\!] \circ \tilde{\mathbb{F}}_1^{\sharp p}[\![\, c_1 \,]\!] \\
& \quad \dot{\cup}^\sharp \tilde{\mathbb{D}}_2^{\sharp p}[\![\, \pi_2(s) \,]\!] \circ \tilde{\mathbb{D}}_1^{\sharp p}[\![\, \pi_1(t) \,]\!] \circ \tilde{\mathbb{F}}_2^{\sharp p}[\![\, c_2 \,]\!] \quad \circ \tilde{\mathbb{F}}_1^{\sharp p}[\![\, \neg c_1 \,]\!] \\
\tilde{\mathbb{D}}^{\sharp p}[\![\, \textbf{if } c \textbf{ then } s \textbf{ else } t \,]\!] & \triangleq \tilde{\mathbb{D}}^{\sharp p}[\![\, \textbf{if } c \parallel c \textbf{ then } s \textbf{ else } t \,]\!] \\
\tilde{\mathbb{D}}^{\sharp p}[\![\, \textbf{while } c_1 \parallel c_2 \textbf{ do } s \,]\!]\, R^\sharp & \triangleq (\tilde{\mathbb{F}}_2^{\sharp p}[\![\, \neg c_2 \,]\!] \circ \tilde{\mathbb{F}}_1^{\sharp p}[\![\, \neg c_1 \,]\!])(\lim H^{R^\sharp}) \\
\tilde{\mathbb{D}}^{\sharp p}[\![\, \textbf{while } c \textbf{ do } s \,]\!] & \triangleq \tilde{\mathbb{D}}^{\sharp p}[\![\, \textbf{while } c \parallel c \textbf{ do } s \,]\!]
\end{array}$$

where $\tilde{\mathbb{F}}_k^{\sharp p}[\![\, c \,]\!] \triangleq \tilde{\mathbb{C}}^{\sharp p}[\![\, \tau_k(c) \,]\!] \;;\; k \in \{1, 2\}$

$$\tau_k(x) \triangleq \begin{cases} x_k \textbf{ if } x \in \mathcal{V} \\ x \quad \textbf{if } x \in \mathcal{Q} \end{cases}$$

and $$H^{R^\sharp}(S^\sharp) \triangleq S^\sharp \triangledown \left(R^\sharp \cup^\sharp \left(\begin{array}{lll} \tilde{\mathbb{D}}^{\sharp p}[\![\, s \,]\!] & \circ \tilde{\mathbb{F}}_2^{\sharp p}[\![\, c_2 \,]\!] & \circ \tilde{\mathbb{F}}_1^{\sharp p}[\![\, c_1 \,]\!] \quad \dot{\cup}^\sharp \\ \tilde{\mathbb{D}}_1^{\sharp p}[\![\, \pi_1(s) \,]\!] & \circ \tilde{\mathbb{F}}_2^{\sharp p}[\![\, \neg c_2 \,]\!] & \circ \tilde{\mathbb{F}}_1^{\sharp p}[\![\, c_1 \,]\!] \quad \dot{\cup}^\sharp \\ \tilde{\mathbb{D}}_2^{\sharp p}[\![\, \pi_2(s) \,]\!] & \circ \tilde{\mathbb{F}}_2^{\sharp p}[\![\, c_2 \,]\!] & \circ \tilde{\mathbb{F}}_1^{\sharp p}[\![\, \neg c_1 \,]\!] \end{array} \right)(S^\sharp) \right)$$

Figure 3.26: Abstract semantics of double programs with a standard numerical domain

to plain interval arithmetics in that case, so that the domain cannot exploit the bound it infers on $a - b$ to bound $\mathtt{x}$, for efficiency reasons. The transfer function of octagons for

$$(\mathcal{P}(\tilde{\mathcal{D}}_p), \subseteq) \underset{\alpha_-}{\overset{\gamma_-}{\leftrightarrows}} (\mathcal{P}(\tilde{\mathcal{D}}_p), \subseteq)$$
$$\alpha_-(\tilde{R}_p) \triangleq \{\, (\rho_1, \rho_2 \dot{-} \rho_1, \delta^\star, q) \,|\, (\rho_1, \rho_2, \delta^\star, q) \in \tilde{R}_p \,\}$$
$$\gamma_-(\tilde{R}_p) \triangleq \{\, (\rho_1, \rho_1 \dot{+} \delta_\rho, \delta^\star, q) \,|\, (\rho_1, \delta_\rho, \delta^\star, q) \in \tilde{R}_p \,\}$$

Figure 3.27: Abstraction of double environments with environment differences

assignments could be improved to handle this case more precisely. The interval domain is not able to express the invariants, hence it cannot be used directly for a conclusive analysis.

3.5.6 Introducing a dedicated numerical domain

Considering the necessary invariants for a successful analysis of Example 25 (Fig. 3.3(c)), we remark that it is sufficient to bound the difference $x_2 - x_1$ for any variable `x` to express these invariants, where x_1 (resp. x_2) represents the value of `x` for the left (resp. right) version P_1 (resp. P_2) of a double program P. Thus, we now design an abstract domain that is specialized to infer these bounds. We abstract the values x_1 and x_2 by the pair $(x_1, \delta x)$, where $\delta x \triangleq x_2 - x_1$. This abstraction amounts to changing the representation of states of double program P. It does not lose information. A formalization of this abstraction is shown on Fig. 3.27. Note that we lift operators $+$ and $-$ over $\mathbb{Z}$ pointwise to operators over maps from variables to values.

Proposition 9. *The pair (α_-, γ_-) defined in Fig. 3.27 is a Galois isomorphism.*

$\mathbb{\Delta}^p \triangleq \alpha_- \circ \tilde{\mathbb{D}}^p \circ \gamma_-$ is able to represent two-variable equalities $x_1 = x_2 \Leftrightarrow \delta x = 0$, even after numerical abstraction using non relational domains, such as intervals. Transfer functions rely on symbolic simplifications to let such equalities propagate through linear expressions. For instance, the semantics $\mathbb{\Delta}^p$ of statements 6 and 9 of the motivating `UnchLoop` Example 25 of Fig. 3.3(c) are shown on Fig. 3.28, before and after simple symbolic simplifications of affine expressions.

The Delta domain

Like for $\tilde{\mathbb{D}}^p$, any numerical domain over variables in $\mathcal{V}_1 \cup \mathcal{V}_\delta \cup \mathcal{Q}$ can be used to abstract $\mathbb{\Delta}^p$, where $\mathcal{V}_1 \triangleq \{\, x_1 \,|\, x \in \mathcal{V} \,\}$, $\mathcal{V}_\delta \triangleq \{\, \delta x \,|\, x \in \mathcal{V} \,\}$, and $\mathcal{Q} \triangleq \{\, \delta^\star \,\} \cup \{\, q_n \,|\, 0 \leq n < p \,\}$. As in Sec. 3.5.5, we use a generic numerical domain $\mathcal{N}$, with values in $\mathcal{D}^\sharp$, order $\sqsubseteq^\sharp$, concretization $\gamma_{\mathcal{N}} \in \mathcal{D}^\sharp \to \mathcal{P}(\mathbb{Z}^{2|\mathcal{V}|+p+1})$, and operators $\tilde{\mathbb{S}}^{\sharp p}[\![\, s \,]\!], \tilde{\mathbb{C}}^{\sharp p}[\![\, c \,]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ for assignments and tests of simple programs over variables in $\mathcal{V}_1 \cup \mathcal{V}_\delta \cup \mathcal{Q}$. Let $\cup^\sharp$ and $\cap^\sharp$ be the abstractions of set union and intersection of domain $\mathcal{N}$, and $\triangledown$ be its widening operator. Composing this generic numerical abstraction with the isomorphic abstraction of Fig. 3.27, we obtain a novel numerical domain:

$$\langle \mathcal{D}^\sharp, \sqsubseteq^\sharp, \gamma_- \circ \gamma_{\mathcal{N}}, \cup^\sharp, \cap^\sharp, \triangledown \rangle$$

$$
\begin{aligned}
&\triangle^p[\![\, c \leftarrow c + b]\!]\, \tilde{R}_p \\
&= \left\{ (\rho_1[c \mapsto c'_1], \delta_\rho[c \mapsto \delta c'], \delta^\star, q) \;\middle|\; \begin{array}{l} \delta c' = c'_2 - c'_1 \\ c'_1 \in \mathbb{E}[\![\, c + b \,]\!]\rho_1 \\ c'_2 \in \mathbb{E}[\![\, c + b \,]\!](\rho_1 + \delta_\rho) \\ (\rho_1, \delta_\rho, \delta^\star, q) \in \tilde{R}_p \end{array} \right\} \\
&= \{\, (\rho_1[c \mapsto \rho_1(c) + \rho_1(b)], \delta_\rho[c \mapsto \delta_\rho(c) + \delta_\rho(b), \delta^\star, q) \mid (\rho_1, \delta_\rho, \delta^\star, q) \in \tilde{R}_p \,\} \\
&\triangle^p[\![\, r \leftarrow c \parallel c + 1]\!] \\
&= \left\{ (\rho_1[r \mapsto r'_1], \delta_\rho[r \mapsto \delta r'], \delta^\star, q) \;\middle|\; \begin{array}{l} \delta r' = r'_2 - r'_1 \\ r'_1 \in \mathbb{E}[\![\, c \,]\!]\rho_1 \\ r'_2 \in \mathbb{E}[\![\, c + 1 \,]\!](\rho_1 + \delta_\rho) \\ (\rho_1, \delta_\rho, \delta^\star, q) \in \tilde{R}_p \end{array} \right\} \\
&= \{\, (\rho_1[r \mapsto \rho_1(c)], \delta_\rho[r \mapsto \delta_\rho(c) + 1, \delta^\star, q) \mid (\rho_1, \delta_\rho, \delta^\star, q) \in \tilde{R}_p \,\}
\end{aligned}
$$

Figure 3.28: Examples of $\triangle^p$ semantics

which we name the Delta domain. This domain is parameterized by a numerical abstraction, which can be a (non-relational) value abstraction or, a (possibly relational) numerical domain.

Transfer functions

The abstract semantics $\triangle^{\sharp p}$ is defined by induction on the syntax of double programs. Fig. 3.29 shows the transfer functions of atomics statements, while Fig. 3.30 shows the transfer functions of compound statements. The semantics of the $s_1 \parallel s_2$ construct is defined as $\triangle^{\sharp p}[\![\, s_1 \parallel s_2]\!] \triangleq \triangle_2^{\sharp p}[\![\, s_2]\!] \circ \triangle_1^{\sharp p}[\![\, s_1]\!]$ on Fig. 3.30, where $\triangle_1^p[\![\, s]\!] \triangleq \triangle^p[\![\, s \parallel \mathbf{skip}]\!]$ and $\triangle_2^p[\![\, s]\!] \triangleq \triangle^p[\![\, \mathbf{skip} \parallel s]\!]$ for simple statement s. Nonetheless, we add some particular cases on Fig. 3.29, to gain both efficiency and precision on δV, for all variables V, through simple symbolic simplifications. Note that we use the syntactic renaming operator τ'_2, defined by induction on the syntax, to replace the variables V_2 of the right version of a double program by their abstraction $V_1 + \delta V$.

Inputs and assignments. The first particular case is that of input statements $V \leftarrow \mathbf{input}(a, b)$ for both program versions, in environments such that both programs have read the same number of input values, *i.e.* $\delta^\star = 0$, where $\delta^\star$ represents the difference between input indexes. In this case, we may assign $\delta V \leftarrow 0$ directly, and leave $\delta^\star$ unchanged. For instance, after statement $a \leftarrow \mathbf{input}(-1000, 1000)$ at line 1 of the `Unchloop` example on Fig. 3.3(c), we have $a \in [-1000, 1000]$, and $\delta a = 0$. The second particular case is that of affine assignments $V \leftarrow e$, where $e = \mu + \sum_{x \in \mathcal{V}} \lambda_x \times x$. We call such expressions "differentiable", as it is easy to compute δV directly as a function of all the δx variables. A third particular case is that of arbitrary (non necessarily affine) assignments $V \leftarrow e$, when e is deterministic, and all the variables x occurring in e are such that $\delta x = 0$. Then $\delta V = 0$, as we know that both expressions always evaluate to equal values in P_1 and P_2.

$\underline{\triangle^{\sharp p}[\![\, s]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp}$

$$\triangle^{\sharp p}[\![\, V \leftarrow \mathbf{input}(a,b)]\!] \triangleq \left(\begin{array}{r} \tilde{\mathbb{S}}^{\sharp p}[\![\, \delta V \leftarrow 0\,]\!] \circ \tilde{\mathbb{S}}^{\sharp p}[\![\, V_1 \leftarrow q_0\,]\!] \circ \tilde{\mathbb{S}}^{\sharp p}[\![\, q_0 \leftarrow \mathbf{rand}(a,b)\,]\!] \circ_{i=0}^{p-2} \tilde{\mathbb{S}}^{\sharp p}[\![\, q_{i+1} \leftarrow q_i\,]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, \delta^\star = 0\,]\!] \cup^\sharp \\ \triangle_2^{\sharp p}[\![\, V \leftarrow \mathbf{input}(a,b)]\!] \circ \triangle_1^{\sharp p}[\![\, V \leftarrow \mathbf{input}(a,b)]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, \delta^\star \neq 0\,]\!] \end{array}\right)$$

$$\begin{array}{ll}
\triangle^{\sharp p}[\![\, V \leftarrow e \parallel e+\alpha]\!] & \triangleq \bar{\tilde{\triangle}}^{\sharp p}[\![\, V \leftarrow e \parallel e+\alpha]\!] \circ \triangle_1^{\sharp p}[\![\, V \leftarrow e]\!] \\
\triangle^{\sharp p}[\![\, V \leftarrow e]\!] & \triangleq \triangle^{\sharp p}[\![\, V \leftarrow e \parallel e+0]\!] \\
\triangle^{\sharp p}[\![\, e \bowtie 0? \parallel e \not\bowtie 0?]\!] & \triangleq \bot \ \textbf{if}\ \text{is_deterministic}(e) \wedge \forall x \in \text{Vars}(e) : \delta x = 0 \\
\triangle^{\sharp p}[\![\, e \bowtie 0? \parallel e \bowtie 0?]\!] & \triangleq \triangle_1^{\sharp p}[\![\, e \bowtie 0?]\!] \ \textbf{if}\ \text{is_deterministic}(e) \wedge \forall x \in \text{Vars}(e) : \delta x = 0 \\
\triangle^{\sharp p}[\![\, \mathbf{assert_sync}(V)]\!] & \triangleq \tilde{\mathbb{C}}^{\sharp p}[\![\, \delta V = 0\,]\!] \\
\triangle^{\sharp p}[\![\, \mathbf{assert}(c)]\!] & \triangleq \triangle_2^{\sharp p}[\![\, \mathbf{assert}(c)]\!] \circ \triangle_1^{\sharp p}[\![\, \mathbf{assert}(c)]\!] \\
\triangle^{\sharp p}[\![\, \mathbf{skip}]\!]\, R^\sharp & \triangleq R^\sharp
\end{array}$$

where

$$\bar{\tilde{\triangle}}^{\sharp p}[\![\, V \leftarrow e \parallel e+\alpha]\!] \triangleq \begin{cases} \tilde{\mathbb{S}}^{\sharp p}[\![\, \delta V \leftarrow \alpha\,]\!] & \textbf{if}\ \text{is\_deterministic}(e) \wedge \forall x \in \text{Vars}(e) : \delta x = 0 \\ \tilde{\mathbb{S}}^{\sharp p}[\![\, \delta V \leftarrow \alpha + \sum_{x\in\mathcal{V}} \lambda_x \delta x\,]\!] & \textbf{if}\ \exists (\mu, (\lambda_x)_{x\in\mathcal{V}}) \in \mathbb{Z}^{|\mathcal{V}|+1} : e = \mu + \sum_{x\in\mathcal{V}} \lambda_x x \\ \tilde{\mathbb{S}}^{\sharp p}[\![\, \delta V \leftarrow \alpha + (\tau_2' - \tau_1)(e)\,]\!] & \textbf{otherwise} \end{cases}$$

$$\tau_2'(x) \triangleq \begin{cases} x_1 + \delta x & \textbf{if}\ x \in \mathcal{V} \\ x & \textbf{if}\ x \in \mathcal{Q} \end{cases}$$

Figure 3.29: Abstract semantics of atomic double statements with the Delta numerical domain

Remark 22 (deterministic expression). We say that an expression $e \in \mathit{expr}$ is deterministic if its value is entirely determined by the memory state $\rho \in \mathcal{E}$, *i.e.* $\mathbb{E}[\![\, e]\!]\rho$ is a singleton. In the case of Nimp expressions, it suffices to check that e does not contain any sub-expression $\mathbf{rand}(a,b)$ to ensure that e is deterministic.

To further enhance precision on some examples, we slightly generalize these particular cases to double assignments $V \leftarrow e_1 \parallel e_2$, when expressions e_1 and e_2 are found syntactically equal, modulo some semantics preserving transformations, such as associativity, commutativity, and distributivity. We also generalize symbolic simplifications to some double assignments $V \leftarrow e \parallel e+\alpha$, when α is a constant. For instance, for line 9 of the `Unchloop` example on Fig. 3.3(c), we have $\triangle_2^{\sharp p}[\![\, r \leftarrow c \parallel c+1]\!] = \tilde{\mathbb{S}}^{\sharp p}[\![\, \delta r \leftarrow \delta c + 1\,]\!]$.

As a consequence, the interval domain is able to infer the invariant $\delta r = 0$ for semantics $\triangle^{\sharp p}$ at line 10 of `Unchloop` example, resulting in a conclusive analysis with linear cost, which is much more efficient than using polyhedra with $\tilde{\mathbb{D}}^{\sharp p}$.

Tests. The default transfer function for double tests $c_1? \parallel c_2?$ is defined as $\triangle^{\sharp p}[\![\, c_1? \parallel c_2?]\!] = \tilde{\mathbb{C}}^{\sharp p}[\![\, \tau_2'(c_2)\,]\!] \circ \tilde{\mathbb{C}}^{\sharp p}[\![\, \tau_1(c_1)\,]\!]$. We nonetheless introduce symbolic simplifications to improve efficiency and precision in two common cases introduced by the semantics of **if** and **while** statements. Statements of the form **if** $e \bowtie 0$ **then** s **else** t introduce indeed two stable tests of the form $e \bowtie 0? \parallel e \bowtie 0?$ and two unstable tests of the form

$\mathbb{A}^{\sharp p}[\![s]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$

$$
\begin{array}{ll}
\mathbb{A}^{\sharp p}[\![s_1 \parallel s_2]\!] & \triangleq \mathbb{A}_2^{\sharp p}[\![s_2]\!] \circ \mathbb{A}_1^{\sharp p}[\![s_2]\!] \\
\mathbb{A}_1^{\sharp p}[\![s]\!] & \triangleq \tilde{\mathbb{S}}^{\sharp p}[\![\tau_1(s)]\!] \\
\mathbb{A}_2^{\sharp p}[\![s]\!] & \triangleq \tilde{\mathbb{S}}^{\sharp p}[\![\tau_2'(s)]\!] \\
\mathbb{A}^{\sharp p}[\![c_1? \parallel c_2?]\!] & \triangleq \mathbb{A}_2^{\sharp p}[\![c_2?]\!] \circ \mathbb{A}_1^{\sharp p}[\![c_1?]\!] \\
\mathbb{A}_1^{\sharp p}[\![c?]\!] & \triangleq \tilde{\mathbb{C}}^{\sharp p}[\![\tau_1(c)]\!] \\
\mathbb{A}_2^{\sharp p}[\![c?]\!] & \triangleq \tilde{\mathbb{C}}^{\sharp p}[\![\tau_2'(c)]\!] \\
\mathbb{A}^{\sharp p}[\![s \,;\, t]\!] & \triangleq \mathbb{A}_2^{\sharp p}[\![t]\!] \circ \mathbb{A}_1^{\sharp p}[\![s]\!] \\
\mathbb{A}^{\sharp p}[\![\textbf{if } c_1 \parallel c_2 \textbf{ then } s \textbf{ else } t]\!] & \triangleq \mathbb{A}^{\sharp p}[\![s]\!] \circ \mathbb{A}^{\sharp p}[\![c_1? \parallel c_2?]\!] \\
& \quad \dot{\cup}^\sharp \mathbb{A}^{\sharp p}[\![t]\!] \circ \mathbb{A}^{\sharp p}[\![\neg c_1? \parallel \neg c_2?]\!] \\
& \quad \dot{\cup}^\sharp \mathbb{A}^{\sharp p}[\![\pi_1(s) \parallel \pi_2(t)]\!] \circ \mathbb{A}^{\sharp p}[\![c_1? \parallel \neg c_2?]\!] \\
& \quad \dot{\cup}^\sharp \mathbb{A}^{\sharp p}[\![\pi_1(t) \parallel \pi_2(s)]\!] \circ \mathbb{A}^{\sharp p}[\![\neg c_1? \parallel c_2?]\!] \\
\mathbb{A}^{\sharp p}[\![\textbf{if } c \textbf{ then } s \textbf{ else } t]\!] & \triangleq \mathbb{A}^{\sharp p}[\![\textbf{if } c \parallel c \textbf{ then } s \textbf{ else } t]\!] \\
\mathbb{A}^{\sharp p}[\![\textbf{while } c_1 \parallel c_2 \textbf{ do } s]\!]\, R^\sharp & \triangleq \mathbb{A}^{\sharp p}[\![\neg c_1? \parallel \neg c_2?]\!]\, (\operatorname{lim} H^{R^\sharp}) \\
\mathbb{A}^{\sharp p}[\![\textbf{while } c \textbf{ do } s]\!] & \triangleq \mathbb{A}^{\sharp p}[\![\textbf{while } c \parallel c \textbf{ do } s]\!]
\end{array}
$$

where

$$
H^{R^\sharp}(S^\sharp) \triangleq S^\sharp \triangledown \left(R^\sharp \cup^\sharp \left(\begin{array}{c} \mathbb{A}^{\sharp p}[\![s]\!] \circ \mathbb{A}^{\sharp p}[\![c_1? \parallel c_2?]\!] \\ \dot{\cup}^\sharp \\ \mathbb{A}_1^{\sharp p}[\![\pi_1(s)]\!] \circ \mathbb{A}^{\sharp p}[\![c_1? \parallel \neg c_2?]\!] \\ \dot{\cup}^\sharp \\ \mathbb{A}_2^{\sharp p}[\![\pi_2(s)]\!] \circ \mathbb{A}^{\sharp p}[\![\neg c_1? \parallel c_2?]\!] \end{array} \right) (S^\sharp) \right)
$$

$$
\tau_2'(x) \triangleq \begin{cases} x_1 + \delta x & \textbf{if } x \in \mathcal{V} \\ x & \textbf{if } x \in \mathcal{Q} \end{cases}
$$

Figure 3.30: Abstract semantics of compound double statements with the Delta numerical domain

$e \bowtie 0? \parallel e \not\bowtie 0?$. If e is a deterministic expression, and the environment is such that $\delta x = 0$ holds for all the variables x occurring in e, then $\tau_2'(e)$ and $\tau_1(e)$ have equal values. In that case, unstable tests are contradictory, and stable tests are redundant: $\mathbb{A}^{\sharp p}[\![e \bowtie 0? \parallel e \not\bowtie 0?]\!] = \bot$ and $\mathbb{A}^{\sharp p}[\![e \bowtie 0? \parallel e \bowtie 0?]\!] = \mathbb{A}_1^{\sharp p}[\![e \bowtie 0?]\!]$. The former optimization is beneficial to both efficiency and precision, while the latter improves efficiency, with no effect on precision. For instance, the loop guard of the `Unchloop` example is proved stable, so the two unstable branches are not analyzed, and each of the two stable tests is analyzed only once.

3.6 Evaluation

We implemented a prototype abstract interpreter for the semantics $\tilde{\mathbb{D}}^{\sharp p}$ and $\mathbb{A}^{\sharp p}$ of the NIMP$_2$ language introduced in this chapter. It is about 2,500 lines of OCaml source code. It uses the APRON [93] library to experiment with the polyhedra and octagon abstract

domains, and the BddApron [94] library to implement state partitioning.

3.6.1 Benchmarking

We compare results on small examples selected from other authors' benchmarks [172, 153, 154]. Some examples are based on synthetic C programs, while others originate from real patches in GNU core utilities. The source code of benchmarks is available in appendix C.1. We added a larger benchmark (also from a Coreutils patch), to evaluate scalability. For most benchmarks, patches preserve most of the loop and branching structure, except for the `seq` benchmark from [153, 154], which features deep modifications of the control structure. The related works do not address streams. As a consequence, these benchmarks do not feature unbounded reads into input streams, except for the `remove` benchmark, which we presented in the introduction: see Fig. 3.1 and Fig. 3.2. Note that we simplified this benchmark to `fstatat` caching for a single file, in order to compare with [153].

[172, 153, 154] analyze pairs of C programs directly, albeit in restricted subsets of C featuring mostly integer programs. In particular, the code from GNU core utilities is simplified. In contrast, we encode their benchmarks as double programs in our toy language Nimp_2 for now. We will address the automatic construction of double programs in chapter 4, and the direct analysis of C programs in Chapters 5 and 6. In addition, [172, 153, 154] not only prove equivalences, but also characterize differences, while we focus on equivalence for now. We therefore selected benchmarks relevant to equivalence only, except for the [172, Fig. 2] example, which we modified slightly to restore equivalence of terminating executions: see App.C.1.3. On the other hand, [172] provide multiple versions of some of their benchmarks, depending on the maximum numbers of loop iterations of the examples. Indeed, the symbolic execution technique they use is very sensitive to this parameter. We do not have this constraint, as we use widening instead of fully unrolling loops, so that we handle directly unbounded loops in a sound way.

Figure 3.31 summarizes the results of our analysis. It shows the analysis timings and results of our prototype, as well as timings of the analyses of the related work, when they are available (their analyses are all successful). All experiments were conducted on a Intel® Core-i7™ processor. The precision of our analyses is comparable with that of the original authors, with speedups of one order of magnitude or more. Some timing differences, of the order of milliseconds, cannot be considered significant, especially as the experiments are not performed on the same machines. A significant point, however, is that the benchmark `UnchLoop` takes 2.8 seconds in [172] when the loop is limited to five iterations. This is five times slower than the benchmark `Comp`. In contrast, with our method, `Const` and `UnchLoop` are analyzed at roughly the same speed, though `UnchLoop` is analyzed for an unbounded number of loop iterations. This difference in behaviors can be explained as a benefit of widening over unrolling loops. Hence, our timing comparison proves that our method can achieve at worst a similar speed, and it is also much more scalable for problems difficult in previous work. Note that [172] compared their method to well-established tools, such as Symdiff [111] and RVT [79], and observed speedups of

one order of magnitude and more with respect to them. Therefore, it is not useful to compare our prototype with these tools on these benchmarks.

Most benchmarks are analyzed successfully with the polyhedra domain, without partitioning. The `seq` benchmark, for instance, is analyzed precisely despite significant changes in the control structure, as the matching of statements is established as part of the syntax of double programs. Only the `remove` benchmark requires partitioning for a successful analysis with the polyhedra domain. Four other benchmarks are analyzed very efficiently with the non relational interval domain, thanks to the $\mathbb{A}^{\sharp p}$ semantics. Partitioning the state with respect to the values of Booleans improves the precision on three other, but reduces efficiency. Nonetheless, some benchmarks, such as `LoopSub`, cannot be analysed conclusively using a non relational numerical domain with semantics $\tilde{\mathbb{D}}^{\sharp p}$ nor $\mathbb{A}^{\sharp p}$. Indeed, related patches exchange the roles of two variables `a` and `b`, so that the challenge is not to infer $a_1 = a_2 \wedge b_1 = b_2$, but $a_1 = b_2 \wedge b_1 = a_2$. We therefore developed a dedicated abstract domain, to refine $\tilde{\mathbb{D}}^{\sharp p}$ with automatically inferred variable equalities. This domain is based on union-find data structure that maintains a partitioning of the set $\mathcal{V}_1 \cup \mathcal{V}_2 \cup \mathcal{Q}$ of program variables. Two variables are part of the same equivalence class if they are guaranteed to be equal. The associated abstract lattice is the dual of the standard geometric lattice of partitions of a finite set: $a \sqsubseteq b$ means that partition b refines partition a, *i.e.* every equivalence class of a is a union of classes of b; $\top$ is the set of singleton variables; and the smallest non $\bot$ element is the whole set of variables. This abstract lattice has finite height, so we use union in place of widening. The `LoopSub` benchmark is analyzed successfully using a reduced product between intervals and this domain.

Related origin	Benchmark	See App.	LOC	Related time	$\mathbb{D}^{\sharp 1}$(polyhedra) Partitioning No		$\mathbb{D}^{\sharp 1}$(polyhedra) Partitioning Yes		$\mathbb{D}^{\sharp 1}$(octagon) Partitioning No		$\mathbb{D}^{\sharp 1}$(octagon) Partitioning Yes		$\mathbb{A}^{\sharp 1}$(interval) Partitioning No		$\mathbb{A}^{\sharp 1}$(interval) Partitioning Yes	
[172]	Comp	C.1.1	13	539 ms	14 ms	✓			18 ms	✗			2 ms	✗		
	Const	C.1.2	9	541 ms	7 ms	✓			17 ms	✓			1 ms	✓		
	Fig. 2	C.1.3	14	–	4 ms	✓			5 ms	✓			1 ms	✓		
	LoopMult[2]	C.1.4	14	49 s	31 ms	✓			165 ms	✗			1 ms	✗		
	LoopSub	C.1.5	15	1.2 s	19 ms	✓			53 ms	✗			2 ms	✗		
	UnchLoop	C.1.6	13	2.8[3] s	15 ms	✓			36 ms	✗			2 ms	✓		
[153]	sign	C.1.7	12	–	6 ms	✓			8 ms	✗	420 ms	✓	2 ms	✗	400 ms	✓
	sum	C.1.8	14	4 s	14 ms	✓			30 ms	✓			6 ms	✗	3.2 s	✓[4]
[153, 154]	copy[1]	C.1.9	37	2 s	23 ms	✓			31 ms	✓			9 ms	✓		
	remove[1]	C.1.10	19	3 s	31.6 s	✗	481 ms	✓	42 ms	✗	322 ms	✓	7 ms	✗		
	seq[1]	C.1.11	41	11 s	75 ms	✓			500 ms	✗			2 ms	✗		
	test[1]	C.1.12	158	–	96 ms	✓			521 ms	✓			4 ms	✓		

Figure 3.31: Benchmarks

[1]Coreutils [2]only 20 loop iterations [3]only 5 loop iterations [4]only 32 values of `len`

```
s = input(-5,5);
b = input(0,1);
{ x = input(0,10); } || {/* skip */}
while  ( b == 1 ) {
  {/* skip */} || { x = input(0,10); }
  s = s + x;
  b = input(0,1);
  { x = input(0,10); } || {/* skip */}
}
assert_sync(s);
```

Figure 3.32: Reordering reads from an input stream

3.6.2 Handling streams

All the benchmarks of Table 3.31 were analyzed using fixed-length queues of length 1, as the related works do not handle input streams. Note that abstracting infinite input streams with fixed-length queues of length 1 is also enough to analyze some patches of infinite-state programs with unbounded loops reading from a stream (e.g. a file), even when patches reorder input statements across the body of unbounded loops.

Fig.3.32 shows an example. This patch reorders input statements in the loop, and changes the number of input statements in terminating executions. The loop is unbounded, and the program is infinite-state. Terminating executions of the left and right projections compute equal values for `s`, though possibly not for `x`. This double program is analyzed successfully with $\tilde{\mathbb{D}}^{\sharp 1}$, using any relational numerical domain: 33 ms for polyhedra, 43 ms for octagon, and 18 ms for the reduced product between the domains of intervals and variable equalities. To the best of our knowledge, no previous work has sound and precise automatic analyses for patches of this type.

In the bounded abstraction of streams, the unbounded FIFO queue represents the subsequence of input values read by the program ahead in the sequence, and not yet read by the program behind. Though we are bounding this queue in the abstract, we retain precise information on executions reading arbitrary long input sequences. The bounded queue allows retaining relational information between all input values read with delays less or equal to the bound, while non relational (interval) information is retained for values read with larger delays. Fig. 3.33 shows a simple example. Using a queue of length 1 is enough to infer the range of variable `s` in both projections of the double program. On the contrary, a queue of length at least 2 is necessary to prove that both programs compute equal values for `s`.

3.7 Related work

The problem of proving the functional equivalence of two programs, or program parts, is fundamental [76]. It aims at comparing the behaviors of two programs running in the

```
{ a = input(0,5); a = input(-5,0); } || {/* skip */}
x = input(0,5);
x = input(-5,0);
s = a || x;
assert(-5 <= s && s <= 0); // inferred with with a queue of length p ≥ 1
assert_sync(s); // inferred with a queue of length p ≥ 2
```

Figure 3.33: Relational and non relational information versus lengths of queues

same environment, *i.e.* their input-output relationships. Our approach to patch analysis is an instance of this general problem, primarily focused on proving the equivalence of syntactically close versions of a program. We thus first compare our work to other state-of-the-art formal approaches in this field in Sec. 3.7.1. Then, we comment in Sec. 3.7.2 on related work on other candidate applications of our analysis, such as inferring information flow properties.

3.7.1 Program equivalence

[92] pioneered the field of semantic differencing between two versions of a procedure by comparing dependencies between input and output variables. While they relied on unsound program analysis techniques, several formal approaches have been developed since this work, leveraging symbolic execution, deductive methods or abstract interpretation.

Symbolic execution

Symbolic execution methods [156, 172, 141] have proposed analysis techniques for programs with small state space and bounded loops, which may support modular regression verification. Unlike that line of work, we can soundly handle programs with unbounded loops like the example of Figs. 3.1 and 3.2, by covering the (possibly infinite) set of execution paths. Some approaches [122] combine symbolic execution and program analysis techniques to improve the coverage of patches with tests suites, but such testing coverage criteria bring no formal guarantee of correctness, unlike our method.

Deductive methods

RVT [79] and SymDiff [111, 112] combine two versions of the same program, with equality constraints on their inputs, and compile equivalence properties into verification conditions to be checked by SMT solvers. Rêve [72] uses Horn constraint solving to infer coupling relations and relational procedure summaries, which works well for similarly structured integer programs. They rely on user-provided relational invariants to enable proofs of equivalence. [109] improves automation by leveraging SMT solvers for Horn constraints to infer relational invariants, in a CEGAR-based approach. On the contrary, we rely on abstract domains to infer equivalence properties. [109] targets pointer programs, which our type Nimp_2^- toy language does not support. Yet, we will present our

implementation of patch analysis for C programs including full support for pointers in Chapter 5. We will evaluate this implementation on benchmarks from [109] in Sec. 5.4.1.

Abstract interpretation

The DIZY [153] and SCORE [154] tools leverage numerical abstract interpretation to establish equivalence under abstraction. In particular, the authors give a semi-formal description of an operational concrete trace semantics. This semantics is not defined by induction on the syntax, and does not support streams. Our main contribution, with respect to this work, is a novel, fully formalized, denotational concrete collecting semantics by induction on the syntax, which can deal with programs reading from infinite input streams, and a novel numerical domain to bound differences between the values of the variables in the two programs. Another difference is that [153] relies on program transformations to build a correlating program, which they analyze according to simple program semantics, while our semantics is defined for double programs directly. In their follow-up paper [154], the authors of [153] improve the precision of their analysis in the case of program versions with very different control structures: they use analysis results to choose an interleaving of the statements of program versions that minimizes abstract semantic difference. We will show in Chapter 4 our approach to automating the construction of a double program from a pair of program versions, and compare it informally to [154].

The FLUCTUAT [123, 80] static analyzer compares the real and floating-point semantics of numerical programs to bound errors in floating-point computations. Like in our concrete semantics, they address unstable test analysis [81]. The authors use the zonotope abstract domain to bound the difference between real and floating-point values. In future work, we plan to experiment combining the Delta domain with the zonotope abstract domain to bound the differences between the variables of two program versions.

3.7.2 Information flow

Program equivalence is concerned with comparing pairs of executions of two programs P and P' running in the same environment. The case $P = P'$ is of little interest for deterministic programs, as they obviously have the same behavior. However, it is relevant for non-determistic programs, such as NIMP_2^- programs using the **rand** primitive. In that case, the program semantics $\mathbb{P}_2[\![P]\!]$ from Definition 25 maps an input stream to a set of reachable pairs of output sequences of program P. Such a set is a program property coined "2-safety property" in [169], which defines it informally as a "property that can be refuted by observing two finite traces". 2-safety properties are a class of hyperproperties [42] that includes information flow properties such as termination insensitive noninterference and secrecy.

As a side-effect of our method, our analysis is able to prove that two sets of executions of the same program write equal values into some output variables. This is useful for proving information flow properties, such as secrecy. For instance, Fig. 3.34 shows two programs with public variable `pub` and secret variable `sec`. These programs read `pub` as

```
pub = input(-10,10);
sec = rand(-5,5);
if (sec < 0) pub = 1;
pub = 0;
assert_sync(pub); // OK
```

(a) secure program

```
pub = input(-10,10);
sec = rand(-5,5);
if (sec < 0) pub = 1;
pub = pub + 1;
assert_sync(pub); // failed
```

(b) insecure program

Figure 3.34: Proving information flow properties

an input value, and choose `sec` non-deterministically. For all pairs of executions reading equal values in `pub`, but possibly different values in `sec`, Program 3.34(a) computes equal values for `pub`, hence ensuring secrecy. On the contrary, Program 3.34(b) leaks the sign of `sec`. Our analysis is able to distinguish these two programs. Indeed, it compares the semantics of two versions of each program. In this case, both versions have exactly the same code.

[21] introduced self-composition, as part of a proof method for information flow properties. Given a simple programs P, they prove safety properties of the simple program $P_1; P_2$, where P_k denotes program P with variables x renamed to x_k to avoid conflicts. Their proof method is sound and relatively complete. However, unlike our method, it cannot be automated, as it would require computing the input-output relations of P_1 and P_2 before comparing them, *i.e.* their concrete semantics. Yet, the proofs derived with their method are equivalent to successful analyses of $P \parallel P$ in our setting, which is the same as analyzing directly P as a double program, the two versions having exactly the same code. Our double program semantics can thus be seen as a form of self-composition in this context. [169] generalized the applicability of self-composition to the verification of any 2-safety property, and several works [18, 20, 60, 145, 110, 143] rely on product programs based on self-compositions for the analysis of hyperproperties. In particular, [19, 110, 143] can deal with syntactically dissimilar programs, as in our method. In addition, [110, 143] automate both the construction of the self-composition (of control-flow graphs) and the verification of 2-hypersafety properties (by abstract interpretation). Though hyperproperties are not the primary focus of this thesis, we conducted preliminary experiments on small pieces of code from the papers of other authors [70, 169, 20, 18], demonstrating that our analysis could prove automatically information flow properties such as secrecy and noninterference. For instance, our analysis of double C programs proves successfully noninterference on a C benchmark from [20, 18]. This will be reported as part of our experimental evaluation in Sec. 5.4.1.

This new application of our analysis is worth exploring in future work. In particular, we could investigate various abstractions of our concrete semantics to infer different kinds of information flow properties, and tackle some challenges such as declassification.

3.8 Conclusion

In this chapter, we have presented a static analysis for software patches. Given two syntactically close versions of a program, our analysis can infer a semantic difference, and prove that both programs compute the same outputs when run on the same inputs. Our method is based on abstract interpretation, and parametric in the choice of an abstract domain. We presented a novel concrete collecting semantics, expressing the behaviors of two syntactically close versions of a program at the same time. This semantics deals with programs reading from unbounded input streams. We also introduced a novel numerical domain to bound differences between the values of the variables in the two programs, which has linear cost. We implemented a prototype for a toy language, and experimented on a few small examples from the literature.

Our analysis does not directly run on a pair of program versions. Instead, we analyze a double program, which we have assumed given so far. We will show how to automate the construction of a double program in Chapter 4. Then, we will show in Chapter 5 how to implement our analysis on the Mopsa platform, to enable the analysis of realistic patches of non purely numerical C programs. In the following chapters, we will consider optimizations and generalizations of our approach to address portability analysis, a related problem where we wish to compare the semantics of the same program in two different environments.

In future work, we plan to experiment with other abstract domains for our analysis, such as zonotopes, to investigate different trade-offs between precision and cost of our analysis. We will also explore the connections between our semantics and information flow problems.

Chapter 4

Double program construction

The patch analysis presented in Chapter 3 is designed to run on a double program, a syntactic structure introduced in Sec. 3.2. We have so far assumed this structure given.

Yet, finding a suitable double program is no trivial issue. Indeed, given two simple programs P_1 and P_2, one can construct multiple double programs P such that $P_1 = \pi_1(P)$ and $P_2 = \pi_2(P)$. In addition, as we will demonstrate on examples in Sec. 4.1, the expressiveness of the invariants that need to be inferred to prove equivalence properties between P_1 and P_2 may depend critically on the chosen double program P.

The design of a front-end that constructs a *suitable* double program P from a pair of program versions P_1 and P_2 is thus subject to a trade-off between its computational cost and that of the subsequent patch analysis. Our a approach relies on syntactic similarities to merge simple statements of P_1 and P_2 into double statements of P. Our heuristic will be presented in Sec. 4.2.

Finally, we will present related works in Sec. 4.3.

4.1 Motivating examples

In this section, we give three examples selected from related works. Each example consists of a pair of program versions P_1 and P_2, both of which contain an unbounded loop.

1. Example 28 is inspired by a motivating example from [18], also referred to as `barthe2` in [72, 107, 109]. P_1 and P_2 are syntactically similar and feature identical control structures. Yet P_2 features an unrolling of the loop of P_1.
2. Example 29 is a simplified version of the `seq` benchmark from [153], which is also the motivating example of [154]. P_1 and P_2 feature significant syntactic difference and dissimilar control structures.
3. Example 30 is a variant of Example 28, and a simplified version of the motivating example of [39]. With respect to P_1, P_2 features a loop optimization known as vectorization, which results in a significant change in the number of loop iterations.

For each example, we describe multiple possible double programs, sorted by increasing sophistication of the construction:

- The simplest double program is the parallel composition $P_1 \parallel P_2$;
- A straightforward double program can be constructed by merging identical statements at matching lines, while double statements or double expressions are introduced wherever syntactic differences arise.
- More involved double program constructions are also demonstrated: program transformations such as loop unrollings are applied to P_1 and P_2 before statements are merged in the straightforward way.

Each such double program features a different trade-off between the sophistication the construction, and the expressiveness of the invariants that should be inferred for a successful patch analysis.

Example 28 (Loop unrolling)**.** Consider the program versions P_1 and P_2 on Fig. 4.1 – inspired by a motivating example from [18]. Both versions write equal outputs $c = a \times b$ at line 9, assuming they have read equal inputs in a and b at lines 1 and 2. Multiple double programs can be built from P_1 and P_2. We describe three options below, sorted by increasing tuning of the double program structure, and decreasing expressiveness of the invariants required for a successful analysis.

1. The most naive double program is the parallel composition $P_1 \parallel P_2$. A successful analysis of this double program requires computing the input-output relations of P_1 and P_2, and comparing these relations. This amounts to computing their concrete semantics, which requires inferring non-linear loop invariants of the form $c_1 = i_1 \times b_1$ and $c_2 = (i_2 + 1) \times b_2$. Such invariants require an expressive domain, which may be detrimental to scalability.
2. A second possible double program P is shown on Fig. 4.1(c). P is built in a straightforward way: identical statements at equal lines are merged, while double statements or expressions are introduced in case of syntactic differences. A successful analysis of P requires inferring the loop invariant $c_2 = c_1 + b_2 \wedge i_2 = i_1 + 1$. This invariant can be inferred by the polyhedra domain.
3. A third possible double program P' is shown on Fig. 4.1(d). The design of P' is more slightly more involved: the first iteration of the loop of P_1 is unrolled before aligning statements straightforwardly. A successful analysis of P' requires inferring the trivial loop invariant $c_2 = c_1 \wedge i_2 = i_1$. This invariant can be inferred by the polyhedra domain, a simple equality domain, or a weakly relational domain with near-linear cost, such as the $\mathbb{A}^\sharp$ domain.

Example 29 (Simplified Coreutils `seq` benchmark)**.** Fig. 4.2 shows a simplified version of the `seq` benchmark from [153], which is also the motivating example of [154]. This benchmark is based on a patch of the `print_numbers(first,step,last)` function of the GNU Coreutils `seq` program. This function prints out a sequence of numbers, starting from $first$ and ending with $last$, in intervals of length $step$. Fig. 4.2 focuses on a constant $step = 2$ and a single point of output for simplicity. Figs. 4.2(a) and 4.2(b) show two versions of `print_numbers`, denoted P_1 and P_2. We give line numbers for both versions,

```
1 :  a ← input(1, 100);
2 :  b ← input(−100, 100);
3 :  c ← 0;
4 :  i ← 0;
5 :  while (i < a) {
6 :     c ← c + b;
7 :     i ← i + 1
8 :  }
9 :  output(c)
```

(a) Left version P_1

```
1 :  a ← input(1, 100);
2 :  b ← input(−100, 100);
3 :  c ← b;
4 :  i ← 1;
5 :  while (i < a) {
6 :     c ← c + b;
7 :     i ← i + 1
8 :  }
9 :  output(c)
```

(b) Right version P_2

```
1 :  a ← input(1, 100);
2 :  b ← input(−100, 100);
3 :  c ← 0 ‖ b;
4 :  i ← 0 ‖ 1;
5 :  while (i < a) {
6 :     c ← c + b;
7 :     i ← i + 1
8 :  }
9 :  output(c)
```

(c) Straightforward double program P

```
1 :  a ← input(1, 100);
2 :  b ← input(−100, 100);
3 :  c ← 0 ‖ b;
4 :  i ← 0 ‖ 1;
     if (i < a) {        ‖
        c ← c + b;       ‖ skip
        i ← i + 1;       ‖
     }                   ‖
5 :  while (i < a) {
6 :     c ← c + b;
7 :     i ← i + 1
8 :  }
9 :  output(c)
```

(d) Double program with unrolling P'

Figure 4.1: Equivalent program versions

as a support for explanations. Note that we added some blank lines in P_1, solely to help the human eye match its statements with those of P_2 – the way we build double programs does not rely on line numbers.

P_1 computes the current number x of the sequence (line 9), as part of a loop starting from index $i = 0$, and writing x to the output r as long as $x \leq last$ (line 10). P_2, in contrast, first checks whether $first \leq last$ (line 5) before computing the sequence of numbers. It then pre-computes the first number and enters the loop with index $i = 1$. While the body of the loop of P_1 computes the number x before writing it to r, the body of the loop of P_2 starts with writing the previous number x to r (line 9) before computing the next number (line 12). P_2 may print an extra number under some conditions, modelled by Boolean *more* on Fig. 4.2(b). We assume *more* = false, as this is a necessary condition for equivalence.

Multiple double programs can be constructed from P_1 and P_2. The most naive one is parallel composition $P_1 \parallel P_2$. To prove equivalence on the resulting double program, the analysis must infer loop invariants of the form $r = first + 2 \times \max\{\, n \in \mathbb{N} \mid 0 \leq n <$

```
1 :  first ← input(0, 100);
2 :  last ← input(0, 100);
3 :  break ← false;
4 :
5 :
6 :
7 :  i ← 0;
8 :  while (¬break) {
9 :    x ← first + i × 2;
10 :   if (last < x)
11 :   then break ← true
12 :   else r ← x;
13 :
14 :   i ← i + 1
15 :  }
16 :
17 :  output(r)
```

(a) P_1 (Coreutils v6.9)

```
1 :  first ← input(0, 100);
2 :  last ← input(0, 100);
3 :  break ← false;
4 :  out ← (last < first);
5 :  if (¬out) {
6 :    x ← first;
7 :    i ← 1;
8 :    while (¬break) {
9 :      r ← x;
10 :     if (out)
11 :     then break ← true
12 :     else { x ← first + i × 2; out ← (last < x);
13 :            if (out ∧ ¬more) then break ← true };
14 :     i ← i + 1
15 :   }
16 : }
17 : output(r)
```

(b) P_2 (Coreutils v6.10)

```
1 :  first ← input(0, 100);
2 :  last ← input(0, 100);
3 :  break ← false;
4 :  i ← 0    ‖ out ← (last < first);
5 :  if (true ‖ ¬out) {
6 :    skip  ‖  x ← first;
7 :          ‖  i ← 1;
8 :    while (¬break) {
9 :      x ← first + i × 2  ‖  r ← x;
10 :     if (last < x ‖ out)
11 :     then break ← true
12 :     else r ← x  ‖  x ← first + i × 2; out ← (last < x);
13 :                 ‖  if (out ∧ ¬more) then break ← true
14 :     i ← i + 1
15 :   }
16 : }
17 : output(r)
```

(c) Double program P

Figure 4.2: Simplified Coreutils `seq` benchmark

$i \wedge first + 2 \times n \leq last\,\}$, which would require an expressive domain.

Like with Example 28, a straightforward double program may be built by merging identical statements at equal lines. This strategy will merge only the statements at

lines 1, 2, 3 and 17. Indeed, the test on line 5 of P_2 has no counterpart in P_1. As a consequence, lines 7 to 15 of P_1 and lines 5 to 16 P_2 can only be aligned in "raw" double statement $P_1^{7..15} \parallel P_2^{5..16}$. To prove equivalence with the resulting double program, the analysis must infer the same invariants as with the most naive alignment $P_1 \parallel P_2$. Yet, the resulting double program may be improved by first applying a simple program transformation to P_1: rewrite $P_1^{7..15}$ to **if** (true) $P_1^{7..15}$, in order to match the test on line 5 of P_2. Then, applying the straightforward merging strategy to the resulting P_1' and P_2, we obtain the double program P shown on Fig. 4.2(c). P aligns the loops of P_1 and P_2. As a consequence, a successful patch analysis of P must infer a single loop invariant, of the form $r_1 = first_1 + 2 \times (i_1 - 1) \wedge i_2 = i_1 + 1 \wedge r_2 = first_2 + 2 \times (i_1 - 2)$. Such invariants may be inferred by the polyhedra domain.

A more involved double program P' may be constructed, by applying aggressive program transformations to P_1' and P_2 before merging statements. Let P_1'' and P_2'' denote the results of these preliminary program transformations. They are shown on Figs. 4.3(a) and 4.3(b), respectively.

Consider P_1''. The assignment of x line 9 of P_1 has been moved before the loop (line 7), and after incrementation of the loop counter (line 14). It has also been transformed into a trivial test statement **if** (false) **then skip else** $x \leftarrow \ldots$ (line 13), mimicking the control structure of lines 10, 11, and 12 of P_2. Line 9 of P_2'' exhibits a similar transformation, mimicking the control structure of lines 9, 10 and 11 of P_1''. Finally, independent lines 6 and 7 of P_2 are swapped.

Let us now construct a double program from P_1'' and P_2''. We merge the test at lines 9, 10 and 11 of P_1'' with the trivial test at line 9 of P_2''. We also merge the trivial test at line 13 of P_1'' with the test at lines 10, 11, and 12 of P_2''. We obtain the double program P' shown on Fig. 4.3(c), which ensures that statements $x \leftarrow first + i \times 2$ and $r \leftarrow x$ are analyzed jointly, with the double program semantics. As a consequence, a simple domain maintaining equalities between the left and right version of every program variable is expressive enough for a successful analysis of double program P'. Note that [154] comes up with a similar alignment of program versions, by heuristically searching for an optimal interleaving of statements of P_1 and P_2 during their analysis.

Example 30 (vectorization)**.** Fig. 4.4 shows a variant of Example 28. This variant is also a simplified version of the motivating example of [39]. Fig.4.4(a) shows program P_1, while Fig.4.4(b) shows program P_2. P_2 vectorizes the loop of P_1. The original example of [39] motivates this optimization by featuring 32-bit operations for P_1 and 64-bit operations for P_2. Like with Example 28, both versions write equal outputs $c = a \times b$ at line 13, and a successful analysis of the parallel composition $P_1 \parallel P_2$ requires inferring non-linear loop invariants. Yet, unlike with Example 28, a straightforward double program P obtained by merging identical statements at equal lines does not improve the situation. Though the loops of P_1 and P_2 are (partly) merged, a successful analysis requires inferring non-linear invariants, as loops do not run a the same paces: linear invariants of the form $i_2 = 2 \times i_1 \wedge c_2 = 2 \times c_1$ (if a_2 is even) or $i_2 = 2 \times i_1 + 1 \wedge c_2 = 2 \times c_1 + b_1$ (if a_2 is odd) hold only as long as $i_2 < a_2$.

```
1 :  first ← input(0, 100);
2 :  last ← input(0, 100);
3 :  break ← false;
4 :
5 :  if (true) then
6 :    i ← 0;
7 :    x ← first + i × 2;
8 :    while (¬break) do
9 :      if (last < x)
10 :     then break ← true
11 :     else r ← x;
12 :     i ← i + 1;
13 :     if (false) then skip
14 :     else x ← first + i × 2
15 : output(r)
```

(a) P_1'' (P_1 transformed)

```
1 :  first ← input(0, 100);
2 :  last ← input(0, 100);
3 :  break ← false;
4 :  out ← (last < first);
5 :  if (¬out) then
6 :    i ← 1;
7 :    x ← first;
8 :    while (¬break) do
9 :      if (false) then skip else r ← x;
10 :     if (out)
11 :     then break ← true
12 :     else { x ← first + i × 2; out ← (last < x);
13 :            if (out ∧ ¬more) then break ← true };
14 :     i ← i + 1
15 : output(r)
```

(b) P_2'' (P_2 transformed)

```
1 :  first ← input(0, 100);
2 :  last ← input(0, 100);
3 :  break ← false;
4 :  skip     ‖ out ← (last < first);
5 :  if (true ‖ ¬out) then
6 :    i ← 0 ‖ 1;
7 :    x ← first + i × 2 ‖ first;
8 :    while (¬break) do
9 :      if (last < x ‖ false)
10 :     then break ← true ‖ skip
11 :     else r ← x;
12 :     i ← i + 1 ‖ skip;
13 :     if (false ‖ out) then skip ‖ break ← true
14 :            { x ← first + i × 2;                                  }
         else { skip  ‖  x ← first + i × 2; out ← (last < x);         };
         skip ‖ i ← i + 1
15 : output(r)
```

(c) Double program P'

Figure 4.3: More involved double program construction

For a successful analysis using only linear invariants, the loop of P_1 must be unrolled before the construction of the double program, as shown on Fig 4.4(c). The first iteration of the loop is unrolled to match the test at line 5 of P_2, and cyclic unrolling is applied to the loop body, to match the increment of the loop counter of P_2. The resulting simple

```
1 :  a ← input(1, 100);
2 :  b ← input(−100, 100);
3 :  c ← 0;
4 :  i ← 0;
5 :
6 :
7 :
8 :
9 :  while (i < a) {
10 :     c ← c + b;
11 :     i ← i + 1
12 :  }
13 : output(c)
```

(a) Left version P_1

```
1 :  a ← input(1, 100);
2 :  b ← input(−100, 100);
3 :  c ← 0;
4 :  i ← 0;
5 :  if (a % 2 = 1) {
6 :     c ← b;
7 :     i ← 1
8 :  }
9 :  while (i < a) {
10 :     c ← c + 2 × b;
11 :     i ← i + 2
12 :  }
13 : output(c)
```

(b) Right version P_2

```
1 :  a ← input(1, 100);
2 :  b ← input(−100, 100);
3 :  c ← 0;
4 :  i ← 0;
5 :  if (i < a) {
6 :     c ← c + b;
7 :     i ← i + 1
8 :  }
9 :  while (i < a) {
10 :    if (i + 1 < a) {
11 :       c ← c + b;
12 :       i ← i + 1;
13 :       c ← c + b;
14 :       i ← i + 1;
15 :    } else {
16 :       c ← c + b;
17 :       i ← i + 1
18 :    }
19 :  }
20 : output(c)
```

(c) Left version P_1' after unrollings

```
1 :  a ← input(1, 100);
2 :  b ← input(−100, 100);
3 :  c ← 0;
4 :  i ← 0;
5 :  if (i < a ‖ a % 2 = 1) {
6 :     c ← c + b ‖ b;
7 :     i ← i + 1 ‖ 1
8 :  }
9 :  while (i < a) {
10 :    if (i + 1 < a ‖ true) {
11 :       c ← c + b ‖ c + 2 × b;
12 :       i ← i + 1 ‖ i + 2;
13 :       c ← c + b;
14 :       i ← i + 1;
15 :    } else {
16 :       c ← c + b; || skip
17 :       i ← i + 1 ||
18 :    }
19 :  }
20 : output(c)
```

(d) Double program P' after unrollings

Figure 4.4: Simplified motivating example of [39]

program P_1' is then merged with a simple program P_2' obtained by wrapping the body of the loop of P_2 inside an **if** (true) statement – to match the test line 10 of P_1'. The resulting double program P' is shown on Fig. 4.4(d).

	$P_1 \parallel P_2$	P	P'
Example 28	non-linear	linear	equalities
Example 29	non-linear	linear	equalities
Example 30	non-linear	non-linear	equalities

Figure 4.5: Summary of possible double programs and invariants

Discussion Fig. 4.5 shows a summary of possible double programs for Examples 28, 29 and 30, with the classes of invariants that need be inferred to prove equivalence. For all three examples, we have been able to construct double programs such that proofs of equivalence only require inferring linear invariants (or weaker invariants), which can be done by the polyhedra abstract domain (or less expressive domains such as an equality domain or the $\triangle^\sharp$ domain). As noted by multiple authors [59, 39, 84], this observation generalizes to most programs transformations, including some of the loop optimizations performed by compilers: such a double program exists in most cases. Nonetheless, as demonstrated by Examples 29 and 30, automating the construction of such a double program may be challenging. Example 30, in particular, features an optimization known as vectorization that changes the number of iterations of a loop significantly. As noted by [39], this is also the case of other loop optimizations, such as loop unrolling, and loop peeling. For such cases, syntactic similarity is not enough to guide the construction of a double program: alignments of loop iterations should be guided by semantical information. Such cases occur typically in the context of translation validation, when targeting black-box verification of optimizations performed by modern compilers [121]. On the contrary, such challenging benchmarks occur rarely in patches of hand-written code, which is our primary target. In our experiences on open source and industrial code, we have encountered patches of this kind solely in some hand-crafted optimizations of C library functions such as `memcpy`.

4.2 Program merging algorithm

In this section, we present our approach to the construction of a double programs P from a pair of simple program versions P_1 and P_2. It takes the form of an algorithm, named **merge_stmt**, which computes a double statement $s \in \mathit{dstat}$ from two simple statements $s_1, s_2 \in \mathit{stat}$.

As shown in Sec. 4.1, a successful patch analysis relies on a trade-off between the computational resources invested in the construction of a "suitable" double program P, and the expressive power of the abstract domain used to infer necessary invariants. As we primarily target patch analysis – rather than translation validation, our approach focuses on heuristics based on syntactic similarity and simple program transformations, leaving more advanced approaches for future work.

As a consequence, **merge_stmt** exhibits the following results on the examples of Sec. 4.1:

- When run on Example 28 it produces the double program P shown on Fig. 4.1(c).

- When run on Example 29, it produces the double program P shown on Fig. 4.2(c).
- When run on Example 30, it produces the straightforward double program P obtained by merging identical statements at equal lines.

The resulting double programs enable successful patch analyses of Examples 28 and 29 with the polyhedra abstract domain (linear invariants). In constrast, the analysis of Example 30 (vectorization) is inconclusive.

In the remainder of this section, we describe our **merge_stmt** algorithm. Our implementation leverages the MOPSA [97] platform to support C programs. We thus delay experimental evaluation to the related Chapter 5. We nonetheless use the more restraint syntax of NIMP and NIMP$_2$ to simplify the presentation in the present chapter.

4.2.1 Overview

Let us start with a high-level overview of our **merge_stmt** heuristic, which constructs a double statement s from two input simple statements s_1 and s_2. s can be seen as a refinement of $s_1 \parallel s_2$. A natural idea is therefore to "push" $\parallel$ symbols from the root of the abstract syntax tree down to nested nodes, as deeply as possible. The goal of this approach is to benefit from our double program semantics, which takes advantage of syntactic similarities to enable joint analyses of program versions, by induction on the syntax. This makes it possible to infer relations between variables of different program versions, without the need for very expressive abstract domains.

To this aim, our **merge_stmt** algorithm operates in three steps. First, identical statements from top-level sequences are merged. Then, sequences of unmerged pairs of statements from P_1 and P_2 are gathered into maximal pairs of sequences. Finally, similar control structures are aligned in each maximal pair.

First step: merging identical statements

In a first step, assume s_1 and s_2 are sequences of statements. **merge_stmt** merges the representations of identical statements in these sequences, leaving version-specific statements unmatched. This first step creates simple statements for every pair of matched statements, and double statements for all unmatched statements.

For instance, starting with simple statements P_1 and P_2 of Example 29 on Fig. 4.2, this first step produces the double statement Φ_1:

$$\Phi_1 \triangleq P^{1..3}\ ;\ (P_1^7 \parallel \textbf{skip})\ ;\ (P_1^{8..15} \parallel \textbf{skip})\ ;\ (\textbf{skip} \parallel P_2^4)\ ;\ (\textbf{skip} \parallel P_2^{5..16})\ ;\ P^{17}$$

where $s^{i..j}$ denotes the sub-statement of s at lines i to j, and s^i is short for $s^{i..i}$. The subsequence $(P_1^7 \parallel \textbf{skip})\ ;\ (P_1^{8..15} \parallel \textbf{skip})\ ;\ (\textbf{skip} \parallel P_2^4)\ ;\ (\textbf{skip} \parallel P_2^{5..16})$ of Φ_1 demonstrates that this first step may create sequences of top-level double statements. Such subsequences are obvious candidates for further merging heuristics. A natural idea is to gather these subsequences and construct, for each of them, a new pair of simple statements s'_1 and s'_2 to be merged further. For instance, $s'_1 \triangleq P_1^7\ ;\ P_1^{8..15} = P_1^{7..15}$ and $s'_2 \triangleq P_2^4\ ;\ P_2^{5..16} = P_2^{4..16}$ in the case of Example 29.

Second step: merging sequences of top-level double statements

In a second step, **merge_stmt** thus greedily merges sequences of top-level double statements into top-level double sequences of simple statements.

For instance, in the case of Example 29, this second step reduces Φ_1 to $\Phi_2 \triangleq P^{1..3}\,;\,(s'_1 \parallel s'_2)\,;\,P^{17}$. The problem is now to find a good alignment of the sequences s'_1 and s'_2, which are known to contain no identical statements. Yet, we would like to "push" $\parallel$ symbols further down in the syntax tree. A simple idea is thus to consider syntactic similarities between statements that are part of the sequences s'_1 and s'_2, *e.g.* similar control structures, which may provide opportunities for partial merges of statements.

Third step: aligning similar control structures

In a third step, **merge_stmt** attempts indeed to merge every remaining top-level double statement into a sequence of simple or double statements, using heuristics based on syntactic similarity. For instance, assignments to identical targets and loops with similar bodies are merged together.

No obvious similarity occurs between s'_1 and s'_2 in the case of Example 29: P_1^7 and P_2^4 feature assignments to different targets, while $P_1^{8..15}$ and $P_2^{5..16}$ feature dissimilar control structures. Yet, $P_2^{5..16}$ includes the nested $P_2^{8..15}$ statement, which exhibits similarities to $P_1^{8..15}$: $P_1^{8..15}$ and $P_2^{8..15}$ feature loops with identical guards and bodies including identical statements ($P_1^{14} = P_2^{14}$ and $P_1^{11} = P_2^{11}$), as well as similar control structures ($P_1^{10..12}$ and $P_2^{10..13}$). A natural semantics-preserving program transformation is to rewrite $P_1^{8..15}$ to the equivalent **if** true $P_1^{8..15}$, in order to match the enclosing control structure of $P_2^{5..16}$. Finally, our problem is now to merge $P_1^7\,;$ **if** true $P_1^{8..15}$ and s'_2. By merging similar control structures at identical lines, we obtain the double statement $P^{4..16}$ shown on Fig. 4.2 of Sec. 4.1, which completes the construction of double program P for Example 29.

4.2.2 Formalization

Let us now formalize the description of **merge_stmt**. To this aim, we use a set of term-rewriting systems:

$$\textbf{merge\_stmt}(s_1, s_2) = s \quad \overset{\triangle}{\Longleftrightarrow} \quad s_1 \parallel s_2 \longrightarrow^{?}_{eq} \longrightarrow^{\omega}_{glue} \longrightarrow^{?}_{ctrl} s$$

The three rewriting systems $\longrightarrow^{?}_{eq}$, $\longrightarrow^{?}_{ctrl}$ and $\longrightarrow^{\omega}_{glue}$ formalize the three steps introduced in the informal overview (Sec. 4.2.1). The definitions for $\longrightarrow^{?}_{eq}$, $\longrightarrow^{?}_{ctrl}$ and $\longrightarrow^{\omega}_{glue}$ are shown on Figs. 4.8 and 4.9. We use the formalism of rewriting systems with conditionals and priority ordering [12, 13, 136]. We use priorities to enforce determinism, while keeping conditionals compact. Indeed, while rewriting systems are often used to describe relations or non-deterministic procedures, **merge_stmt** is a deterministic algorithm. Priorities are defined as a partial ordering over rewrite rules, which is also a total order for every pair of conflicting rules. We use Hasse diagrams to define rule priorities. For instance, rule NOMINAL has priority over rule DEFAULT1 in Fig. 4.8: $\longrightarrow^{?}_{eq}$ (resp. $\longrightarrow^{?}_{ctrl}$) attempts to reduce the input $s_1 \parallel s_2$ with $\longrightarrow_{eq}$ (resp. $\longrightarrow_{ctrl}$),

$$\textsc{id.stmt}\quad s \parallel s \longrightarrow_r s$$

$$\textsc{seq}1.1\ \frac{s_1 \parallel s_2 \longrightarrow_r s \qquad s_1' \parallel s_2' \longrightarrow_r s'}{(s_1; s_1') \parallel (s_2; s_2') \longrightarrow_r s; s'} \qquad \frac{s_1 \parallel s_2 \longrightarrow_r s}{(s_1; s_1') \parallel (s_2; s_2') \longrightarrow_r s; (s_1' \parallel s_2')}\ \textsc{seq}1.2$$

$$\textsc{seq}2.1\ \frac{(s_1; s_1') \parallel s_2' \longrightarrow_r s \qquad s_1' \parallel (s_2; s_2') \longrightarrow_r s' \implies |s'| < |s|}{(s_1; s_1') \parallel (s_2; s_2') \longrightarrow_r (\mathbf{skip} \parallel s_2); s}$$

$$\textsc{seq}2.2\ \frac{s_1' \parallel (s_2; s_2') \longrightarrow_r s' \qquad (s_1; s_1') \parallel s_2' \longrightarrow_r s \implies |s| \leq |s'|}{(s_1; s_1') \parallel (s_2; s_2') \longrightarrow_r (s_1 \parallel \mathbf{skip}); s'}$$

$$\textsc{seq}3.1\ \frac{s_1 \parallel s_2 \longrightarrow_r s}{s_1 \parallel (s_2; s_2') \longrightarrow_r s; (\mathbf{skip} \parallel s_2')} \qquad \frac{s_1 \parallel s_2 \longrightarrow_r s}{(s_1; s_1') \parallel s_2 \longrightarrow_r s; (s_1' \parallel \mathbf{skip})}\ \textsc{seq}3.2$$

$$\textsc{seq}3.3\ \frac{s_1 \parallel s_2' \longrightarrow_r s'}{s_1 \parallel (s_2; s_2') \longrightarrow_r (\mathbf{skip} \parallel s_2); s'} \qquad \frac{s_1' \parallel s_2 \longrightarrow_r s'}{(s_1; s_1') \parallel s_2 \longrightarrow_r (s_1 \parallel \mathbf{skip}); s'}\ \textsc{seq}3.4$$

Figure 4.6: Common reduction rules of double program rewriting systems $\longrightarrow_r$ for $r \in \{\, eq, ctrl \,\}$.

and defaults to the identity if reduction fails. $\longrightarrow^{\omega}_{glue}$, defined on Fig. 4.9, greedily merges sequences of top-level double statements, in order to implement the second step introduced in Sec. 4.2.1.

First step: merging identical statements

The definition of $\longrightarrow_{eq}$ is given by Figs. 4.6 and 4.7. Reduction rules are defined on Fig. 4.6 (with $r = eq$), while the priority ordering between rules is depicted in blue on the Hasse diagram shown in Fig. 4.7. Rule ID.STMT has the highest priority. It is is an axiom that states that every pair of syntactically equal simple statements is merged into a shared simple statement. The lower-priority rules $\textsc{seq}_{x.y}$ merge sequences of simple statements. They reduce a pair of sequences of simple statements $s_1 \parallel s_2 = (s_1^0; \ldots; s_1^n) \parallel (s_2^0; \ldots; s_2^p)$ to a sequence of simple and double statements $s = s^0; \ldots; s^q$, such that $q \leq n + p$, and

$$\forall i < q : \exists i_1 < n, i_2 < p : \left(\begin{array}{l} s^i = s_1^{i_1} = s_2^{i_2} \in stat \quad \vee \\ \exists j_1 < n, j_2 < p : \; s^i = (s_1^{i_1}; \ldots; s_1^{j_1}) \parallel (s_2^{i_2}; \ldots; s_2^{j_2}) \end{array} \right)$$

Note that we use the convention $s_k^i; \ldots; s_k^j = \mathbf{skip}$ if $j < i$. Moreover, the subsequence of statements in s that are simple statements is the longest common subsequence of

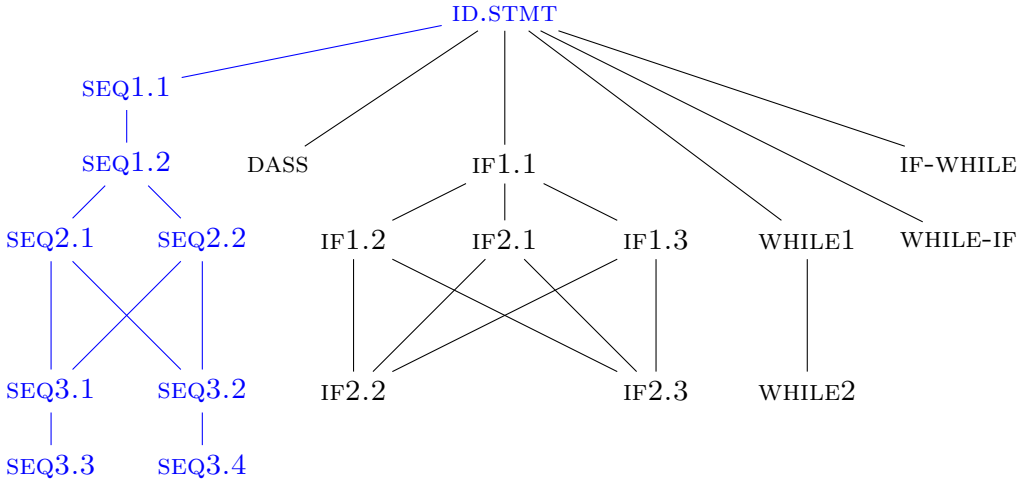


Figure 4.7: Hasse diagram for the priority ordering between rules from Fig. 4.6 and Fig. 4.10.

$$\textsc{nominal}\ \frac{s_1 \parallel s_2 \longrightarrow_r s}{s_1 \parallel s_2 \longrightarrow_r^? s} \qquad \textsc{default1}\ s_1 \parallel s_2 \longrightarrow_r^? s_1 \parallel s_2 \qquad \begin{array}{c}\textsc{Nominal}\\ \vert \\ \textsc{Default1}\end{array}$$

Figure 4.8: Rewriting systems $\longrightarrow_r^?$ for $r \in \{\, eq, ctrl \,\}$.

$$\textsc{glue}\ (s_1 \parallel s_2); (s_1' \parallel s_2') \longrightarrow_{glue} (s_1; s_1') \parallel (s_2; s_2') \qquad \frac{s_0 \longrightarrow_{glue} \cdots \longrightarrow_{glue} s \not\longrightarrow_{glue}}{s_0 \longrightarrow_{glue}^{\omega} s}\ \textsc{iter}$$

Figure 4.9: rewriting system $\longrightarrow_{glue}^{\omega}$

statements of s_1 and s_2. This property is ensured by rules SEQ2.1 and SEQ2.2, with the measure $|s| \triangleq |\{\, i \in \mathbb{N} \mid s^i \in stat \,\}|$.

Consider the case of Example 29. Fig. 4.11 shows a derivation tree for the reduction $P_1 \parallel P_2 \longrightarrow_{eq} \Phi_1$ introduced in Sec. 4.2.1. Note that some premises are omitted in the two occurrences of rule SEQ2.2, for readability: for both occurrences, either reduction in the premises produces the longest common subsequence.

Remark 23 (Implementation with dynamic programming). A straightforward recursive implementation[1] of these rules is inefficient, as intermediate results are recomputed multiple times. It can be improved through memoization[2] techniques. We nonetheless use a classical iterative algorithm[3] based on standard dynamic programming tech-

[1] `https://rosettacode.org/wiki/Longest_common_subsequence#Recursion_5`
[2] `https://rosettacode.org/wiki/Longest_common_subsequence#Memoized_recursion`
[3] `https://rosettacode.org/wiki/Longest_common_subsequence#Dynamic_programming_6`

$$\text{ID.COND}\ \frac{}{c \parallel c \longrightarrow_{ctrl} c} \qquad \text{ID.EXP}\ \frac{}{e \parallel e \longrightarrow_{ctrl} e} \qquad \text{DASS}\ \frac{x_1 \parallel x_2 \longrightarrow_{ctrl} x}{(x_1 \leftarrow e_1) \parallel (x_2 \leftarrow e_2) \longrightarrow_{ctrl} (x \leftarrow e_1 \parallel e_2)}$$

$$\text{WHILE1}\ \frac{c_1 \parallel c_2 \longrightarrow_{ctrl} c \qquad s_1 \parallel s_2 \longrightarrow_{ctrl} s}{\textbf{while}\ c_1\ \textbf{do}\ s_1 \parallel \textbf{while}\ c_2\ \textbf{do}\ s_2 \longrightarrow_{ctrl} \textbf{while}\ c\ \textbf{do}\ s}$$

$$\text{WHILE2}\ \frac{s_1 \parallel s_2 \longrightarrow_{ctrl} s}{\textbf{while}\ c_1\ \textbf{do}\ s_1 \parallel \textbf{while}\ c_2\ \textbf{do}\ s_2 \longrightarrow_{ctrl} \textbf{while}\ c_1 \parallel c_2\ \textbf{do}\ s}$$

$$\text{IF1.1}\ \frac{c_1 \parallel c_2 \longrightarrow_{ctrl} c \qquad s_1 \parallel s_2 \longrightarrow_{ctrl} s \qquad s_1' \parallel s_2' \longrightarrow_{ctrl} s'}{\textbf{if}\ c_1\ \textbf{then}\ s_1\ \textbf{else}\ s_1' \parallel \textbf{if}\ c_2\ \textbf{then}\ s_2\ \textbf{else}\ s_2' \longrightarrow_{ctrl} \textbf{if}\ c\ \textbf{then}\ s\ \textbf{else}\ s'}$$

$$\text{IF1.2}\ \frac{c_1 \parallel c_2 \longrightarrow_{ctrl} c \qquad s_1' \parallel s_2' \longrightarrow_{ctrl} s'}{\textbf{if}\ c_1\ \textbf{then}\ s_1\ \textbf{else}\ s_1' \parallel \textbf{if}\ c_2\ \textbf{then}\ s_2\ \textbf{else}\ s_2' \longrightarrow_{ctrl} \textbf{if}\ c\ \textbf{then}\ s_1 \parallel s_2\ \textbf{else}\ s'}$$

$$\text{IF1.3}\ \frac{c_1 \parallel c_2 \longrightarrow_{ctrl} c \qquad s_1 \parallel s_2 \longrightarrow_{ctrl} s}{\textbf{if}\ c_1\ \textbf{then}\ s_1\ \textbf{else}\ s_1' \parallel \textbf{if}\ c_2\ \textbf{then}\ s_2\ \textbf{else}\ s_2' \longrightarrow_{ctrl} \textbf{if}\ c\ \textbf{then}\ s\ \textbf{else}\ s_1' \parallel s_2'}$$

$$\text{IF2.1}\ \frac{s_1 \parallel s_2 \longrightarrow_{ctrl} s \qquad s_1' \parallel s_2' \longrightarrow_{ctrl} s'}{\textbf{if}\ c_1\ \textbf{then}\ s_1\ \textbf{else}\ s_1' \parallel \textbf{if}\ c_2\ \textbf{then}\ s_2\ \textbf{else}\ s_2' \longrightarrow_{ctrl} \textbf{if}\ c_1 \parallel c_2\ \textbf{then}\ s\ \textbf{else}\ s'}$$

$$\text{IF2.2}\ \frac{s_1' \parallel s_2' \longrightarrow_{ctrl} s'}{\textbf{if}\ c_1\ \textbf{then}\ s_1\ \textbf{else}\ s_1' \parallel \textbf{if}\ c_2\ \textbf{then}\ s_2\ \textbf{else}\ s_2' \longrightarrow_{ctrl} \textbf{if}\ c_1 \parallel c_2\ \textbf{then}\ s_1 \parallel s_2\ \textbf{else}\ s'}$$

$$\text{IF2.3}\ \frac{s_1 \parallel s_2 \longrightarrow_{ctrl} s}{\textbf{if}\ c_1\ \textbf{then}\ s_1\ \textbf{else}\ s_1' \parallel \textbf{if}\ c_2\ \textbf{then}\ s_2\ \textbf{else}\ s_2' \longrightarrow_{ctrl} \textbf{if}\ c_1 \parallel c_2\ \textbf{then}\ s\ \textbf{else}\ s_1' \parallel s_2'}$$

$$\text{WHILE-IF}\ \frac{\textbf{while}\ c_1\ \textbf{do}\ s_1 \parallel s_2 \longrightarrow_{ctrl} s}{\textbf{while}\ c_1\ \textbf{do}\ s_1 \parallel \textbf{if}\ c_2\ \textbf{then}\ s_2 \longrightarrow_{ctrl} \textbf{if}\ \text{true} \parallel c_2\ \textbf{then}\ s}$$

$$\text{IF-WHILE}\ \frac{s_1 \parallel \textbf{while}\ c_2\ \textbf{do}\ s_2 \longrightarrow_{ctrl} s}{\textbf{if}\ c_1\ \textbf{then}\ s_1 \parallel \textbf{while}\ c_2\ \textbf{do}\ s_2 \longrightarrow_{ctrl} \textbf{if}\ c_1 \parallel \text{true}\ \textbf{then}\ s}$$

Figure 4.10: *ctrl* rewriting system

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P_1^1 \parallel P_2^1 \longrightarrow_{eq} P^1}\text{ID.STMT}\quad \dfrac{}{P_1^2 \parallel P_2^2 \longrightarrow_{eq} P^2}\text{ID.STMT}\quad \dfrac{}{P_1^3 \parallel P_2^3 \longrightarrow_{eq} P^3}\text{ID.STMT}\quad \dfrac{}{P_1^{17} \parallel P_2^{17} \longrightarrow_{eq} P^{17}}\text{ID.STMT}}{P^{17} \parallel P_2^{5..17} \longrightarrow_{eq} (\mathbf{skip} \parallel P_2^{5..16}); P^{17}}\text{SEQ3.3}}{P^{17} \parallel P_2^{4..17} \longrightarrow_{eq} (\mathbf{skip} \parallel P_2^{4}); (\mathbf{skip} \parallel P_2^{5..16}); P^{17}}\text{SEQ3.3}}{P_1^{8..17} \parallel P_2^{4..17} \longrightarrow_{eq} (P_1^{8..15} \parallel \mathbf{skip}); (\mathbf{skip} \parallel P_2^{4}); (\mathbf{skip} \parallel P_2^{5..16}); P^{17}}\text{SEQ2.2}}{P_1^{7..17} \parallel P_2^{4..17} \longrightarrow_{eq} (P_1^{7} \parallel \mathbf{skip}); (P_1^{8..15} \parallel \mathbf{skip}); (\mathbf{skip} \parallel P_2^{4}); (\mathbf{skip} \parallel P_2^{5..16}); P^{17}}\text{SEQ2.2}}{P_1^{3..17} \parallel P_2^{3..17} \longrightarrow_{eq} P^3 \ ; \ (P_1^{7} \parallel \mathbf{skip}); (P_1^{8..15} \parallel \mathbf{skip}); (\mathbf{skip} \parallel P_2^{4}); (\mathbf{skip} \parallel P_2^{5..16}); P^{17}}\text{SEQ1.1}}{P_1^{2..17} \parallel P_2^{2..17} \longrightarrow_{eq} P^{2..3} \ ; \ (P_1^{7} \parallel \mathbf{skip}); (P_1^{8..15} \parallel \mathbf{skip}); (\mathbf{skip} \parallel P_2^{4}); (\mathbf{skip} \parallel P_2^{5..16}); P^{17}}\text{SEQ1.1}}{P_1 \parallel P_2 \longrightarrow_{eq} \Phi_1}\text{SEQ1.1}$$

Figure 4.11: Derivation tree of $\longrightarrow_{eq}$ for Example 29

niques [174, 87], as this algorithm exhibits slightly better performance than memoized recursion in our experiments.

$\longrightarrow_{eq}$ is sound, as every reduction rule satisfies property 4.1:

$$\forall s_1, s_2 \in stat : \ s_1 \parallel s_2 \longrightarrow_{eq} s \implies \pi_1(s) = s_1 \wedge \pi_2(s) = s_2 \tag{4.1}$$

where we use the version extraction functions π_1 and π_2 defined in Sec. 3.3 of Chapter 3. Note that $\longrightarrow_{eq}$ is also deterministic by construction, as every pair of non comparable rules either operate on different nodes of the AST, or have incompatible sets of assumptions. In addition, $\longrightarrow_{eq}$ always terminates (or blocks), as every reduction rule R enjoys the following property: to reduce some term t, R requires the reductions of only finitely many terms t', such that t' is strictly smaller than t. $\longrightarrow_{eq}$ is not complete, but $\longrightarrow^{?}_{eq}$ complements it, so the first step of our heuristic does not block. Consequently, the first step of our heuristic is a terminating algorithm.

Remark 24 (Algorithm cost). The cost of this algorithm is dominated by the computations of longest common subsequences of statements performed when merging two sequence of statements s_1 and s_2. As noted in Remark 23 we rely on dynamic programming techniques that iterate over 2-dimensional arrays of size $n_1 \times n_2$, where $n_i = |s_i|$. Such a scheme features quadratic asymptotic complexity. Also note that the overall cost of the overall **merge_stmt** heuristic is quadratic, as these longest common subsequences computations dominate the algorithm cost both theoretically and practically.

Second step: merging sequences of top-level double statements

$\longrightarrow_{glue}$, defined on Fig. 4.9, features a single rule GLUE. GLUE reduces a sequence of pairs to a pair of sequences. $\longrightarrow_{glue}$ is derministic, incomplete and terminating. $\longrightarrow^{\omega}_{glue}$

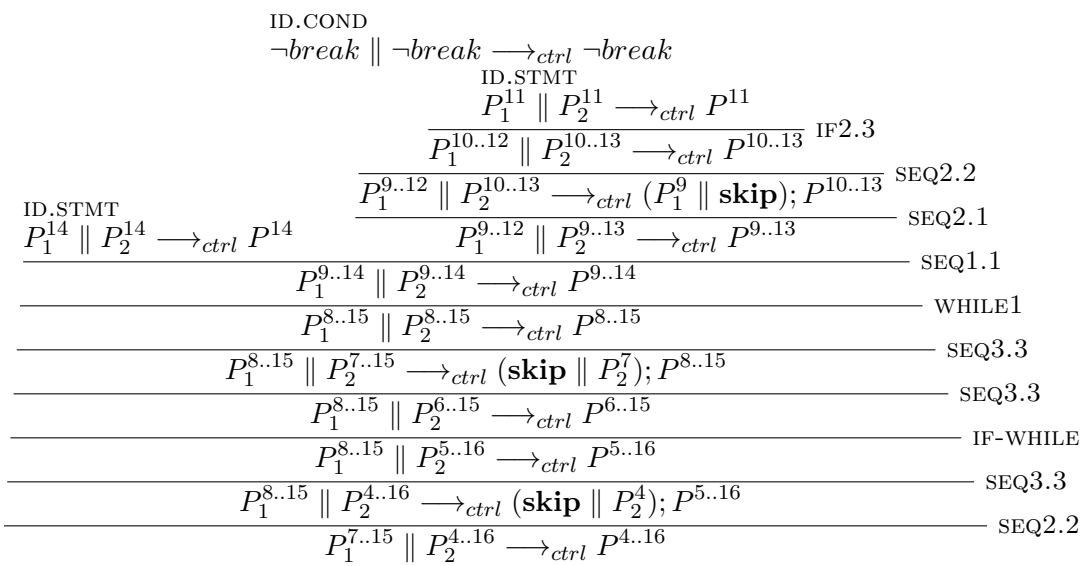


Figure 4.12: Derivation tree of $\longrightarrow_{ctrl}$ for Example 29

iterates $\longrightarrow_{glue}$ until reduction blocks, and returns the final irreducible term. This process terminates, as the number of $\parallel$ nodes in the AST decreases strictly at every iteration.

In the case of Example 29, $\Phi_1 \longrightarrow_{glue}^{3} \Phi_2 \not\longrightarrow_{glue}$ is a derivation for the greedy merge introduced in Sec. 4.2.1.

Third step: aligning similar control structures

The $\longrightarrow_{ctrl}$ rewrite system is defined by two sets of reduction rules: rules defined by Fig. 4.6 (with $r = ctrl$), and rules defined by Fig. 4.10. The priority ordering between rules is defined by Fig. 4.7: rules from Fig. 4.6 are shown in blue, while rules from Fig. 4.10 are shown in black. The goal of $\longrightarrow_{ctrl}$ is to merge statements with similarities, in particular control structures such as loops and tests. To this aim, rules WHILE1 and WHILE2 merge loops if their bodies can be merged, rules IF1.1 to IF2.3 merge tests if one of the branches can be merged, and rule DASS merges assignments to identical targets. All these rules are supported by the axioms ID.EXP and ID.COND, of maximum priority, which state that every pair of syntactically equal simple expressions (resp. conditions) is merged into a shared simple expression (resp. condition). ID.EXP and ID.COND are omitted from Fig. 4.7 for clarity. Additional rules can be added, to represent further similarities between program versions, despite refactoring of control structures. For instance, IF-WHILE and WHILE-IF merge dissimilar control structures that include similar loops. Our implementation features more of such additional rules, which are omitted for conciseness.

Note that rule WHILE-IF is necessary for our algorithm to construct the double program P shown on Fig. 4.2(c) for Example 29. Indeed, it is used by the derivation tree

for the reduction $s'_1 \parallel s'_2 \longrightarrow_{ctrl} P^{4..16}$ introduced in Sec. 4.2.1. This tree is shown on Fig. 4.12. Note that $P^{4..16}$ is slightly rearranged with respect to the exact output of $\longrightarrow_{ctrl}$, for the sake readability. For instance, the exact output of $\longrightarrow_{ctrl}$ for P^4 is $(i \leftarrow 0 \parallel \textbf{skip}); (\textbf{skip} \parallel out \leftarrow (last < first));$. Such rearrangements have no impact on our subsequent static analysis.

Unlike that of $\longrightarrow_{eq}$, the reduction rules of $\longrightarrow_{ctrl}$ do not satisfy property 4.1. Because of rules IF-WHILE and WHILE-IF, they only satisfy the weaker property:

$$\begin{array}{rcl} \forall s_1, s_2 \in stat : s_1 \parallel s_2 \longrightarrow_{eq} s & \Longrightarrow & \mathit{flatten}(\pi_1(s)) = \mathit{flatten}(s_1) \\ & \wedge & \mathit{flatten}(\pi_2(s)) = \mathit{flatten}(s_2) \end{array}$$

where *flatten* is defined by induction on the syntax, and greedily reduces to s any simple sub-statement of the form **if** true **then** s. This property is sufficient for soundness. In addition, $\longrightarrow_{ctrl}$ is deterministic, incomplete, and always terminates or blocks, for the same reasons as $\longrightarrow_{eq}$. $\longrightarrow^{?}_{ctrl}$ complements $\longrightarrow_{ctrl}$, so the third step of our heuristic does not block. Consequently, our heuristic is a terminating algorithm.

4.3 Related works

Multiple independent works on translation validation, patch analysis, security analysis and relational verification rely on product programs, a general class of constructions which informally transform two programs P_1 and P_2 into a single program P that captures the behaviors of P_1 and P_2. Our own double program structure is a form of product program.

Some works, like ours, use product programs as part of a two-step analysis: first construct the product program, then perform an analysis on the result of the construction. In contrast, other works construct product programs during the analysis, in order to guide the construction with semantical information. We term the former approaches "static product constructions" and the latter approaches "dynamic product constructions". In this section, we report on related works on both approaches.

4.3.1 Static product constructions

Various kinds of product constructions have been introduced by multiple authors: self-compositions, cross-products, program products, biprograms and correlating programs. We present them briefly below, and compare them with our approach.

Self-composition

[21] introduced self-composition, as part of a proof method for information flow properties. Given a simple programs P, they prove safety properties of the simple program $P_1; P_2$, where P_k denotes program P with variables x renamed to x_k to avoid conflicts. Their method is sound and relatively complete. Yet, their method is equivalent to analyzing $P_1 \parallel P_2$ in our setting. As demonstrated on Example 28 of Sec. 4.1, this requires computing the input-output relations of P_1 and P_2 before comparing them, *i.e.* their

concrete semantics, which cannot be automated. [169] showed limitations to the practical usability of this method with of-the-shelf software verification tools, and proposed a type-based generalization. Yet their approach does not reduce the expressiveness of the invariants required for the proofs.

Cross-product

[175] introduced the notion of cross-product of two input programs in the context of translation validation of compiler optimizations in unstructured languages. Their approach is limited to so-called "consonant" programs, i.e. programs exhibiting equivalent control structures, and where executions follow the same control flows (no unstable tests).

General product programs

[18, 20] define a more general notion of product program, that overcomes the limitations of self-composition or cross-products, and formalize rules for constructing valid product programs. Their product program structure supports combinations of so-called "synchronous steps", in which statements of P_1 and P_2 are executed in lockstep, and "asynchronous steps", in which statements of P_1 and P_2 are executed separately. The former is supported by simple statements with stable tests in our setting, while the latter is supported by double statements or unstable branches. Their system includes a rule called refinement (or unfolding) that allows preliminary transformations over the components of the product. Simple transformations such as our IF-WHILE and WHILE-IF reduction rules are inspired by this approach. Their approach allows to build products between programs with different control flow structures, so as to address complex loop optimizations. Yet, their product construction is non-deterministic and usually interactive, in contrast to our approach. The authors acknowledge that developing methods and tools for building products is an important goal for further work.

Product programs are also used in SymDiff [111, 112].

Biprograms

[16] introduce biprograms, a variant of product programs that are syntactically similar to our double programs. Biprograms are designed to support relational reasoning using deductive methods. They make explicit the reasoner's choice of program alignments, using a "weaving relation" that rewrites biprograms to equivalent forms, while aligning control points. This weaving relation has connections with the term-rewriting systems we presented in Sec. 4.2.2. Yet, biprogram construction is not automated, in contrast to our work. Their approach has some setbacks: the semantics of the language must be kept deterministic, and procedures with relational specifications must be called by both executions. Our approach does not have these limitations. Note that our treatment of C function calls will be explained in chapter 5.

Correlating programs

[153] present an algorithm for constructing a kind of product program, coined "correlating program". This algorithm takes two program versions as inputs, and processes them in two steps:

1. both program versions are compiled to a guarded command language. Every block is broken down into a set of atomic statements prefixed by a guard encoding the path to the block.
2. guarded commands of both versions are interleaved into a common correlating program. The interleaving is based on a textual diff.

[78] report a soundness issue in this construction, due to the textual diff not taking unstructured control flow into account. They propose an improved algorithm, which they formalize and check mechanically within the `Coq` proof assistant. While [153] is as source of inspiration for our own work, our double program construction does not have a similar issue, as it proceeds by merging abstract syntax trees.

4.3.2 Dynamic product constructions

Speculative correlation

[154] is a follow-up paper to [153]. In this work, the authors move away from their original correlating program structure that defines a fixed interleaving of statements for the subsequent analysis. Instead, they let the analysis alternate between multiple interleavings, and use a speculative search algorithm to choose an interleaving minimizing abstract semantic difference. This approach benefits from semantic information to guide the alignment of program versions, reducing the need for syntactic similarity of program versions. Yet it is costly, as the exploration process involves performing all possible analyses, in all possible interleavings, of pairs of sequences of statements with lengths below some parameter of the analysis. Note that analyses are run with an expressive abstract domain – a power-set domain similar to polyhedra with trace partitioning.

Implicit product programs

[164] present a logic, and a related automatic verification algorithm. This algorithm constructs multiple partial product programs implicity, rather than constructing a single full product program explicitly. Yet they only align syntactically equal statements, as they target hyperproperties of simple programs.

LLRêve [107] relies on program transformations to align two program versions, rather than constructing a product program explicitly. It especially focuses on harmonizing the structures of unbounded loops. To this aim, its exploits dynamic analyses of concrete executions traces to suggest suitable program transformations.

Product-CFGs

A line of work targeted at translation validation rely on SMT-solving to construct product programs. Programs are represented at control-flow graphs (CFGs), and product programs are product-CFGs.

[39] rely on a data-driven approach to align a source program P_1 and compiled program P_2. They first guess an "alignment predicate" (AP) that must hold at all nodes of the product-CFG to build. Then, they construct a candidate product-CFG, using concrete execution traces of P_1 and P_2 (on equal inputs). Execution traces are used to find pairs of program points satisfying the guessed AP, and correlated transitions between these pairs. The obtained candidate product-CFG is called a "Program Alignment Automaton" (PAA). The PAA accepts the available concrete traces by construction. This approach exploits semantic information to find correlations that do not depend on syntactic similarity. In addition, it allows to generate multiple candidate PAAs, on which proof of equivalence is attempted. As a consequence, it can successfully prove the correctness of vectorization optimizations, such as Example 30 of Sec. 4.1. Yet it has shortcomings: it requires execution traces with sufficient path coverage of P_1 and P_2, and relies on a good guess for the AP: a weak AP may result in a large number of candidate PAAs, most of which would not allow for a proof of equivalence, while too strong an AP may rule out the desired PAA. In addition, considering this work targets pairs of small programs from compiler test suites [121], it is not clear to us why guessing "good" APs should be significantly easier than building a product programs by hand. This approach was successfully evaluated on single-loop source programs with no control flow within loop bodies. AP synthesis for programs with multiple loops is left for future work.

[59] addresses the shortcomings of [39]. They build a product-CFG incrementally, through counterexample-guided pruning of the exponentially large search space. They do not rely on concrete execution traces, but generate concrete machine states through SMT queries. Pairs of concrete machines states are used to add new nodes to the product-CFG. They also rely on SMT-solvers to find optimal correlations across sets of paths, and add new edges to the product-CFG. This approach achieves robust and efficient equivalence checking across vectorizing transformations in the presence multiple loops. Yet it is fitted towards translation validation purposes. In particular, it is not symmetric: exchanging P_1 and P_2 may give different results. As a consequence, their algorithm fails in some cases from patch analysis where our simpler approach based on syntactic patterns succeeds. In addition, their approach is costly: their tool runs in minutes on benchmarks of a few tenths of lines of assembly code. It thus cannot be used directly for patch analysis.

4.4 Conclusion

In this chapter, we presented the **merge_stmt** algorithm. **merge_stmt** constructs a double program P from two program versions P_1 and P_2. We used the NIMP and NIMP$_2$ syntax to describe **merge_stmt** in Sec. 4.2, in order to keep the presentation simple. Yet our implementation is a front-end to the analysis of double C programs, which will be presented in chapter 5. **merge_stmt** will thus be evaluated as part of chapter 5. We will show that in most pratical patch cases, the double program constructed by **merge_stmt** is sufficient to enable a successful patch analysis relying on linear invariants only.

Chapter 5

Implementing patch analysis with Mopsa

Chapters 3 and 4 have introduced patch analyses for a purely numerical toy language, NIMP$_2$. In this chapter, we extend this work to the analysis of realistic patches of low-level C programs. We also extend the scope of the analysis to patches of data structures, *e.g.* adding, removing or permuting fields of C structs, changing the lengths of arrays, etc. Such patches are typically syntactically small, but they may have a large semantic impact, as they may change the semantics of any statements involving data of a patched type. In addition, they may be viewed as a way to address a first kind of portability property: robustness to variations of the offsets of scalar fields, such as those introduced by changes in the Application Binary Interface (ABI), compiler options or language extensions such as attributes of types or variables.

We describe our implementation on top of MOPSA, a framework which supports the modular design of static analyzers for multiple programming languages, among which C. Our implementation relies on distinctive features of MOPSA that enable highly modular development of abstract interpreters. We therefore start with a presentation of the MOPSA platform in Sec. 5.1. In particular, MOPSA enabled us to reuse directly most of the implementation of the analysis of simple C programs into our analysis of double C programs. We therefore introduce the analysis of simple C programs with MOPSA in Sec. 5.2. This section stresses the presentation of the cell-based memory model originally introduced in [127], and used by MOPSA to analyze precisely low-level programs that abuse unions and pointers to bypass the type system of C. Note that this memory model will be extended in later chapters (Chapters 6 and 7). Then, we describe our implementation of patch analysis for C programs in Sec. 5.3. Finally, Sec. 5.4 provides an experimental evaluation of our implementation, Sec. 5.5 describes related works, and Sec. 5.6 concludes.

5.1 The Mopsa platform

During this thesis, I was part of the MOPSA project [134, 97], which aims at the modular development of a family of static analyzers for multiple languages and multiple properties. MOPSA stands for *Modular Open Platform for Static Analysis*. MOPSA supports the design of semantics-based abstract interpreters that operate by induction on the syntax, compute sound program invariants and report run-time errors, undefined behaviors, and uncaught exceptions. A distinctive feature of MOPSA is its highly extensible, modular design: semantic abstractions of numeric values, pointers, objects, control, as well as syntax-driven iterators, are defined in small, reusable domains with loose coupling, that can be combined and reused to analyze widely different programming languages. MOPSA currently supports:

- Universal, a toy-language featuring elementary control structures (loops, conditionals, functions) and an idealized data-type (unbounded integers);
- most of C [100], through Clang's parser, and a contract annotation language [148] inspired by ACSL [58] to model library functions;
- a large subset of Python 3 [77, 139, 137] (using a dedicated parser);
- multi-language programs mixing Python and C [140].

MOPSA is written in OCaml. The analysis of C programs consists in 11,700 LOC, while the analysis of Python takes 12,600 LOC. The analysis of multi-language Python and C programs is implemented with 2,500 additional LOC. All these analyses rely on Universal domains, representing 5,600 LOC, and a common framework consisting in 13,200 LOC. Parsers and utilities account for 19,000 additional lines of code.

In this section, we present the MOPSA platform, closely following previously published descriptions [97, 99, 138].

5.1.1 Extensible syntax

Many static analyzers operate on a predefined intermediate language, rather than the source language of analyzed programs. For instance, Ikos [30], from NASA, relies on LLVM bitcode [113]. Such analyzers may nonetheless support multiple source languages: a front-end is developed for each source language, which translates it into the fixed intermediate language. The goal of this approach is to factor the development of transfer functions for multiple source languages. The number of transfer functions may additionally be minimized, if the design of the intermediate language minimizes the number of nodes of the Abstract Syntax Tree (AST). Moreover, the semantics of elementary nodes may be kept simple. For instance, the Infer [69, 168] static analyzer, from Meta, targets Java and C/C++/Objective-C programs. Input programs are translated statically into a reduced intermediate representation called SIL, using only four instructions[1]. However, this approach has also downsides. In particular, translating to a fixed intermediate language does not preserve the high-level structure of programs, which may be detrimental to the precision or the performance of static analyses. For instance, LLVM forgets

[1] `https://github.com/facebook/infer/blob/v1.1.0/infer/src/IR/Sil.mli#L40`

the sign of integer types, and compilation to bytecode or 3-address code makes static analysis harder [118, 144]: the cost of relational analysis is increased, due to a large number of additional variables, while the precision of non relational analysis is likely to be decreased.

In contrast, MOPSA aims at supporting the development of precise static analyses by induction on the syntax of source programs, taking advantage of their high-level structure to discover relational invariants. Analyses are implemented on top of a single Abstract Syntax Tree (AST), defined using an extensible data-type. It is extended in practice to support new languages, or new language constructs. Any module of a MOPSA analyzer can add variants for syntactic objects: statements, expressions, types, variables, etc. For instance, the abstract syntax for `while` loops of the Universal language is introduced as follows, using OCaml's extensible variant types:

```
type stmt_kind += S_while of expr * stmt
```

Then, the C syntax module defines loops as:

```
type stmt_kind += S_c_for of stmt * expr option * expr option * stmt
               |  S_c_do_while of stmt * expr
```

We do not redefine `while` loops for C, as they share the syntax of Universal loops. However, we add AST nodes for C's `for` and `do-while` loops, to keep them separate as long as possible in the program representation, instead of lowering them to `while` loops in the front-end. Additional AST nodes can be added for other languages, such as Python:

```
type stmt_kind += S_py_for of expr * expr * stmt * stmt
               |  S_py_while of expr * stmt * stmt
```

They feature an additional statement for the `else` clause of Python loops, the semantics of which is very different from that of C's `for` loops, as they can iterate on arbitrary objects.

Adding a support for the double program structure defined in this thesis required only, for the AST part, introducing a new kind of statement node and a new expression node. Indeed, the nodes for double expressions *expr* ‖ *expr* and double statements *stat* ‖ *stat* introduced in Sec. 3.3 are defined as follows:

```
type expr_kind += E_double of expr * expr

type stmt_kind += S_double of stmt * stmt
```

5.1.2 Distributed iterator

Compound statements of the AST are handled by the global iterator of an analyzer. The definition of this iterator can be distributed among multiple modules in MOPSA, just like the AST is. Each module contributing to the global iterator implements a partial `exec` function, which handles a given subset of the AST, and returns `None` for other AST nodes. The central controller of the global iterator calls `exec` functions in turn, until one of them returns a value different from `None`. For instance, the semantics of Universal `while` loops

is defined as $\mathbb{S}[\![\textbf{while } c \textbf{ do } s]\!]X = \mathbb{S}[\![\neg c?]\!](\text{lfp}\, f)$, where $f(X') = X \cup \mathbb{S}[\![s]\!](\mathbb{S}[\![c?]\!]X')$, using the notations of Sec. 2.2.2. Its abstract tranfer function is implemented as part of a U.loops iterator for the Universal language as:

```
let exec stmt man flow = match stmt_kind stmt with
| S_while (c, s) ->
    let f flow' = Flow.join
      flow
      (man.exec s (man.exec (mk_assume c) flow'))
   in Some (man.exec (mk_assume (mk_not c)) (lfp f))

| _ ->
   None (* pass-through to the next domain *)
```

where:

- `flow` represents an abstraction of the set of concrete execution traces. It is organised as a structure that contains the representation of the current abstract state. This structure will be presented with Fig. 5.1;
- `Flow.join` computes the abstract union;
- `mk_not` is a helper function to construct an expression denoting the negation of another expression;
- `mk_assume` is a helper function to construct an **assume** statement (of kind `S_assume`);
- `lfp` implements an accelerated fixpoint computation, as introduced in Sec 2.2.4.

The above `exec` function implements the transfer function for Universal loops by combining the semantics of the loop body `s` with that of conditions `c` and `E_not c`. To this aim, it calls the global iterator `man.exec` of the whole analysis recursively, which will in turn find the proper domain-module to handle these statements. This is the role of the `man` argument of `exec` functions, a *manager* recording the global abstract state. The *manager* is aware of all the domains in the analysis. It dispatches every `exec` call to the appropriate domains.

Now, a C.loops iterator for C programs defines the transfer function of `for` loops by rewriting them to Universal loops at analysis time:

```
let exec stmt man flow = match stmt_kind stmt with
| S_c_for (init, cond, incr, body) ->
   let body' = mk_block [body;incr] in
   let stmt' = mk_block [init; mk_while (cond, body')]
   in Some (man.exec stmt' flow)

| _ ->
   None
```

This way, the C.loops iterator delegates the fixpoint computation to the U.loops iterator.

```
module type DOMAIN =
sig
  (* Lattice operators *)
  type t
  val bottom: t
  val top: t
  val is_bottom: t -> bool
  val subset: t -> t -> bool
  val join: t -> t -> t
  val meet: t -> t -> t
  val widen: 'a ctx -> t -> t -> t

  (* Transfer functions *)
  val init : program -> ('a, t) man -> 'a flow -> 'a flow
  val exec : stmt -> ('a, t) man -> 'a flow -> 'a post option
  val eval : expr -> ('a, t) man -> 'a flow -> 'a eval option
end
```

Figure 5.1: Unified domain signature

5.1.3 Domains

Most static analyzers distinguish iterators from abstract domains. The former iterate on control-flow graphs, or compound statements of ASTs, letting the later handle atomic statements, such as assignments and tests. In constrast, Mopsa uses the same signature for modules implementing iterators, and modules implementing abstract domains.

Unified domain signature

Key elements of this signature are displayed on Fig. 5.1. Each domain defines a local type `t` to represent the set of abstract values it can handle, together with associate lattice operators: `join`, `meet`, etc. The internal abstract state of a domain is private: the representation of type `t` is opaque to other domains. This separation, together with the common signature, aims at modularity: minimizing inter-domain coupling so that domains may be easily plugged in and out.

Nonetheless, domains receive the global abstract state of the analysis as an argument of their local transfer functions. The type of this global abstract state is known only at analysis run-time, therefore it is represented as a type variable `'a` in Fig. 5.1. More specifically, `'a` is a parameter to the signature of functions `init`, `exec` and `eval`, which implement the domain's transfer functions. `init` initializes the domain, while `exec` (resp. `eval`) implements the abstract semantics for statements (resp. expressions). The `init` transfer function returns the global abstract state after local initialization of the domain, while the `exec` (resp. `eval`) transfer function returns the postcondition of a statement (resp. an evaluation of an expression).

These three functions are parameterized by a *flow*, of type `'a flow`. This flow, introduced in Sec. 5.1.2, is a wrapper for the global abstract state. MOPSA iterators proceed by induction on the syntax of programs, using continuations to handle non-local control flow operators such as C's `break` and `goto` statements. Therefore the flow represents the global abstraction as a map from a finite, extensible set of control-flow identifiers, termed *tokens*, to continuations of type `'a`. The flow also stores additional context- and flow-insensitive information that may be used throughout the analysis, such as:

- a report of previously raised alarms. Note that some non-recoverable alarms warn that the analysis of some control flows was interrupted: for example, the effect of writing through an invalid pointer is ignored by the analysis.
- a flow-insensitive analysis context, of type `'a ctx`. This context is used, in particular, to store the name of the currently analyzed function, and to collect a list of constants of the program, to be treated as candidate widening thresholds. This context is thus an argument of `widen` lattice operators. It can also be used to tag abstract states of double programs with information on the current program version, as will be presented in Sec. 5.3.5.

Domains need to handle the global abstract state of the analysis, of type `'a`, in order to access their own abstract element in the global abstraction, apply lattice operations, and execute transfer functions. This is the role of the *manager*, of type `('a, t) man`, which was introduced in Sec. 5.1.2. As shown on Fig 5.2, a manager provides `get` and `set` functions, `join`, `meet` and `widen` operators, as well as `exec` and `eval` transfer functions on the global state.

Iterators as stateless domains

Loop iterators such as C.loops and U.loops are defined using this unified signature, as abstract domains with empty abstract state: `t` is OCaml's primitive type `unit`. Such iterators must be combined to implement the global distributed iterator introduced in Sec 5.1.2. In particular, the policy of finding the first domain that handles a specific statement is an example of domain composition, called sequence. Other domain composition operators, such as reduced and Cartesian products, are discussed in Sect. 5.1.5. A typical analysis instance contains dozens of domains. This is illustrated in Fig. 5.3 for the case of the C analyzer, which we will comment further in Sect. 5.2.5. Note that many domains defined for Universal are reused in the C analysis. Also note that Fig. 5.3 shows a slightly simplified C analysis: for instance, it does not include MOPSA's analysis of stubs for external libraries [97, 147].

5.1.4 Dynamic expression rewriting

The ASTs of analyzed programs undergo multiple transformations at analysis time. For instance, control structures may be rewritten, as already shown in Sec. 5.1.2 for the case of loops. In addition, expressions may also be transformed at analysis time, in order to take maximum advantage of relational information available in the program syntax,

```
type 'a lattice = {
  bottom:    'a;
  top:       'a;
  is_bottom: 'a -> bool;
  subset:    'a ctx -> 'a -> 'a -> bool;
  join:      'a ctx -> 'a -> 'a -> 'a;
  meet:      'a ctx -> 'a -> 'a -> 'a;
  widen:     'a ctx -> 'a -> 'a -> 'a;
}

type ('a, 't) man = {
  get : 'a -> 't;
  set : 't -> 'a -> 'a;

  lattice : 'a lattice;

  exec : stmt -> 'a flow -> 'a post;
  eval : expr -> 'a flow -> 'a eval;
}
```

Figure 5.2: Type of a manager

and in the current abstract state. More precisely, expressions of analyzed programs are not directly evaluated to abstract values in Mopsa, but rather translated into symbolic expressions, to be evaluated further by a sequence of abstract domains. This feature is key to inter-domain communication in Mopsa, and contributes to the precision of relational analyses.

Example 31 (Rewriting pointer expressions to numerical expressions)**.** For instance, consider the assignment

```
*p = *q + 1;
```

where `p` and `q` are pointers to `int`. A domain handling abstract points-to information, such as C.pointers in Fig. 5.3, allows rewriting it as

```
*((char *)&x + offset(p)) = *((char *)&y + offset(q)) + 1;
```

provided inferred points-to information ensures that `p` and `q` point to `x` and `y`, respectively. Then, synthetic variables `offset(p)` (resp. `offset(q)`) may be rewritten to 4 (resp. 8) using information inferred by some Universal numerical domain. Then, a domain in charge of abstracting the memory of C programs, such as C.cells, may rewrite the whole assignment to

$$\langle x, 4, \mathbf{int}\rangle \leftarrow \langle y, 8, \mathbf{int}\rangle + 1$$

using synthetic numerical variables $\langle x, 4, \mathbf{int}\rangle$ and $\langle y, 8, \mathbf{int}\rangle$, termed *cells*. Cells represent contiguous sequences of bytes at some offset in a memory block, interpreted in some type to provide a scalar (integer or pointer) value. Mopsa's cell-based memory abstraction

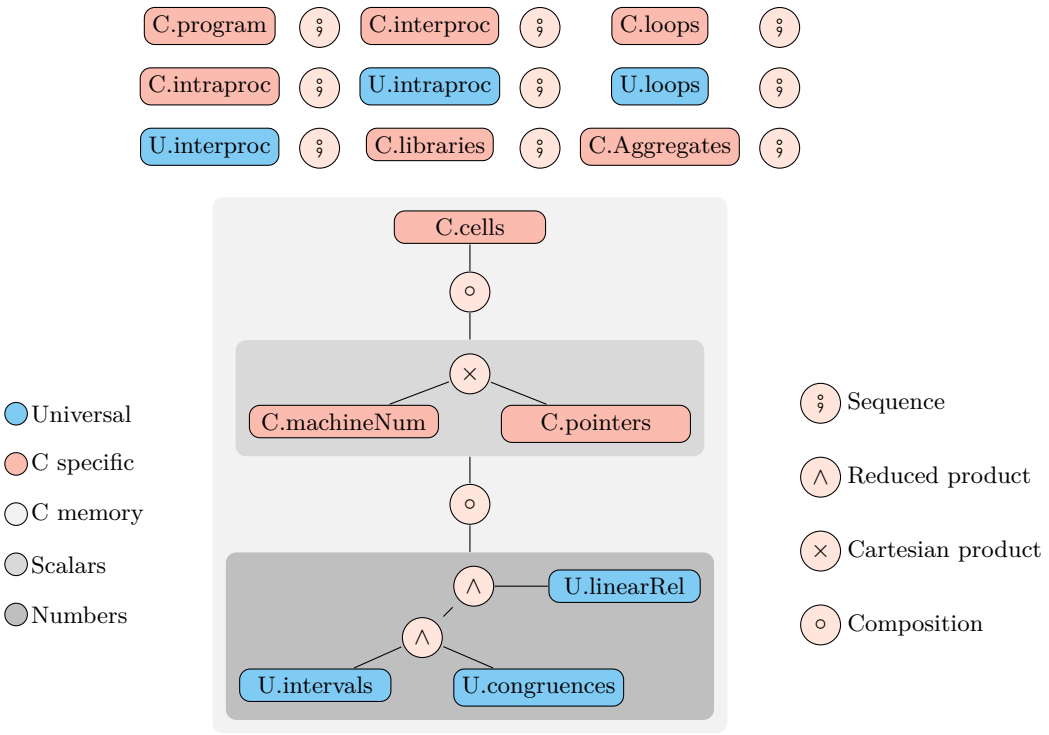


Figure 5.3: Configuration for the analysis of C programs

for C programs will be presented in detail in Sec. 5.2.4. Finally, the right-hand side is rewritten from machine integer arithmetics with implicit wrap-around (in C) to mathematical integer arithmetics (handled by Universal) – by domain C.machineNum. Explicit modular operators are introduced in the translation if the analyzer cannot prove the absence of overflow given the current abstract range of $\langle y, 8, \textbf{int}\rangle$. On the contrary, if the right-hand side can be guaranteed to not overflow, the statement may be fed unchanged to some Universal numerical relational abstract domain operating on unbounded integers, such as polyhedra [51] (U.linearRel), which are able to represent and maintain the linear relation $\langle x, 4, \textbf{int}\rangle = \langle y, 8, \textbf{int}\rangle + 1$.

5.1.5 Domain combination

Domains must be combined in multiple ways to enable rich cooperation schemes, such as the one illustrated in Example 31. To this aim, MOPSA supports several domain combinators, allowing users to construct complex abstractions from elementary domains.

- The sequence D_1 D_2 first executes the transfer functions of domain D_1. It only calls the transfer functions of domain D_2 if D_1 returns `None`, *i.e.* D_1 does not support the AST node of interest. The sequence combinator is primarily used to combine iterators defining transfer functions for disjoint parts of the AST,

```
u8 *p = (u8 *) &V;
for (int i=0; i<sizeof(V); i++)
*p++ = 0;
```

Figure 5.4: Relation between an integer variable and the offset of a pointer

such as loops, function calls, etc. For instance, the sequence C.loops ⨾ U.loops supports C's `for` loop as described in Sec. 5.1.2.

∧ Reduced products are a standard form of cooperation between domains of abstract interpreters [49, 50]. A reduced product computes the intersection of abstract values approximating the same concrete objects in different ways. Transfer functions of D_1 ∧ D_2 call that of both D_1 and D_2, before reductions are performed that allow mutual refinement of abstract values. For instance, consider the classic example of reducing intervals and congruences [132]. Given an interval $[0, 5]$ and a congruence $4\mathbb{Z} + 2$, we can refine both abstract values in two steps. First, using congruence information, the interval is refined to $[2, 2]$. Then, since the interval is now a singleton, the congruence is refined into $0\mathbb{Z} + 2$. As another example, Fig. 5.3 features a reduced product (U.intervals ∧ U.congruences) ∧ U.linearRel between the interval, congruence [82] and polyhedra domains. Note that we use use intervals in addition to polyhedra, although polyhedra are more expressive. This is because intervals may be more precise than polyhedra for some non-linear operations, such as modular wrap-around arithmetic expressions.

× Cartesian products combine domains abstracting orthogonal semantic objects, *e.g.* numeric variables and pointers. For instance, Fig. 5.3 features C.machineNum × C.pointers. These domains are assembled in a Cartesian product as their semantics do not overlap: unlike reduced products, they target orthogonal expressions, hence no reduction can happen between abstract values.

∘ The composition of domains D_1 ∘ D_2 implements the complete mediation of D_1 by D_2: D_1 is a functor domain parameterized by D_2. For instance, Fig. 5.3 shows a composition between C.machineNum × C.pointers and the underlying Universal numerical domain. Indeed, as explained in Example 31, both C.machineNum and C.pointers delegate a part of their abstract state to an underlying numerical domain: integer C variables for the former, and pointer offsets for the latter. A distinctive feature of Mopsa is to let them share this underlying part of their abstract state. This makes it possible for the numerical domain to infer relations between integer variables and pointer offsets, such as the loop invariant $\mathit{offset}(p) = i$ in the snippet of Fig. 5.4.

```
struct { char c; int x; } s = {0};
char *p = (char *) &s;
p[4] = 1;
```

(a) Type puning

	System V alignments	Packed structs
Little-endian	1	2^{24}
Big-endian	2^{24}	1

(b) Final value of `s.x` (32-bit architecture)

Figure 5.5: ABI-dependent C code (Example 3)

5.2 Analysis of C programs

After this overview of distinctive features of the MOPSA platform, we are ready to move on to the implementation of the analysis of C programs shown on Fig. 5.3. This implementation predates the thesis, and was presented in several papers [134, 98, 97, 147]. A key ingredient to the analysis of C, with respect to that of programs written in a toy language such as NIMP, is the memory model. In this section, we therefore start with a presentation of the memory model for C implemented on top of MOPSA, and depicted in light gray on Fig. 5.3. Then, we will present in Sec. 5.2.5 the other domains used by the abstraction of C programs, and shown on Fig. 5.3.

Let us start with a presentation of MOPSA's memory model for C programs. This cell-based memory model was first introduced in [127], and further developed in [131]. In the sequel, we closely follow the presentation from [131, Sec. 5.2].

5.2.1 Motivation

The C standard [91] leaves the encoding of scalar types and the layout of fields in structures partly unspecified. The precise representation of types is standardized in implementation-specific *Application Binary Interfaces* (ABI), such as the System V ABI [11], to ensure the interoperability of compiled programs, libraries, and operating systems. Although it is possible to write fully portable, ABI-neutral C code, the vast majority of C programs rely on assumptions on the ABI of the platform.

We are interested in analyzing not only well-behaved, fully portable C programs, but also software that make explicit use of architecture-dependent features. The case occurs typically with low-level C software, such as device drivers or embedded software, for efficiency reasons. For instance, Fig. 5.5(a) reproduces a snippet of code that was previously presented as part of Example 3 in Sec. 1.1.4. This snippet abuses pointers to bypass the C type system, a common practice in low-level programming known as *type punning*. As a consequence, the final value of `s.x` depends on the offset of field `x` in struct `s` and on the size and representation of type `int`, *i.e.* on the ABI of the platform. The final values of `s.x` for 4 possible 32-bit ABI are shown on Fig. 5.5(b). We refer the reader to Example 3 of Sec. 1.1.4 for a complete presentation.

$$
\begin{array}{lcl}
\textit{int-sign} & ::= & \textbf{signed} \mid \textbf{unsigned} \\
\textit{int-type} & ::= & \textbf{char} \mid \textbf{short} \mid \textbf{int} \\
 & \mid & \textbf{long} \mid \textbf{long long} \\
\textit{scalar-type} & ::= & \textit{int-sign}\ \textit{int-type} \\
 & \mid & \textbf{ptr}
\end{array}
$$

$$
\begin{array}{lcll}
\textit{type} & ::= & \textit{scalar-type} & \\
 & \mid & \textit{type}\,[n] & n \in \mathbb{N} \\
 & \mid & \textbf{struct}\ \{\textit{type}_1, \ldots, \textit{type}_n\} & \\
 & \mid & \textbf{union}\ \{\textit{type}_1, \ldots, \textit{type}_n\} &
\end{array}
$$

$$
\begin{array}{lcl}
\textit{lval} & ::= & *_{\textit{scalar-type}}\ \textit{expr} \\
\circ & ::= & \texttt{-} \mid \texttt{\textasciitilde} \mid (\textit{scalar-type}) \\
\diamond & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{\textasciicircum} \mid \texttt{>>} \mid \texttt{<<}
\end{array}
$$

$$
\begin{array}{lcll}
\textit{expr} & ::= & \textit{lval} & \\
 & \mid & \&V & \\
 & \mid & \textbf{rand}(c_1, c_2) & c_1, c_2 \in \mathbb{Z} \\
 & \mid & \circ\ \textit{expr} & \\
 & \mid & \textit{expr} \diamond \textit{expr} &
\end{array}
$$

$$
\begin{array}{lcl}
\textit{stat} & ::= & \textit{lval} \leftarrow \textit{expr} \\
 & \mid & \textbf{if}\ \textit{cond}\ \textbf{then}\ \textit{stat}\ \textbf{else}\ \textit{stat} \\
 & \mid & \ldots
\end{array}
\qquad
\begin{array}{lcll}
\textit{cond} & ::= & \textit{expr} \bowtie 0 & \bowtie \in \{\leq, \geq, =, \neq, <, >\}
\end{array}
$$

Figure 5.6: Syntax of simple C-like programs.

5.2.2 Syntax

The current implementation on top of Mopsa supports C programs directly. We nonetheless use a simplified C-like language to simplify the presentation of this section. This language is defined as an extension of the Nimp language defined in Sec. 2.1. Its syntax is shown on Fig. 5.6. Statements *stat* are built on top of expressions *expr* and Boolean conditions *cond*. This is a difference with C, which does not distinguish conditions from expressions. Expressions rely on a C-like type-system. Integer and pointer types are collectively referred to as scalar types (floats are omitted from the presentation for simplicity). Expressions support pointer arithmetic, expressed as byte-level offset arithmetic. All left-values are assumed to be pre-processed to dereferences $*_\tau e$ (*i.e.* `*((`τ`*)e)` in C) where τ is a scalar type, and e is a pointer expression.

Remark 25 (Scalar dereferences only). Dereferences are limited to scalar types, and the dereferenced type is explicit in the syntax.

The language allows structured types: arrays of fixed size, structures and unions. For instance, struct `s` of Example 3 features a field `x` of type `int`. Assume the System V ABI is used, for a 32-bit architecture. The offset of `x` in `s` is 4. Any occurrence of the left-value `s.x` in the program is assumed to be pre-processed to $*_{\textbf{int}}\,(\&s + 4)$.

We assume a fixed, finite set $\mathcal{V}$ of program variables, and a function $\textit{typeof} \in \mathcal{V} \to \textit{type}$ that provides their types.

Remark 26 (Extensions controlling the layout of C structs). Our implementation supports standard C extensions (through Clang's parser) used to control the alignment of structure fields, or the packing of structures. Such extensions include local compiler directives, such as type attributes, or global directives, such as `#pragma` directives or compiler options. We assume the relevant layout information is part of the syntax of types (and of the output of the *typeof* function), though we do not reflect this in Fig. 5.6 for brevity.

5.2.3 Semantics of low-level C programs

In this section, we present a semantics that supports unions and pointers, and is able to model the behavior of non-portable C programs such as Example 3 precisely. This semantics relies on a byte-based representation of C variables.

As introduced in Sec. 5.2.1, the semantics of C programs is parameterized by an ABI, which defines the sizes of scalar types, the ordering of bytes in scalar types (endianness), and the offsets of fields in structs. The low-level semantics therefore makes these choices explicit. In the rest of this section, unless otherwise specified, we assume a 32-bit little-endian System V ABI. We assume a function $\mathit{sizeof} \in \mathit{type} \to \mathbb{N}$ given, which provides the sizes of types (in bytes).

Pointer values are modeled as (semi-)symbolic addresses of the form $\langle V, i \rangle \in \mathcal{V} \times \mathbb{Z}$, which indicates an offset of i bytes from the first byte of V. Special pointer values are defined for C's `NULL` and dangling pointers:

$$\mathcal{P}tr \triangleq \mathcal{V} \times \mathbb{Z} \cup \{\mathbf{NULL}, \mathbf{invalid}\}$$

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

$$\mathcal{A}ddr \triangleq \{\, \langle V, o \rangle \mid V \in \mathcal{V} \wedge 0 \leq o < \mathit{sizeof}(\mathit{typeof}(V)) \,\} \subseteq \mathcal{P}tr$$

We rely on the byte-level modeling of memory contents from [131, Sec. 5.2]. Each concrete memory byte has a value in $[0, 255]$ which may be (part of) the representation of a concrete numerical value, or part of the representation of a pointer. As the the precise numeric values of pointers depend on memory allocation strategies outside the scope of the analysis, we use a symbolic representation for pointer values: $\langle p, i \rangle \in \mathcal{P}tr \times \mathbb{N}$ denotes the $i-$th byte in the memory representation of the pointer value p. We thus denote the set of byte values as $\mathbb{B} \triangleq [0, 255] \cup (\mathcal{P}tr \times \mathbb{N})$. Expressions manipulate scalar values, which may be numeric (machine integers) or pointer values. We denote the set of values as $\mathbb{V} \triangleq \mathbb{Z} \cup \mathcal{P}tr$. The definition of the most concrete semantics requires a family of representation functions $\mathit{benc}_\tau \in \mathbb{V} \to \mathcal{P}(\mathbb{B}^*)$, that convert a scalar value of a given type $\tau \in \mathit{scalar\text{-}type}$ into a sequence of $\mathit{sizeof}(\tau)$ byte values. We denote as $\mathit{bdec}_\tau \in \mathbb{B}^* \to \mathcal{P}(\mathbb{V})$ the reverse conversion. For instance, on a 32-bit little-endian platform, $\mathit{benc}_{\mathbf{unsigned\,int}}(1) = \{\, (1,0,0,0) \,\}$, $\mathit{bdec}_{\mathbf{unsigned\,short}}(0,1) = \{\, 2^{16} \,\}$, and $\mathit{benc}_{\mathbf{ptr}}(p) = \{\, (\langle p, 0\rangle, \langle p, 1\rangle, \langle p, 2\rangle, \langle p, 3\rangle) \,\}$. This seemingly trivial encoding allows modeling copying pointer values byte per byte, as done *e.g.* by `memcpy`. Note that the benc_τ and bdec_τ functions return a set of possible values. For instance, reinterpreting a pointer value as an integer, as in $\mathit{bdec}_{\mathbf{int}} \circ \mathit{benc}_{\mathbf{ptr}}(p)$, returns the full range of type **int**. In contrast, reinterpreting an integer value as a pointer returns the singleton $\{\, \mathbf{invalid} \,\}$. We do not detail the definitions of these functions here, for the sake of conciseness. An example may be found in [131, Sec. 5.2].

Concrete byte-level memory states are elements of $\mathcal{M} \triangleq \mathcal{A}ddr \to \mathbb{B}$. The semantics $\mathbb{E}[\![\, \mathit{expr} \,]\!] \in \mathcal{M} \to \mathcal{P}(\mathbb{V})$ and $\mathbb{S}[\![\, \mathit{stat} \,]\!] \in \mathcal{P}(\mathcal{M}) \to \mathcal{P}(\mathcal{M})$ of expressions and statements is defined by standard induction on the syntax. We therefore only show, on Fig. 5.7, the semantics of memory reads and writes $\mathbb{E}[\![\, *_\tau e \,]\!]$ and $\mathbb{S}[\![\, *_\tau e_1 \leftarrow e_2 \,]\!]$. Bytes are fetched

$$\begin{array}{l}\mathbb{E}[\![*_\tau e]\!]\mu \triangleq \{ v \mid \langle V, o\rangle \in \mathbb{E}[\![e]\!]\mu \wedge 0 \leq o \leq sizeof(typeof(V)) - sizeof(\tau) \\ \qquad\qquad \wedge v \in bdec_\tau(\mu\langle V, o\rangle, \ldots, \mu\langle V, o + sizeof(\tau) - 1\rangle) \} \\ \mathbb{S}[\![*_\tau e_1 \leftarrow e_2]\!]M \triangleq \\ \quad \bigcup_{\mu\in M} \{ \mu[\forall i < sizeof(\tau) : \langle V, o + i\rangle \mapsto b_i] \mid \langle V, o\rangle \in \mathbb{E}[\![e_1]\!]\mu \\ \qquad\qquad \wedge 0 \leq o \leq sizeof(typeof(V)) - sizeof(\tau) \wedge (b_0, \ldots, b_{sizeof(\tau)-1}) \in benc_\tau(\mathbb{E}[\![e_2]\!]\mu) \}\end{array}$$

Figure 5.7: Concrete semantics of memory reads and writes.

and decoded with $bdec_\tau$ when reading from memory in expression $*_\tau e$, while values computed by expression e_2 are encoded into bytes with $benc_\tau$ when writing to memory in assignment $*_\tau e_1 \leftarrow e_2$. Note that illegal memory accesses are silently omitted to simplify the presentation.

This byte-based concrete semantics is very expressive. It allows modelling all scalar expressions precisely. This includes both single-byte and multi-byte accesses to memory. For instance, given 2-byte variables `x` and `y` of type `unsigned short`, the transfer function for the assignment `x=y+1;` is

$$\begin{cases} \langle x, 0\rangle &= (\langle y, 0\rangle + 2^8 \times \langle y, 1\rangle + 1) \mod 2^8 \\ \langle x, 1\rangle &= \lfloor ((\langle y, 0\rangle + 2^8 \times \langle y, 1\rangle + 1) \mod 2^{16})/2^8 \rfloor \end{cases}$$

However, it is not advisable to abstract this semantics with numerical domains directly. Indeed, abstracting such transfer functions would require an expressive numerical domain, able to both represent linear equalities and handle modular arithmetic precisely. This would prevent the use of scalable abstract domains such as intervals. As another example, if `x` has type `unsigned short`, while `y` has pointer type, then $\mathbb{E}[\![y]\!]\mu = \mathbb{E}[\![*_{\mathbf{int}} \&y]\!]\mu = bdec_{\mathbf{int}} \circ benc_{\mathbf{ptr}}(y)$ is the full range of type **int**. Hence the transfer function assigns the full range of numeric bytes $[0, 255]$ to both $\langle x, 0\rangle$ and $\langle x, 1\rangle$. As a final example, if `x` and `y` have pointer type, then the individual bytes $\langle x, i\rangle$ of `x` are set to the symbolic representation of the pointer $y + 1$, without loss of precision.

5.2.4 Cell-based memory model

We rely instead on a more abstract semantics, based on the Cells memory model first introduced in [127]. In order to handle C programs computing with machine integers of multiple sizes, with byte-level access to their encoding through type-punning, memory is represented as a dynamic collection of multi-byte scalar variables termed cells. Cells hold values for the scalar memory dereferences discovered during the analysis, without requiring to abstract each byte individually. The Cells memory domain maintains a consistent abstract state despite the introduction of overlapping cells by type-punning.

Memory abstraction

Let $\mathcal{C}ell \subseteq \mathcal{V} \times \mathbb{N} \times$ *scalar-type* denote the finite set of possible scalar dereferences. Each cell $\langle V, o, \tau\rangle \in \mathcal{C}ell$ is denoted as a variable V, an offset o, and a scalar type τ

indicating the encoding of values. The cell $\langle V, o, \tau \rangle$ represents the value of the dereference $*_\tau(\&V + o)$.

$$\mathcal{C}ell \triangleq \{ \langle V, o, \tau \rangle \,|\, V \in \mathcal{V},\ \tau \in \textit{scalar-type},\ 0 \leq o \leq \textit{sizeof}(\textit{typeof}(V)) - \textit{sizeof}(\tau) \}$$

All bytes in a cell are addressable by construction:

$$\forall \langle V, o, \tau \rangle \in \mathcal{C}ell : \forall i < \textit{sizeof}(\tau) : \langle V, o + i \rangle \in \mathcal{A}ddr$$

Domain. We define an abstract memory state as a set of pairs, consisting of a set of cells $C \subseteq \mathcal{C}ell$ and a scalar environment over C. Let $\mathcal{E}$ denote the associated abstract domain:

$$\mathcal{E} \triangleq \bigcup_{C \subseteq \mathcal{C}ell} \{ \langle C, \rho \rangle \,|\, \rho \in C \to \mathbb{V} \}$$

An abstract property $X \in \mathcal{P}(\mathcal{E})$ represents the set of concrete byte-level memory states $\gamma_{\mathcal{C}ell}(X) \in \mathcal{P}(\mathcal{M})$ satisfying environment constraints over cells. The values of the bytes of these memories must satisfy all of the numerical constraints implied by the scalar environment over some set of cells:

$$\gamma_{\mathcal{C}ell}(X) \triangleq \left\{ \mu \in \mathcal{M} \;\middle|\; \begin{array}{l} \exists (C, \rho) \in X : \forall \langle V, o, \tau \rangle \in C : \\ \exists (b_0, \ldots, b_{\textit{sizeof}(\tau)-1}) \in \textit{benc}_\tau(\rho \langle V, o, \tau \rangle) : \\ \forall i < \textit{sizeof}(\tau) : \mu \langle V, o + i \rangle = b_i \end{array} \right\}$$

This semantics captures the intuition that union types and type punning provide several typed views of the same underlying sequences of bytes, and that all these views are compatible. The lattice of properties $(\mathcal{P}(\mathcal{E}), \preccurlyeq, \cup)$ is equipped with partial order

$$X \preccurlyeq X' \iff^{\triangle} \forall (C, \rho) \in X : \exists (C', \rho') \in X' : C' \subseteq C \wedge \rho' = \rho_{|C'}$$

where $\rho_{|C'}$ denotes the restriction of ρ to C'.

Note that [127] and [131, Sec. 5.2] use a slightly more abstract domain. They define abstract states as a choice of a set of cells $C \subseteq \mathcal{C}ell$ and a set of scalar environments on C. Our definition is more flexible, as it allows retaining several environments with heterogenous support (cell set). This will ease the extension of the memory model to double programs in Chapters 6 and 7.

Transfer functions: cell addition and removal. Removing any cell is always sound: it amounts to losing information. It is also possible to add new cells, provided the values assigned to them are consistent with those of existing overlapping cells. This consistency is ensured by a value synthesize function $\phi \in \mathcal{C}ell \to \mathcal{P}(\mathcal{C}ell) \to \textit{expr}$ such that $\phi(c)(C)$ returns a syntactic expression denoting (an abstraction of) the value of the cell c as a function of cells in C.

An example implementation is proposed in Fig. 5.8. Firstly, if the cell already exists ($c \in C$), it is directly returned by ϕ. Otherwise, ϕ looks for integer cells of the same size and different signedness, and converts them using function *wrap* to model wrap-around,

$$\phi\langle V,o,t\rangle(C) \triangleq \begin{cases} \langle V,o,t\rangle \textbf{ if } \langle V,o,t\rangle \in C \\ \mathit{wrap}(\langle V,o,t'\rangle, \mathit{range}(t)) \textbf{ else if } \langle V,o,t'\rangle \in C \wedge t,t' \in \mathit{int\text{-}type} \wedge \mathit{sizeof}(t) = \mathit{sizeof}(t') \\ \mathit{byte}(\langle V,o-b,t'\rangle, w(\mathcal{L},b,\mathit{sizeof}(t'))) \\ \qquad\qquad \textbf{else if } \langle V,o-b,t'\rangle \in C \wedge\ t = \textbf{u8} \wedge t' \in \mathit{int\text{-}type} \wedge b < \mathit{sizeof}(t') \\ \mathit{wrap}(\sum_{i=0}^{\mathit{sizeof}(t)-1} 2^{8\times w(\mathcal{L},i,\mathit{sizeof}(t))} \times \langle V,o+i,\textbf{u8}\rangle, \mathit{range}(t)) \\ \qquad\qquad \textbf{else if } \forall i < \mathit{sizeof}(t) : \langle V,o+i,\textbf{u8}\rangle \in C \wedge t \in \mathit{int\text{-}type} \\ \mathit{range}(t) \qquad \textbf{else if } t \in \mathit{scalar\text{-}type} \\ \textbf{invalid} \qquad \textbf{else if } t = \textbf{ptr} \end{cases}$$

Figure 5.8: Generic cell synthesizing function.

and function *range* for the range of the type:

$$\mathit{wrap}(v,[l,h]) \triangleq \min\{\, v' \in [l,h] \mid \exists k \in \mathbb{Z} : v = v' + k(h-l+1) \,\}$$

$$\mathit{range}(t) \triangleq \begin{cases} [0, 2^{8\times \mathit{sizeof}(t)} - 1] & \text{if } t \text{ is unsigned} \\ [-2^{8\times \mathit{sizeof}(t)-1}, 2^{8\times \mathit{sizeof}(t)-1} - 1] & \text{if } t \text{ is signed} \end{cases}$$

If no matching cell is found and $t = \textbf{u8} \triangleq$ `unsigned char`, ϕ attempts to extract an unsigned byte from an existing integer cell. If this does not work, ϕ aggregates unsigned bytes into integers. A final wrap-around is used to support negative numbers and signed types. Function $w \in \{\, \mathcal{L}, \mathcal{B} \,\} \times \mathbb{N}^2 \to \mathbb{N}$ is used to model the endianness-dependent weight of bytes in integers: $w(\mathcal{L},b,s) \triangleq b$ for little-endian encoding, and $w(\mathcal{B},b,s) \triangleq s-b-1$ for big-endian encoding. Recall that we only consider a little-endian platform for now: therefore ϕ synthesizes cells with little-endian encoding only. The value of the byte of weight 2^{8w} in an unsigned integer x is: $\mathit{byte}(x,w) = \lfloor x/2^{8w} \rfloor \mod 2^8$. When all fails, ϕ returns the full range of the type (or **invalid**, for a pointer).

Cell addition, $\mathit{add\text{-}cell} \in \mathcal{C}ell \to \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$, then simply adds the cell and initializes its value using the ϕ function:

$$\mathit{add\text{-}cell}(c)(X) \triangleq \{\, \langle C \cup \{c\}, \rho[c \mapsto v]\rangle \mid v \in \mathbb{E}[\![\, \phi(c)(C) \,]\!]\rho,\ (C,\rho) \in X \,\}$$

Many definitions are possible for ϕ, e.g. adding cases to support floats, or to synthesize integer cells from cells of opposite endianness. Note that the latter extension will be addressed in Chapter 7. Also, some of the above patterns of ϕ for integer cells may be extended for pointers cells, e.g. extracting 1-byte cells from existing pointer cells, and aggregating existing 1-byte cells into a pointer cell. The choice of the memory patterns supported by ϕ is part of a trade-off between the efficiency and the precision of the subsequent analysis. The sole requirement for soundness is that *add-cell* should over-approximate the identity function:

$$\forall X \in \mathcal{P}(\mathcal{E}),\ c \in \mathcal{C}ell :\ \gamma_{\mathcal{C}ell}(\mathit{add\text{-}cell}(c)(X)) \supseteq \gamma_{\mathcal{C}ell}(X)$$

In practice, we have equality. Indeed the converse inclusion holds, as states of $\mathit{add\text{-}cell}(c)(X)$ feature more cells and constraints than states of X.

$$(\mathcal{P}(\mathcal{E}), \preccurlyeq) \underset{\alpha^\flat}{\overset{\gamma^\flat}{\leftrightarrows}} (\mathcal{E}^\flat, \preccurlyeq^\flat)$$

$$\alpha^\flat(X) \triangleq \langle \bar{C}, \{\, \rho_{|\bar{C}} \mid \langle C, \rho\rangle \in X \,\}\rangle \text{ where } \bar{C} = \bigcap\{\, C \mid \langle C, \rho\rangle \in X \,\}$$

$$\gamma^\flat\langle C, R\rangle \triangleq \{\, \langle C, \rho\rangle \mid \rho \in R \,\} \text{ and } \langle C, R\rangle \preccurlyeq^\flat \langle C', R'\rangle \overset{\triangle}{\iff} C' \subseteq C \wedge \{\, \rho_{|C'} \mid \rho \in R \,\} \subseteq R'$$

Figure 5.9: Unified cell environments

Transfer functions: assignments and tests. The semantics of C programs running on a 32-bit, little-endian, System V platform is given in [131, Sec. 5.2]. We briefly sketch transfer functions $\mathbb{E}[\![\, expr \,]\!] \in \mathcal{E} \to \mathcal{P}(\mathbb{V})$ and $\mathbb{S}[\![\, stat \,]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ below.

Expressions are first transformed into purely scalar expressions by resolving left-values bottom up. More precisely, any left-value $*_\tau e$ where e does not contain any dereference is transformed into a cell set by: evaluating e into a set of pointer values P, gathering the set of cells L corresponding to valid pointers in P, and realizing all the cells in L using *add-cell*: $L \triangleq \{\, \langle V, o, \tau\rangle \in \mathcal{C}ell \mid \langle V, o\rangle \in P \,\} \subseteq \mathcal{A}ddr$.

The semantics of cell sets $\mathbb{E}[\![\, \{\, c_1, \ldots, c_n \,\} \,]\!]\rho$ appearing in expressions boils down to standard scalar evaluation $\{\, \rho(c_1) \ldots \rho(c_n) \,\}$. The semantics of assignments $\mathbb{S}[\![\, \{\, c_1, \ldots, c_n \,\} \leftarrow e \,]\!]$ is more involved. Because of the conjunctive semantics, it is not sufficient to update the value of $c_1, \ldots, c_n$ using standard scalar transfer functions $\mathbb{S}[\![\, c_i \leftarrow e \,]\!]$. It is also necessary to update the values of any cells overlapping $c_1, \ldots, c_n$. A simple and efficient solution is to remove them. The semantics of other statements is standard.

Unification

Like in [127] and [131, Sec. 5.2], we aim at abstracting $\mathbb{S}$ further, using numerical domains. Yet, this cannot be done directly, as states have heterogeneous cell support, while a numerical abstract element naturally represents a set of environments with homogenous support. We therefore unify cell sets first. To this aim, we only retain cells which are part of every state in a set, soundly disregarding all others. Our new domain is thus a choice of a set of cells C and a set of scalar environments on C:

$$\mathcal{E}^\flat \triangleq \bigcup_{C \subseteq \mathcal{C}ell} \{\, \langle C, R\rangle \mid R \in \mathcal{P}(C \to \mathbb{V}) \,\}$$

A formalization is shown on Fig. 5.9.

Proposition 10. *The pair $(\alpha^\flat, \gamma^\flat)$ defined in Fig. 5.9 is a Galois connection.*

The adaptation of *add-cell* $\in \mathcal{C}ell \to \mathcal{E}^\flat \to \mathcal{E}^\flat$ to this new domain is straightforward. Adaptations of transfer functions $\mathbb{S}^\flat[\![\, stat \,]\!] \in \mathcal{E}^\flat \to \mathcal{E}^\flat$ are also straightforward, assuming a sound abstract join is provided.

Yet, such a join must merge environment sets defined on heterogeneous sets of cells. We therefore define a unification function *unify* $\in \mathcal{E}^\flat \times \mathcal{E}^\flat \to \mathcal{E}^\flat \times \mathcal{E}^\flat$. $(\langle C_1', R_1'\rangle, \langle C_2', R_2'\rangle) =$ *unify*$(\langle C_1, R_1\rangle, \langle C_2, R_2\rangle)$ adds, with *add-cell*$^\flat$, any missing cells to $\langle C_1, R_1\rangle$ and $\langle C_2, R_2\rangle$:

respectively $C_2 \setminus C_1$ and $C_1 \setminus C_2$. Thus $C_1' = C_2' = C_1 \cup C_2$. The abstract join may now be defined as:

$$\langle C_1, R_1 \rangle \sqcup^\flat \langle C_2, R_2 \rangle \triangleq \langle C_1' \cup C_2', R_1' \cup R_2' \rangle$$

Pointer and numerical abstractions

We finally rely on numerical abstractions to abstract further $\mathbb{S}^\flat$ into a computable abstract semantics $\mathbb{S}^\sharp$, resulting in an effective static analysis.

Like in [131, Sec. 5.2], we assume that we are given, for each set of cells $C \subseteq \mathcal{C}ell$, an abstract domain $\mathcal{E}_C^\sharp$, with concretization γ_C. It abstracts $\mathcal{P}(C \to \mathbb{Z}) \simeq \mathcal{P}(\mathbb{Z}^{|C|})$, *i.e.*, sets of points in a $|C|-$dimensional vector space. A cell of integer type naturally corresponds to a dimension in an abstract element. We also associate a distinct dimension to each cell of pointer type; it corresponds to the offset o of a symbolic pointer $\langle V, o \rangle \in \mathcal{P}tr$. In order to abstract fully pointer values, we enrich numerical abstract environments with a map P associating to each pointer cell the set of variables it may point to. Hence, the abstract domain becomes:

$$\mathcal{E}^\sharp \triangleq \{ \langle C, R^\sharp, P \rangle \mid C \subseteq \mathcal{C}ell,\ R^\sharp \in \mathcal{E}_C^\sharp,\ P \in P_C \to \mathcal{P}(\mathcal{V} \cup \{ \mathbf{NULL}, \mathbf{invalid} \}) \}$$

where $P_C \subseteq C$ is the subset of cells of pointer type. We refer to [131, Sec. 5.2] for a formal presentation of the concretization and abstract operators.

5.2.5 Analysis of C programs with Mopsa

After this presentation of the memory model, we are now ready to describe the current implementation of the analysis of C programs on top of the Mopsa infrastructure. Fig. 5.3 shows a configuration of Mopsa for the analysis of C programs. This analysis aims at inferring invariant properties of reachable program states, and prove the absence of run-time errors, such as arithmetic overflows, invalid memory dereferences, and failed assertions.

Mopsa first parses the source files using a front-end based on Clang, and converts the AST to OCaml, keeping the C syntax and type information at a high-level. The individual ASTs corresponding to each compilation unit (compiled C file) are then linked, *i.e.* merged into a single AST by resolving symbol definitions. Then, the analyzer is called on the `main` entry-point, using an abstraction defined by the combination of abstract domains shown on Fig. 5.3. Many other configurations are possible, and existing configurations may evolve as new abstractions are introduced.

Iterators

The configuration starts with a sequence of C and Universal iterators, from C.program to C.Aggregates. As explained in Sec 5.1.3, they are implemented as stateless abstract domains, and handle distinct subsets of the AST nodes of the C and Universal languages. For instance, C.program handles the analysis of the program startup from its `main` function, while C.loops supports `for` and `do-while` loops. C-specific iterators often delegate

to Universal iterators, *e.g.* C.loops delegates fixpoint computation to U.loops. As another example, MOPSA currently handles function calls by semantic inlining, *i.e.* calling the iterator recursively on the function body at every call site. Inlining is implemented in the Universal iterator U.interproc. A C-specific iterator, C.interproc, translates C function calls into Universal calls, taking care of C-specific aspects, such as calls through function pointers. Further iterators support intraprocedural statements (C.intraproc and U.intraproc), calls to analyzer built-in functions (C.libraries), and the static initialization of C aggregates at program startup (C.Aggregates.

Domains

Following these iterators, the C analysis contains a combination of domains that handle atomic statements such as assignments and tests. The C.cells memory model presented in Sec. 5.2.4 decomposes program variables into a set of cells. The C.cells domain leverages the cell abstraction from [127] to handle union types and type-punning in a transparent way. Synthetic cells are then handled by a Cartesian product:

- C.machineNum translates machine integer arithmetic into mathematical arithmetic on unbounded integers, handling overflow-checking and wrap-around semantics;
- C.pointers maintains points-to information, and translates pointer arithmetic into offset arithmetic at the level of bytes.

This Cartesian product rewrites scalar expressions into mathematical expressions on unbounded integers, which are then handled natively by classic Universal numerical abstract domains. We use a reduced product between intervals (U.intervals), congruences (U.congruences) and polyhedra (U.linearRel).

5.3 Analysis of double C programs

After this presentation of the analysis of C programs with MOPSA, we are now ready to design a patch analysis for C. Our analysis of double C programs reuses indeed most of the domain-modules introduced in Fig. 5.3 of Sec. 5.1.3 and 5.2.5. In this section, we first describe two analysis front-ends we implemented to construct analyzable double C programs, either automatically or manually. Then, we will first assume double C programs, and describe our implementation of patch analysis on top of MOPSA, leveraging the semantics introduced in Sec. 3.6 for the toy NIMP_2 language, and the memory model introduced in Sec. 5.2.4 for C programs.

5.3.1 Front-ends

Our patch analysis operates on the AST of a double C program. Yet, a patch consists in a pair of C programs in practice. We must thus construct a suitable double program before running the analysis. Our implementation on top of MOPSA supports two front-ends: double programs can be constructed automatically from simple programs, like in

Chapter 4, or encoded by hand using dedicated primitives, like in Chapter 3. We present both approaches in this section.

Merging the AST of simple programs automatically

Let us start with the former approach. It consists in merging the AST of two simple programs P_1 and P_2 into the AST of a double program P.

Double parsing. We first rely on Clang to parse P_1 and P_2 independently, with possibly different pre-processing options. Note that Clang may yield different AST nodes for identical source statements. Indeed, we are interested not only in patches of program statements and expressions, but also in patches of type definitions, *e.g.* of the definitions of C structs. Some program constants depend on the offsets of struct fields in memory, *e.g.* `sizeof` and symbolic constants. Identical syntactic expressions may thus yield different constants. We thus obtain different AST nodes after resolution of identifiers and constants by Clang.

An example is shown on Fig. 5.10. Program P_1 is displayed on the first column. Programs P_2 and P are displayed on the second and third columns, respectively. See line 8: the patch adds field `x` of type `int` to `struct S`. This changes the offset of field `b`, and the size of `struct S`. Assuming P_1 and P_2 are compiled for 32-bit platforms implementing the *System V Application Binary Interfaces* (ABI) [11], `sizeof(struct S`$_1$`)=8` and `sizeof(struct S`$_2$`)=12` . Line 31 is thus simplified to `l=8;` for program P_1 and `l=12;` for program P_2.

Clang produces two AST for P_1 and P_2, which are then normalized by Mopsa. Let A_1 and A_2 denote the resulting AST for P_1 and P_2, and let A denote the AST for P. Our goal is then to merge A_1 and A_2 into A.

Gathering syntactic objects. We say that P_1 and P_2 share some syntactic object (function or variable) named x if x is found in P_1 and P_2 with the same scope – *e.g.* source file with the same name, or local variable of a shared function. Note that we do not rely on syntactic locations: a shared syntactic object may be declared at different lines P_1 and P_2. For instance on Fig. 5.10, programs P_1 and P_2 share function `main`, function `init`, variable `s` and variable `s0`. In contrast, function `f` and variable `x` are defined by program P_1 only.

We rely on A_1 and A_2 to distinguish syntactic objects defined by one program version only from syntactic objects shared by both program versions. A single instance of the former are added to A, with side information. For instance, double program P features a single instance of function `f`, and the declaration of variable `x` (line 33) is defined as a simple statement in P, for the left version only.

We also rely on A_1 and A_2 to distinguish syntactic objects shared by both program versions with identical definitions, from syntactic objects shared by program versions with different definitions. A pair of syntactic objects is added to A in the latter case. For instance, function `main` and variable `s` have different definitions in P_1 and P_2: `main`

```
1  typedef struct S0 {
2    int a;
3    int b;
4  } T0;
5
6  typedef struct S {
7    int a;
8
9    int b;
10 } T;
11
12 void init(T0 *p0, T *p) {
13   p0->a = 0;
14   p0->b = 23;
15   p->a = 0;
16   p->b = 42;
17
18 }
19
20 int f(T *p) {
21   int c;
22   c = p->a + p->b;
23   return c;
24 }
25
26 void main(void) {
27   T0 s0;
28   T s;
29   unsigned int l0, l;
30   l0 = sizeof(s0);
31   l = sizeof(s);
32   init(&s0, &s);
33   int x;
34   x = f(&s);
35 }
```

```
typedef struct S0 {
  int a;
  int b;
} T0;

typedef struct S {
  int a;
  int x;
  int b;
} T;

void init(T0 *p0, T *p) {
  p0->a = 0;
  p0->b = 23;
  p->a = 0;
  p->b = 42;
  p->x = 666;
}

void main(void) {
  T0 s0;
  T s;
  unsigned int l0, l;
  l0 = sizeof(s0);
  l = sizeof(s);
  init(&s0, &s);

}
```

```
struct S0 {
  int a;
  int b;
};

struct S1 { int a; int b1; };

struct S2 { int a; int x; int b2; };

void init1(struct S0 *p0, struct S1 *p) ||
void init2(struct S0 *p0, struct S2 *p) {
  p0->a = 0;
  p0->b = 23;
  p->a = 0;
  p->b1 = 42 || p->b2 = 42;
  skip || { p->x = 666; }
}

signed int f(struct S1 *p) {
  signed int c;
  c = (p->a + p->b1);
  return c;
}

void main1(void) || void main2(void) {
  struct S0 s0;
  { struct S1 s1; } || { struct S2 s2; }
  unsigned int l0, l;
  l0 = 8;
  l = 8 || 12;
  init1||init2(&s0, &s1 || &s2);
  { signed int x;
    x = f(&s1); } || skip
}
```

Figure 5.10: Merging two versions of a C program

features extra statements in P_1 (lines 32 and 33) while the type `struct S` of variable `s` features an extra field in P_2 (line 8). As a consequence, two versions of `main` (line 26) are declared: $\texttt{main}_1$ and $\texttt{main}_2$. Likewise, the declaration of `s` (line 8) is translated to a pair of simple statements, declaring $\texttt{s}_1$ in $\texttt{main}_1$ and $\texttt{s}_2$ in $\texttt{main}_2$. $\texttt{s}_1$ has type `struct` $\texttt{S}_1$ (P_1's definition of `struct S`), while $\texttt{s}_2$ has type `struct` $\texttt{S}_2$ (P_2's definition of `struct S`). The case of the `init` function is an additional interesting example: It enjoys different prototypes in P_1 and P_2 (line 12) as the second parameters have incompatible pointer types. The semantics of the dereference `p->b` (line 16) is also changed, as is the offset of `b` in `struct S`: *offsetof*`(struct` $\texttt{S}_1$`, b1)`=4, while *offsetof*`(struct` $\texttt{S}_2$`, b2)`=8. P_2 also features

an additional assignment (line 17). Two versions of `init` (line 12) are thus declared in A: $\texttt{init}_1$ and $\texttt{init}_2$.

In contrast, syntactic objects shared by both program versions with identical definitions are represented by a single syntactic object in A. For instance, the same variable `s0` is declared in $\texttt{main}_1$ and $\texttt{main}_2$ (line 27) as its type `struct S0` is not changed by the patch.

Populating the bodies of simple functions. The previous step has introduced version-specific functions into the AST A of P, such as `f`, $\texttt{main}_1$ and $\texttt{init}_2$, which we call (left or right) "simple functions". We then construct the body of every left (resp. right) simple function from the body of the associate function of P_1 (resp. P_2), by renaming all references to syntactic objects from A_1 (resp. A_2) to references to the corresponding objects of A. For instance, $\texttt{main}_1$ calls $\texttt{init}_1$, with `&s0` as a first argument, $\texttt{\&s}_1$ as a second argument.

Merging simple functions. Every (left or right) simple function has a prototype, and its body is a simple statement. As a third step, we merge left and right simple functions into so-called "double functions". For instance, $\texttt{init}_1$ and $\texttt{init}_2$ are merged into a double function named $\texttt{init}_1\|\texttt{init}_2$. $\texttt{init}_1\|\texttt{init}_2$ enjoys a double prototype, shown line 11 of Fig. 5.10.

In contrast, $\texttt{init}_1\|\texttt{init}_2$ has a single body, which is a double statement obtained by merging the bodies of $\texttt{init}_1$ and $\texttt{init}_2$. The merge procedure implements the **merge_stmt** algorithm described in Chapter 4. It adapts it to the C syntax (instead of the NIMP$_2$ syntax used in Chapter 4). Consistently, identical statements of $\texttt{init}_1$ and $\texttt{init}_2$ are merged into simple statements of the body of $\texttt{init}_1\|\texttt{init}_2$. This is the case of assignments to fields of a C struct of type `struct S0` (unchanged by the patch) lines 13 and 14. It is also the case of assignments to field `a` of a C struct of type `struct S` (line 15), as this field has the same type and and offset in $\texttt{struct S}_1$ and $\texttt{struct S}_2$. It is however not the case of assignments `p->b = 42;` to field `b` of a C struct of type `struct S` (line 16). Indeed, as fields $\texttt{b}_1$ and $\texttt{b}_2$ have different offsets in $\texttt{struct S}_1$ and $\texttt{struct S}_2$, assignments to fields `b` are represented by a double statement, although P_1 and P_2 feature syntactically identical assignments at line 16. This double statement is $\texttt{p->b}_1 = 42 \parallel \texttt{p->b}_2 = 42;$ *i.e.* `*((int *)(((char *) p +` *offsetof*`(struct` $\texttt{S}_1$`,` $\texttt{b}_1$`))) = 42 ||` $\texttt{*((int *)(((char *) p +}$ *offsetof*$\texttt{(struct S}_2\texttt{, b}_2\texttt{))) = 42;}$.

Our merge procedure for function bodies strives to align loops and tests, as explained in Chapter 4. In addition, it also strives to align function calls. Note that all references to simple functions are replaced by references to appropriate double functions before the merge procedure is applied. For instance, as $\texttt{main}_1$ calls $\texttt{init}_1\texttt{(\&s0,\&s}_1\texttt{)}$ and $\texttt{main}_2$ calls $\texttt{init}_2\texttt{(\&s0,\&s}_2\texttt{)}$, $\texttt{main}_1\|\texttt{main}_2$ calls $\texttt{init}_1\|\texttt{init}_2\texttt{(\&s0,\&s}_1\|\texttt{\&s}_2\texttt{)}$ in a double statement (line 32).

```
1    int x = _patch(0,2);                    int x = 0 ‖ 2;
2    if (_patch(1,0)) x++; else x--;         x++; ‖ x--;
```

Figure 5.11: Encoding double C programs by hand

Fall-back front-end

The double program constructed automatically by our **merge_stmt** algorithm is not optimal in all cases. In Chapter 4, we have described some cases where it does not allow for successful patch analyses of NIMP$_2$ using only linear invariants. Recall Example 30 of Sec. 4.1 for instance.

Our patch analysis therefore supports a fall-back front-end, allowing the user to encode double programs manually when automatic alignment is too coarse. We also use it in Sec. 5.4 as a means to evaluate the performance of our automatic merging procedure. The input language of this fall-back front-end is plain C, extended with a dedicated `_patch` built-in function. Given two simple C expressions $\texttt{e}_1$ and $\texttt{e}_2$, `_patch(`$\texttt{e}_1$`,`$\texttt{e}_2$`)` is translated into the double expression $\texttt{e}_1 \| \texttt{e}_2$. Fig. 5.11 shows an example. The left column features a C program using the `_patch` built-in, while the right column features its translation to a double program. Line 2 of this example shows how double statements can be encoded with `if` statements and `_patch` built-ins. Note that this fall-back front-end is not without limitations: it currently allows encoding patches of expressions and statements, but not patches of type definitions.

Remark 27 (Mixing automatic merge and manual hints). Our two approaches to the construction of double C programs are not exclusive. The user may instrument P_1 and P_2 with some `_patch` statements to help the automatic merge procedure in difficult cases.

5.3.2 Semantics

After this presentation of our front-ends to synthesize double C programs, let us come back to double program analysis. To this aim, we will define the semantics, the memory model, and the abstraction.

As a first step, we lift the semantics $\mathbb{S}$ of C programs introduced in Sec. 5.2.4 to double programs. To this aim, we adapt the semantics of NIMP programs introduced on Fig. 2.6 of Sec. 2.1.2 to the case of C programs. The simple program semantics for low-level C programs presented in Sec. 5.2.4 is parameterized by an ABI. P_1 and P_2 are assumed to run under the same 32-bit little-endian ABI, up to the offsets of fields in structs, which may differ. Differences in the layouts of structs are assumed to be reflected in the syntax of types, as noted in Remark 26. We thus assume a single function $\mathit{sizeof} \in \mathit{type} \to \mathbb{N}$ given, which provide the sizes of types (in bytes) for P_1 and P_2. Simple C programs P_1 and P_2 have variables in $\mathcal{V}_1$ and $\mathcal{V}_2$, respectively. As described in Sec. 5.3.1, our front-end renames variables with the same names, but different types in P_1 and P_2. We can thus assume a single function $\mathit{typeof} \in \mathcal{V} \to \mathbb{N}$ given, which provides the types of variables $V \in \mathcal{V} \triangleq \mathcal{V}_1 \cup \mathcal{V}_2$ for P_1 and P_2. We also let $\mathcal{P}tr \triangleq \mathcal{P}tr_1 \cup \mathcal{P}tr_2$

and $\mathcal{A}ddr \triangleq \mathcal{A}ddr_1 \cup \mathcal{A}ddr_2$, where the set $\mathcal{P}tr_k$ of pointer values of P_k and the subset $\mathcal{A}ddr_k$ of pointers to addressable memory bytes are defined as in Sec. 5.2.3. The (finite) universe of cells that may be dereferenced by program P_k is

$$\mathcal{C}ell_k \triangleq \{ \langle V, o, \tau \rangle \,|\, V \in \mathcal{V}_k,\, \tau \in \textit{scalar-type},\, 0 \leq o \leq \textit{sizeof}(\textit{typeof}(V)) - \textit{sizeof}(\tau) \}$$

As $\mathcal{C}ell_1 \neq \mathcal{C}ell_2$, we particularize the simple program semantics $\mathbb{S}_1$ and $\mathbb{S}_2$ of P_1 and P_2. The simple C program P_k has memory states in

$$\mathcal{E}_k = \bigcup_{C \subseteq \mathcal{C}ell_k} \{ \langle C, \rho \rangle \,|\, \rho \in C \to \mathbb{V} \}$$

A natural choice to lift the semantics of C programs to programs reading inputs and writing outputs would be to parameterize $\mathbb{S}_k$ with an infinite input stream $\iota \in \mathbb{Z}^\omega$, and let $\Sigma_k = \mathcal{E}_k \times \mathbb{N} \times \mathbb{Z}^\star$ denote the sets of programs states, as in Sec. 2.1.2. A triple $(\langle C, \rho \rangle, n, o) \in \Sigma_k$ denotes a memory state $\langle C, \rho \rangle$, an index n in the input sequence ι, and an output sequence o. Let then P be a double C program. As simple programs versions $P_1 = \pi_1(P)$ and $P_2 = \pi_2(P)$ have states in Σ_1 and Σ_2, respectively, P has states in $\mathcal{D} \triangleq \Sigma_1 \times \Sigma_2$, and semantics $\mathbb{D}[\![\, \textit{dstat} \,]\!] \in \mathbb{Z}^\omega \to \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$. $\mathbb{D}[\![\, s \,]\!]_\iota$ describes the relation between input and output states of s, which are pairs of states of simple C programs. We do not detail the definition of $\mathbb{D}[\![\, s \,]\!]$ for C, as it is the same as the definition for NIMP$_2$ shown on Figs. 3.8 and 3.11 of Chapter 3, up to the representation of simple program states, the particularization of simple program semantics, and the replacement of NIMP$_2$ variables with C left-values.

This semantics is very expressive, as it allows modeling arbitrary desynchronizations of input reads and output writes between P_1 and P_2. Such expressivity is not necessary to analyze the patches of C programs that we address in this chapter. Although it is possible to use bounded FIFO queues to abstract sequences of inputs and outputs precisely, as in Chapter 3, we decided, for this first implementation of patch analysis for C programs, to focus on double programs such that P_1 and P_2 read inputs and write outputs in lockstep, which is the vast majority of practical cases, and the only situation addressed by the related works.

We thus leverage the more abstract semantics $\tilde{\mathbb{D}}^0$ introduced in Sec. 3.5.4 for the simplified NIMP$_2^\star$ dialect of NIMP$_2$. NIMP$_2^\star$ has no simple statements $V \leftarrow \textbf{input}(a, b)$ and $\textbf{output}(V)$ for reading inputs and writing outputs. Instead, it defines double statements $V \leftarrow \textbf{input\_sync}(a, b)$ and $\textbf{assert\_sync}(V)$. Executing these statements with the simple program semantics is an error. Any desynchronization of P_1 and P_2 in the execution of these simplified statements is detected and propagated as an error, and no resynchronization is possible. Double program states are pairs (m_1, m_2), where m_1 and m_2 are memory states of P_1 and P_2. The complete semantics of simple and double programs is available in Appendix A.4. To adapt the semantics $\tilde{\mathbb{D}}^0$ of NIMP$_2^\star$ to the case of double C programs, we use the "memory-only" domain $\mathcal{D} \triangleq \mathcal{E}_1 \times \mathcal{E}_2$ for double program states, and the semantics $\mathbb{D}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$ shown on Fig. 5.12. This semantics is the same as the semantics $\tilde{\mathbb{D}}^0$ shown on Fig. A.8 of App. A.4, up to the representation of simple program states, the particularization of simple program semantics, and the

$\mathbb{D}[\![\, dstat \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

$$
\begin{array}{ll}
\mathbb{D}[\![\, \textbf{skip} \,]\!] & \triangleq \lambda X. X \\
\mathbb{D}[\![\, s_1 \parallel s_2 \,]\!] X & \triangleq \bigcup_{(\langle C_1,\rho_1\rangle,\langle C_2,\rho_2\rangle)\in X} (\mathbb{S}_1[\![\, s_1 \,]\!]\, \{\, \langle C_1,\rho_1\rangle \,\} \times \mathbb{S}_2[\![\, s_2 \,]\!]\, \{\, \langle C_2,\rho_2\rangle \,\}) \\
\mathbb{D}[\![\, l \leftarrow e \,]\!] & \triangleq \mathbb{D}[\![\, l \leftarrow e \parallel l \leftarrow e \,]\!] \\
\mathbb{D}[\![\, \textbf{assert}(c) \,]\!] & \triangleq \mathbb{D}[\![\, \textbf{assert}(c) \parallel \textbf{assert}(c) \,]\!] \\
\mathbb{D}[\![\, l \leftarrow \textbf{input\_sync}(a,b) \,]\!] & \triangleq \dot{\bigcup}_{v\in[a,b]} \mathbb{D}[\![\, l \leftarrow v \,]\!] \\
\mathbb{D}[\![\, \textbf{assert\_sync}(l) \,]\!] X & \triangleq \bigcup_{v\in\mathbb{V}} \{\, (\langle C_1,\rho_1\rangle,\langle C_2,\rho_2\rangle) \in X \mid \mathbb{E}_1[\![\, l \,]\!]\langle C_1,\rho_1\rangle = \mathbb{E}_2[\![\, l \,]\!]\langle C_2,\rho_2\rangle = \{\, v \,\} \,\} \\
\mathbb{D}[\![\, s\,;\, t \,]\!] & \triangleq \mathbb{D}[\![\, t \,]\!] \circ \mathbb{D}[\![\, s \,]\!] \\
\mathbb{D}[\![\, \textbf{if } e_1 \bowtie 0 \parallel e_2 \bowtie 0 \textbf{ then } s \textbf{ else } t \,]\!] & \triangleq \mathbb{D}[\![\, s \,]\!] \circ \mathbb{F}[\![\, e_1 \bowtie 0 \parallel e_2 \bowtie 0 \,]\!] \\
& \dot{\cup}\ \mathbb{D}[\![\, \pi_1(s) \parallel \pi_2(t) \,]\!] \circ \mathbb{F}[\![\, e_1 \bowtie 0 \parallel e_2 \not\bowtie 0 \,]\!] \\
& \dot{\cup}\ \mathbb{D}[\![\, \pi_1(t) \parallel \pi_2(s) \,]\!] \circ \mathbb{F}[\![\, e_1 \not\bowtie 0 \parallel e_2 \bowtie 0 \,]\!] \\
& \dot{\cup}\ \mathbb{D}[\![\, t \,]\!] \circ \mathbb{F}[\![\, e_1 \not\bowtie 0 \parallel e_2 \not\bowtie 0 \,]\!] \\
\mathbb{D}[\![\, \textbf{if } c \textbf{ then } s \textbf{ else } t \,]\!] & \triangleq \mathbb{D}[\![\, \textbf{if } c \parallel c \textbf{ then } s \textbf{ else } t \,]\!] \\
\mathbb{D}[\![\, \textbf{while } e_1 \bowtie 0 \parallel e_2 \bowtie 0 \textbf{ do } s \,]\!] X & \triangleq \mathbb{F}[\![\, e_1 \not\bowtie 0 \parallel e_2 \not\bowtie 0 \,]\!](\mathrm{lfp}\ H) \\
\mathbb{D}[\![\, \textbf{while } c \textbf{ do } s \,]\!] & \triangleq \mathbb{D}[\![\, \textbf{while } c \parallel c \textbf{ do } s \,]\!]
\end{array}
$$

$$
\text{where}\quad \mathbb{F}[\![\, e_1 \bowtie 0 \parallel e_2 \bowtie 0 \,]\!] X \triangleq \left\{ (\langle C_1,\rho_1\rangle,\langle C_2,\rho_2\rangle) \in X \;\middle|\; \begin{array}{l} \exists v_1 \in \mathbb{E}_1[\![\, e_1 \,]\!]\langle C_1,\rho_1\rangle : v_1 \bowtie 0 \\ \exists v_2 \in \mathbb{E}_2[\![\, e_2 \,]\!]\langle C_2,\rho_2\rangle : v_2 \bowtie 0 \end{array} \right\}
$$

$$
\begin{array}{rll}
\text{and} & H(I) & \triangleq X \\
& & \dot{\cup}\ \mathbb{D}[\![\, s \,]\!] \circ \mathbb{F}[\![\, e_1 \bowtie 0 \parallel e_2 \bowtie 0 \,]\!] I \\
& & \dot{\cup}\ \mathbb{D}[\![\, \pi_1(s) \parallel \textbf{skip} \,]\!] \circ \mathbb{F}[\![\, e_1 \bowtie 0 \parallel e_2 \not\bowtie 0 \,]\!] I \\
& & \dot{\cup}\ \mathbb{D}[\![\, \textbf{skip} \parallel \pi_2(s) \,]\!] \circ \mathbb{F}[\![\, e_1 \not\bowtie 0 \parallel e_2 \bowtie 0 \,]\!] I
\end{array}
$$

Figure 5.12: Denotational semantics of double C programs.

replacement of NIMP$^\star_2$ variables with C left-values. Note that $\mathbb{D}$ can be rewritten in the same form as $\tilde{\mathbb{D}}^0$, as

$$
\begin{array}{rcccl}
\mathbb{D}[\![\, s_1 \parallel s_2 \,]\!] & = & \mathbb{D}_2[\![\, s_2 \,]\!] \circ \mathbb{D}_1[\![\, s_1 \,]\!] & = & \mathbb{D}_1[\![\, s_1 \,]\!] \circ \mathbb{D}_2[\![\, s_2 \,]\!] \\
\mathbb{F}[\![\, e_1 \bowtie 0 \parallel e_2 \bowtie 0 \,]\!] & = & \mathbb{F}_2[\![\, e_2 \bowtie 0 \,]\!] \circ \mathbb{F}_1[\![\, e_1 \bowtie 0 \,]\!] & = & \mathbb{F}_1[\![\, e_1 \bowtie 0 \,]\!] \circ \mathbb{F}_2[\![\, e_2 \bowtie 0 \,]\!]
\end{array}
$$

$$
\begin{array}{rlll}
\text{where} & \mathbb{D}_1[\![\, s \,]\!] \triangleq \mathbb{D}[\![\, s \parallel \textbf{skip} \,]\!], & \mathbb{D}_2[\![\, s \,]\!] \triangleq \mathbb{D}[\![\, \textbf{skip} \parallel s \,]\!] \\
\text{and} & \mathbb{F}_1[\![\, c \,]\!] \triangleq \mathbb{F}[\![\, c \parallel \textbf{skip} \,]\!], & \mathbb{F}_2[\![\, c \,]\!] \triangleq \mathbb{F}[\![\, \textbf{skip} \parallel c \,]\!]
\end{array}
$$

5.3.3 Memory model

Like for $\mathbb{S}$ in Sec. 5.2.4, we aim at abstracting $\mathbb{D}$ using numerical domains, which naturally represent sets of environments with homogenous support. As in Sec. 5.2.4, double states have heterogeneous cell support, so we first unify pairs of sets of cells. To this aim, we only retain left or right versions of cells which are part of every double state in a set, soundly disregarding all others. Our new domain is thus a choice of a set of labeled cells C and a set of scalar environments on C:

$$
\mathcal{D}^\flat \triangleq \bigcup_{C \subseteq \tau_1(\mathcal{C}ell_1) \cup \tau_2(\mathcal{C}ell_2)} \{\, \langle C, R\rangle \mid R \in \mathcal{P}(C \to \mathbb{V}) \,\}
$$

$$(\mathcal{P}(\mathcal{D}), \preccurlyeq_2) \underset{\alpha_2^\flat}{\overset{\gamma_2^\flat}{\leftrightarrows}} (\mathcal{D}^\flat, \preccurlyeq_2^\flat)$$

$$\begin{array}{rcl} \alpha_2^\flat(X) & \triangleq & \langle \bar{C}_X, \bar{R}_X \rangle \\ \bar{C}_X & = & \bigcap \{\, \tau_1(C_1) \cup \tau_2(C_2) \mid \langle \langle C_1, \rho_1 \rangle, \langle C_2, \rho_2 \rangle \rangle \in X \,\} \\ \bar{R}_X & = & \{\, \langle V_k, o, t \rangle \mapsto \rho_k \langle V, o, t \rangle \mid \langle \langle C_1, \rho_1 \rangle, \langle C_2, \rho_2 \rangle \rangle \in X \,\} \end{array}$$

$$\gamma_2^\flat \langle C, R \rangle \triangleq \bigcup_{\rho \in R} \{\, \langle \langle \tau_1^{-1}(C), \rho \circ \tau_1 \rangle, \langle \tau_2^{-1}(C), \rho \circ \tau_2 \rangle \rangle \,\}$$

$$\langle C, R \rangle \preccurlyeq_2^\flat \langle C', R' \rangle \overset{\triangle}{\iff} C' \subseteq C \wedge \{\, \rho_{|C'} \mid \rho \in R \,\} \subseteq R'$$

Figure 5.13: Unified cell environments for double C programs

where we extend the notation of the syntactic renaming operator τ_1 (resp. τ_2), introduced in Sec. 3.5.5 to distinguish the variables of the left (resp. right) version of a double program with suffix 1 (resp. 2): $\tau_k \langle V, o, t \rangle \triangleq \langle \tau_k(V), o, t \rangle = \langle V_k, o, t \rangle$. A formalization is shown on Fig. 5.13, where $\preccurlyeq_2$ lifts the partial order $\preccurlyeq$ over $\mathcal{P}(\mathcal{E})$ to pairs of states, and $\preccurlyeq_2^\flat$ lifts the partial order $\preccurlyeq^\flat$ over $\mathcal{E}^\flat$ to labeled cells. Note that we write $\tau_k^{-1}(C)$ for the preimage of C under τ_k. Fig. 5.13 is very similar to Fig. 5.9, but using cells renamed with a subscript 1 (resp. 2) to denote cells coming from the memory of P_1 (resp. P_2).

Proposition 11. *The pair $(\alpha_2^\flat, \gamma_2^\flat)$ defined in Fig. 5.13 is a Galois connection.*

Adaptations of transfer functions $\mathbb{D}^\flat [\![\, dstat \,]\!] \in \mathcal{D}^\flat \to \mathcal{D}^\flat$ of atomic statements to this new domain are straightforward. For instance,

$$\mathbb{D}^\flat [\![\, s_1 \parallel s_2 \,]\!] \langle C, R \rangle \triangleq \langle \tau_1(C_1) \cup \tau_2(C_2), \{\, \langle V_k, o, t \rangle \mapsto \rho_k \langle V, o, t \rangle \mid \rho_k \in R_k \wedge k \in \{\, 1, 2 \,\} \,\} \rangle$$

where $(C_k, R_k) = \mathbb{S}_k^\flat [\![\, s_k \,]\!] \langle \tau_k^{-1}(C), \{\, \rho \circ \tau_k \mid \rho \in R \,\} \rangle$ for $k \in \{\, 1, 2 \,\}$.

Adaptations of transfer functions of compound statements are also straightforward, assuming a sound abstract join $\sqcup_2^\flat$ is provided. $\sqcup_2^\flat$ must merge environment sets defined on heterogeneous sets of labeled cells. We therefore define a unification function $unify_2 \in \mathcal{D}^\flat \times \mathcal{D}^\flat \to \mathcal{D}^\flat \times \mathcal{D}^\flat$, which lifts the unification function *unify* from Sec. 5.2.4 to labeled cells. $(\langle C_1', R_1' \rangle, \langle C_2', R_2' \rangle) = unify_2(\langle C_1, R_1 \rangle, \langle C_2, R_2 \rangle)$ adds, with *add-cell*$^\flat$, any missing labeled cells to $\langle C_1, R_1 \rangle$ and $\langle C_2, R_2 \rangle$: respectively $C_2 \setminus C_1$ and $C_1 \setminus C_2$. Thus $C_1' = C_2' = C_1 \cup C_2$. The abstract join is defined as:

$$\langle C_1, R_1 \rangle \sqcup_2^\flat \langle C_2, R_2 \rangle \triangleq \langle C_1' \cup C_2', R_1' \cup R_2' \rangle$$

5.3.4 Abstraction

We rely on numerical abstractions to abstract $\mathbb{D}^\flat$ further, into a computable abstract semantics $\mathbb{D}^\sharp$ resulting in an effective static analysis. Like [131, Sec. 5.2], our memory domain translates memory reads and writes into purely numerical operations on synthetic

cells, that are oblivious to the double semantics of double programs: each cell is viewed as an independent numeric variable, and each numeric operation is carried out on a single bi-cell store, as if emanated from a single program. This property is a key motivation for the Cell domain, and the lifting to double C programs presented in this chapter. Bi-cells may thus be fed, as variables, to a numerical abstract domain for environment abstraction.

Pointer and numerical abstractions

Like with $\mathbb{S}^\flat$ in Sec. 5.2.4, we assume that we are given, for each set of labeled cells $C \subseteq \tau_1(\mathcal{C}ell_1) \cup \tau_2(\mathcal{C}ell_2)$ an abstract domain $\mathcal{D}^\sharp_C$, with concretization γ_C. It abstracts $\mathcal{P}(C \to \mathbb{Z}) \simeq \mathcal{P}(\mathbb{Z}^{|C|})$, *i.e.*, sets of points in a $|C|-$dimensional vector space. A cell of integer type naturally corresponds to a dimension in an abstract element. We also associate a distinct dimension to each cell of pointer type; it corresponds to the offset o of a symbolic pointer $\langle V, o\rangle \in \mathcal{P}tr$. In order to abstract fully pointer values, we enrich numerical abstract environments with a map P associating to each pointer cell the set of variables it may point to. Hence, the abstract domain becomes:

$$\mathcal{D}^\sharp \triangleq \{\, \langle C, R^\sharp, P\rangle \,|\, C \subseteq \tau_1(\mathcal{C}ell_1) \cup \tau_2(\mathcal{C}ell_2),\ R^\sharp \in \mathcal{D}^\sharp_C,\ P \in P_C \to \mathcal{P}(\mathcal{V} \cup \{\, \mathbf{NULL}, \mathbf{invalid} \,\}) \,\}$$

where $P_C \subseteq C$ is the subset of cells of pointer type. Note that the map P does not need to distinguish variables of P_1 and P_2. Indeed, a given labeled cell $\langle V_1, o, t\rangle$ (resp. $\langle V_2, o, t\rangle$) can ony refer to the version of variable V in the memory of P_1 (resp. P_2).

Connecting to numerical domains

Any numerical abstract domain may be used to abstract sets of labeled cells: standard generic domains such as polyhedra, and specific domains dedicated to patch analysis, such as the $\mathbb{\Delta}^{\sharp p}$ numerical abstraction presented in Sec. 3.5.6. Our primary target is memory abstraction, in order to handle patches of statements and data structures, as well as portability properties such as endian portability, which will be presented in Chapter 7. We therefore implemented in Mopsa only a subset of the patch analysis techniques developed in Chapter 3 for now, leaving the remainder for future work. In particular, we left the implemention into Mopsa of the $\mathbb{\Delta}^{\sharp p}$ numerical domain for future work, and focused on using standard numerical domains, such as intervals, congruences, octagons and polyhedra. The result is an analysis that may not be as optimized as it could be (*e.g.*, requiring polyhedra for precision, while the more efficient $\mathbb{\Delta}^{\sharp p}$ domain would have sufficied), but nevertheless allows us to evaluate its precision. Moreover, Chapter 6 will present a refinement of the memory domain that can improve its performance without the need for the specific numerical abstractions from Sec. 3.5.6.

Coarse stream abstraction

As explained in Sec. 5.3.2, we also decided to start with patches of C programs where both versions execute input statements in lockstep. We thus did not implement, for now,

the bounded abstraction of FIFO queues presented in Sec. 3.5.2. Hence, our analysis does not allow any desynchronizations of input reads or output writes. More precisely, we implemented an analysis based on the numerical abstraction of the semantics $\tilde{\mathbb{D}}^0$ of NIMP$_2^\star$ programs over variables in $\mathcal{C}ell_1 \cup \mathcal{C}ell_2$. $\tilde{\mathbb{D}}^0$ defined in Fig. 3.24 of Sec. 3.5.4. It is parameterized by the standard semantics $\tilde{\mathbb{S}}^0$ for simple programs introduced in Sec. 3.5.4. The numerical domain provides abstractions $\tilde{\mathbb{S}}^\sharp[\![V \leftarrow e]\!]$ and $\tilde{\mathbb{C}}^\sharp[\![e \bowtie 0]\!]$ of assignments $\tilde{\mathbb{S}}^0[\![V \leftarrow e]\!]$ and tests $\tilde{\mathbb{S}}^0[\![e \bowtie 0?]\!]$ of simple programs. The abstract semantics for double programs $\tilde{\mathbb{D}}^{\sharp 0}$ ($\tilde{\mathbb{D}}^\sharp$ for short) is the same as the abstract semantics $\tilde{\mathbb{D}}^{\sharp p}$ defined in Fig. 3.26 of Sec. 3.5.5 for NIMP$_2^-$, except for the treatment of input reads. Any desynchronization of inputs is treated as a semantic error:

$$\tilde{\mathbb{D}}^\sharp_1[\![V \leftarrow \textbf{input\_sync}(a,b)]\!] \triangleq \top \triangleq \tilde{\mathbb{D}}^\sharp_2[\![V \leftarrow \textbf{input\_sync}(a,b)]\!]$$

and an error is returned and propagated. Consistently, any reachable double input reads the same value for both program versions:

$$\tilde{\mathbb{D}}^\sharp[\![V \leftarrow \textbf{input\_sync}(a,b)]\!] \triangleq \tilde{\mathbb{C}}^\sharp[\![V_1 = V_2]\!] \circ \tilde{\mathbb{S}}^\sharp[\![V_1 \leftarrow \textbf{rand}(a,b)]\!]$$

Note that $\tilde{\mathbb{S}}^\sharp$ and $\tilde{\mathbb{C}}^\sharp$ are denoted as $\tilde{\mathbb{S}}^{\sharp p}$ and $\tilde{\mathbb{C}}^{\sharp p}$ in Fig. 3.26, respectively.

5.3.5 Domains

Fig. 5.14 shows the combination of domains used to construct this abstraction. This configuration of MOPSA reuses the configuration for the analysis of C programs from Fig. 5.3, and introduces additional domains:

- double programs iterators D.program, D.intraproc, D.loops, and D.interproc;
- the memory domain lifter D.patch;
- the D.builtins domain.

Iterators

The role of double program iterators is to handle coumpound double statements, by induction on the syntax. In practice, they do it by rewriting double expressions and double statements dynamically to compositions of simple expressions and simple statements, as prescribed by the semantics $\tilde{\mathbb{D}}^\sharp$ (see Fig. 3.26).

Side, context and flow Indeed, the abstract semantics of double programs $\tilde{\mathbb{D}}^\sharp$ relies on transfer functions $\tilde{\mathbb{S}}^\sharp$ and $\tilde{\mathbb{C}}^\sharp$ for simple statements and conditions provided by the memory model and underlying numerical domain. The transfer functions of most double statements are defined as compositions of transfer functions of double statements with transfer functions of simple statements, alternating between left and right versions. The transfer functions of simple statements reuse MOPSA's domain for the analysis of C

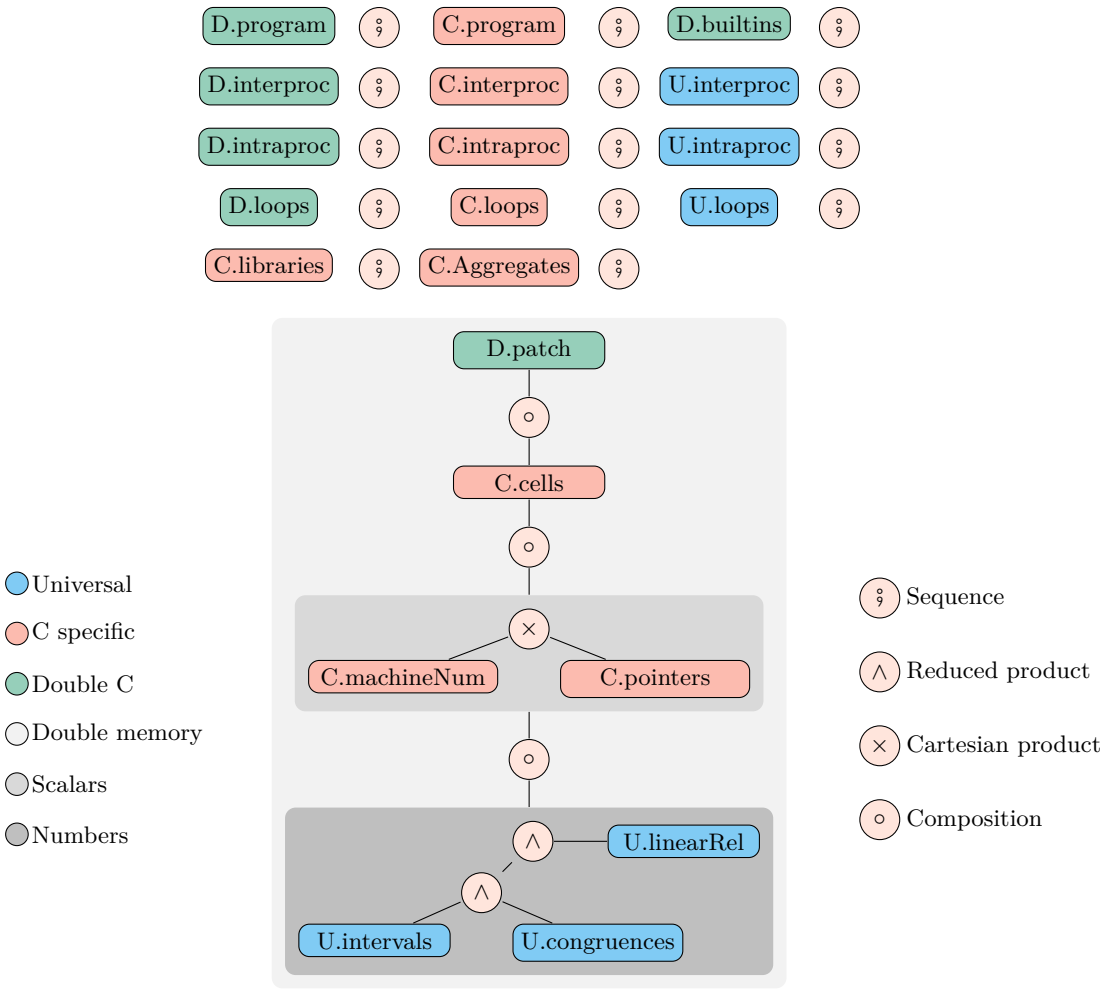


Figure 5.14: Configuration for patch analysis of C programs

programs without modification. For instance,

$$\begin{aligned}\tilde{\mathbb{D}}^\sharp[\![\,\textbf{if } c \textbf{ then } s \textbf{ else } t\,]\!] = \tilde{\mathbb{D}}^\sharp[\![\,s\,]\!] \circ \tilde{\mathbb{C}}^\sharp[\![\,\mu_2(c)\,]\!] \quad \circ \tilde{\mathbb{C}}^\sharp[\![\,\mu_1(c)\,]\!] \\ \dot{\cup}^\sharp\, \tilde{\mathbb{D}}^\sharp[\![\,t\,]\!] \circ \tilde{\mathbb{C}}^\sharp[\![\,\neg\mu_2(c)\,]\!] \circ \tilde{\mathbb{C}}^\sharp[\![\,\neg\mu_1(c)\,]\!] \\ \dot{\cup}^\sharp\, \tilde{\mathbb{S}}^\sharp[\![\,\mu_2(t)\,]\!] \circ \tilde{\mathbb{S}}^\sharp[\![\,\mu_1(s)\,]\!] \circ \tilde{\mathbb{C}}^\sharp[\![\,\neg\mu_2(c)\,]\!] \circ \tilde{\mathbb{C}}^\sharp[\![\,\mu_1(c)\,]\!] \\ \dot{\cup}^\sharp\, \tilde{\mathbb{S}}^\sharp[\![\,\mu_2(s)\,]\!] \circ \tilde{\mathbb{S}}^\sharp[\![\,\mu_1(t)\,]\!] \circ \tilde{\mathbb{C}}^\sharp[\![\,\mu_2(c)\,]\!] \quad \circ \tilde{\mathbb{C}}^\sharp[\![\,\neg\mu_1(c)\,]\!]\end{aligned}$$

where $\mu_1 \triangleq \tau_1 \circ \pi_1$ (resp. $\mu_2 \triangleq \tau_2 \circ \pi_2$) extracts the left (resp. right) version of syntactic objects with π_1 (resp. π_2), and renames the variables of the numerical domain (*i.e.* cells) with τ_1 (resp. τ_2). $\tilde{\mathbb{S}}^\sharp[\![\,\mu_1(s)\,]\!]$ formalizes the analysis of the left version of s, while $\tilde{\mathbb{S}}^\sharp[\![\,\mu_2(s)\,]\!]$ formalizes the analysis of its right version, and $\tilde{\mathbb{D}}^\sharp[\![\,s\,]\!]$ formalizes the joint analysis of both versions. MOPSA encourages distributed iterators and loose coupling between domains. Iterators know only a subset of existing AST nodes, and none of the

variables used by the underlying numerical abstraction. Therefore our implementation does not define centralized operators τ_k and π_k. Instead, it distributes over multiple domains, the support for μ_k by maintaining information on the currently analyzed program version(s) into the flow-insensitive context of the analysis introduced in Sec. 5.1.3:

```
type side = LEFT | RIGHT | BOTH
```

The side indicates on which version of a double program a double statement is executed, or a double expression is evaluated. Domains for the double program semantics may read and set it, to control on which memory, or memories, accesses should ultimately be interpreted. In particular, the side context is set by double program iterators, such as D.intraproc, before delegating implicitly to simple program iterators that are unaware of the side context, but delegate in turn to memory abstract domains such as D.patch, that use the side context to tag variables. Implicit delegation is implemented in practice by recursive `man.exec` calls to the top-level iterator, as explained in Sec. 5.1.2. We will illustrate this cooperation in Figs. 5.15 and 5.16(c) of Sec. 5.3.5. In addition, some domains of the memory abstraction of simple C program, such as C.cells and C.pointers rely on a `sizeof` function to read the sizes of variables. This function is parameterized by the ABI of the target. The side context is used to chose the right one for non-scalar types a analysis time.

Programs The D.program iterator features a single transfer function: the transfer function for whole-program execution. This transfer function first zero-initializes global variables of both program versions. Then, it constructs a double statement, where both versions call the entry point resulting from the syntactic merge of the `main` functions of both program versions, as explained in Sec. 5.3.1. Finally, it delegates to some iterator for double statements. In practice, this statement will be handled by D.intraproc.

Intraprocedural statements D.intraproc implements tranfer functions for **if** and **assume** statements, as well a patch statements $s_1 \parallel s_2$. An implementation of the transfer function of **if** e **then** s **else** s' is shown on Fig. 5.15. It handles **if** statements if the side is `BOTH`, delegating other cases to a simple statement iterator such as C.intraproc. It follows closely the theoretical abstract transfer function, as the join of four cases. There are indeed two stable cases, and two unstable cases, depending on whether program versions agree or disagree on the value of the condition. Stable cases are implemented by functions `stable_then` and `stable_else`, while unstable cases are implemented by functions `stable_then_else` and `stable_else_then`. Slightly simplified implementations of these functions are shown on Fig. 5.16. For instance, The implementation of the first unstable case $\tilde{\mathbb{S}}^\sharp[\![\, \mu_2(s') \,]\!] \circ \tilde{\mathbb{S}}^\sharp[\![\, \mu_1(s) \,]\!] \circ \tilde{\mathbb{C}}^\sharp[\![\, \neg\mu_2(e) \,]\!] \circ \tilde{\mathbb{C}}^\sharp[\![\, \mu_1(e) \,]\!]$ as `stable_then_else e s s'` is shown in Fig. 5.16(c). It illustrates how the side context is used to alternate between the left and right versions. It calls the global transfer function `man.exec` recursively on the **then** or **else** branch of either version, after setting the side context to `LEFT` or `RIGHT`. Subsequent transfer function are thus be delegated implicitly to other domains. For instance, statement **assume** e is delegated to the simple program iterator C.intraproc

```
let exec stmt man flow =
  match skind stmt with
  | S_if (e,s,s') when get_side flow = BOTH ->
      (* compute branches for stable tests *)
      let ttss   = stable_then e s  flow in
      let ffs's' = stable_else e s' flow in
      (* compute branches for unstable tests *)
      let tfss'  = unstable_then_else e s s' flow in
      let fts's  = unstable_else_then e s s' flow in
      (* Union of stable and unstable branches *)
      Flow.join_list man.lattice [ttss; ffs's'; tfss'; fts's]
```

Figure 5.15: Transfer function of **if** statements

```
let stable_then e s flow =
  man.exec (mk_assume e) flow |>
  man.exec s
```

(a) Stable **then** branch

```
let stable_else e s flow =
  man.exec (mk_assume (mk_not e)) flow |>
  man.exec s
```

(b) Stable **else** branch

```
let unstable_then_else e s s' flow =
  flow |> set_side LEFT |>
  man.exec (mk_assume e) |>
  set_side RIGHT |>
  man.exec (mk_assume (mk_not e))
  set_side LEFT |>
  man.exec s |>
  set_side RIGHT |>
  man.exec s' |>
  set_side BOTH
```

(c) First unstable branch

```
let unstable_else_then e s s' flow =
  flow |> set_side LEFT |>
  man.exec (mk_assume (mk_not e))
  set_side RIGHT |>
  man.exec (mk_assume e) |>
  set_side LEFT |>
  man.exec s' |>
  set_side RIGHT |>
  man.exec s  |>
  set_side BOTH
```

(d) Second unstable branch

Figure 5.16: Stable and unstable branches

which, in turn, delegates the evaluation of conditional expression e to the compound domain implementing the double memory abstraction. The result of evaluation depends on the current side context.

The implementation of the three other branches of the transfer function for **if** is similar. The D.loops iterator uses the same approach to implement transfer functions of **while** statements.

Note that the **if** transfer function is slightly simplified in Figs. 5.15 and 5.16(c). It omits some book-keeping details of the flow-insensitive context, as well as management of control-flow tokens and continuations stored in the `flow` to support non-local control flow instructions. Additionally, optimized implementations may use monadic mechanisms to detect when some transfer function returns $\bot$, and bypass subsequent transfer functions. This is indeed a common case in practice: there is no need to analyze unstable branches

when a condition is stable.

Interprocedural statements The D.interproc iterator rewrites double function calls to simple function calls. Double function calls occur when P_1 and P_2 call two versions of the same function, *i.e.* functions with equal names and different bodies, that are defined in source files with equal names. Note that the two versions of the function may have different lists of parameters, or return types. The bodies of the two versions of the function are assumed to be merged into a double statement, as explained in Sec. 5.3.1. The transfer function in D.interproc for a double function call generates assignments of arguments to parameters, only for parameters that are specific to a program version. It then translates the function call to a Universal function call, handled in practice by U.interproc. This generic call assigns arguments to the remaining parameters, *i.e.* parameters that are shared by both versions of the called function.

D.interproc also handles returns from double calls, as well as indirect calls through function pointers. Indirect calls are treated as double function calls or as a pair of independent function calls, depending on the resolution of the function pointer by the pointer domain. The former case is called a "stable call", as P_1 and P_2 call their respective versions of the same function, while the latter case is called an "unstable call". Unstable calls occur when P_1 and P_2 call functions that have not been merged by the front-end, *e.g.* functions with different names f_1 and f_2. In that case the pair of calls is evaluated by the default generic transfer function of double expressions $f_1(x) \parallel f_2(y)$. On the other hand, D.interproc does not handle the case of simple function calls. Simple function calls occur when P_1 and P_2 call a function f featuring identical bodies in both versions, with the same actual argument expressions. The vast majority of direct or indirect function calls in double programs are simple function calls in practice. In this frequent case, the syntactic merge of f is a simple statement, and the evaluation of the call are delegated implicitly to the corresponding transfer function of C.interproc.

Memory abstraction

The memory domain for double programs depicted in light gray in Fig. 5.14 is designed as a parameterized abstraction. The D.patch iterator lifts the C memory domain from Fig. 5.3 to double programs. It implements the memory model and the abstraction introduced in Sec. 5.3.3 and 5.3.4. To this aim, it maintains a left and right version V_1 and V_2 for every variable V of the double program, and handles the evaluation of expressions, and the assignments to left-values, by routing them to the version of the memory corresponding to the current side context. The C.cells domain accordingly synthesizes cells $\langle V_1, o, t\rangle$ and $\langle V_2, o', t'\rangle$, which account for the scalar dereferences of program versions P_1 and P_2 in the bytes of every variable V. The implementation of the C.cells domain is exactly the same as for the analysis of simple C programs: it does not use side information from the flow-insensitive context, and is unaware that cells are labeled to represent two versions of the same memory.

```
1  struct { u32 a; u32 b; } s; ‖ struct { u32 a; u32 x; u32 b; } s;
2  s.a = input_sync(0,1000); // ⟨s2, 0, u32⟩ = ⟨s1, 0, u32⟩ ∈ [0, 1000]
3  u32 x = input_sync(0,10); // ⟨x2, 0, u32⟩ = ⟨x1, 0, u32⟩ ∈ [0, 10]
4  s.b = s.a + x;            // ⟨s1, 4, u32⟩ = ⟨s1, 0, u32⟩ + ⟨x1, 0, u32⟩ ∧
5                            // ⟨s2, 8, u32⟩ = ⟨s2, 0, u32⟩ + ⟨x2, 0, u32⟩
6  assert_sync(s.b);         // ⟨s1, 4, u32⟩ ≟ ⟨s2, 8, u32⟩
```

Figure 5.17: Sum of scalar fields (Example 32)

```
1  struct { u32 a; u32 b; } s; ‖ struct { u32 a; u32 x; u32 b; } s;
2  *u32 (&s + 0) = input_sync(0,1000);
3  u32 x; *u32 (&x + 0) = input_sync(0,10);
4  *u32 (&s + 4) = *u32 (&s + 0) + *u32 (&x + 0) ‖
5  *u32 (&s + 8) = *u32 (&s + 0) + *u32 (&x + 0);
6  assert_sync(*u32 (&s + 4) ‖ *u32 (&s + 8));
```

Figure 5.18: Example 32 after pre-processing of scalar dereferences

Numerical abstraction

As mentioned in the introduction of the current section, we use the same standard numerical domains as for the analysis of C programs – intervals, congruences and polyhedra. We leave for future work the implemention into MOPSA of the $\mathbb{\Delta}^\sharp$ numerical abstraction presented in Sec. 3.5.6. This abstraction can be implemented as a Universal numerical domain. It can for instance replace U.linearRel in the global abstraction defined in Fig. 5.14. The equality abstract domain mentioned in Sec. 3.6 can be treated alike.

Built-in functions

The D.builtins domain mainly supports syntax extensions such as the `_patch` built-in function introduced in Sec. 5.3.1.

Examples

Let us demonstrate our double C program analysis on two examples: Example 32 and Example 33.

Example 32 (Preserving cell equalities through linear computations)**.** Consider the code snippet on Fig. 5.17. The C struct `s` features different definitions in P_1 and P_2 (line 1): the right version has an extra field `s.x`, which changes the size of `s` and the offset of field `s.b`. Scalars `s.a` and `x` are initialized with values read from an input stream shared between program versions P_1 and P_2 (lines 2 and 3). Their sum is assigned to `s.b` (line 4). The assertion (line 6) expresses that `s.b` holds equal values in P_1 and

```
1  struct { u32 a; u32 b; } s; ‖ struct { u32 a; u32 x; u32 b; } s;
2  s.a = input_sync(0,1000); // ⟨s2, 0, u32⟩ = ⟨s1, 0, u32⟩ ∈ [0, 1000]
3  u32 x = input_sync(0,10); // ⟨x2, 0, u32⟩ = ⟨x1, 0, u32⟩ ∈ [0, 10]
4  u32 *pa = (u32 *)&s;      // pta1 ↦ {s1} ∧ offset(pta1) = 0 ∧
5                            // pta2 ↦ {s2} ∧ offset(pta2) = 0
6  u32 *pb = pa+1;           // ptb1 ↦ {s1} ∧ offset(ptb1) = 4 ∧
7                            // ptb2 ↦ {s2} ∧ offset(ptb2) = 4
8  skip ‖ pb++;              // offset(ptb2) = 8
9  *pb = *pa + x;            // ⟨s1, 4, u32⟩ = ⟨s1, 0, u32⟩ + ⟨x1, 0, u32⟩ ∧
10                           // ⟨s2, 8, u32⟩ = ⟨s2, 0, u32⟩ + ⟨x2, 0, u32⟩
11 assert_sync(s.b);         // ⟨s1, 4, u32⟩ ≟ ⟨s2, 8, u32⟩
```

Figure 5.19: Sum of scalar fields (Example 33)

P_2 after this assignment. Fig. 5.18 shows a version of the snippet that makes the pre-processing of scalar dereferences explicit. `s.a` and `x` are pre-processed to $*_{\mathbf{u32}}\,(\&s{+}0)$ and $*_{\mathbf{u32}}\,(\&x+0)$, respectively. `s.b` is pre-processed to $*_{\mathbf{u32}}\,(\&s+4)$ in P_1 and $*_{\mathbf{u32}}\,(\&s+8)$ in P_2. The assignment of line 4 is thus merged into a double statement, following the merge procedure described in Sec. 5.3.1. The D.patch iterator lifts the C memory domain to the double program by maintaining variables $\mathtt{s}_1$, $\mathtt{s}_2$, $\mathtt{x}_1$, and $\mathtt{x}_2$ to account for P_1 and P_2. Accordingly, for $i \in \{1,2\}$, cells $\langle \mathtt{s}_i, 0, \mathbf{u32}\rangle$, $\langle \mathtt{x}_i, 0, \mathbf{u32}\rangle$ and $\langle \mathtt{s}_i, 4\times i, \mathbf{u32}\rangle$ are synthesized by the ϕ function introduced Fig. 5.8 of Sec. 5.2.4.

Fig. 5.17 shows the invariants inferred after each statement from line 2 to line 4, and the property to be checked line 6. These invariants result straightforwardly from the semantics of **input_sync** and assign statements.

Our analysis proves the assertion (line 6 of Example 32) by leveraging relational numerical domains to abstract the values of cells. The configuration shown on Fig. 5.14 uses indeed the polyhedra abstract domain, which is able to represent the necessary invariants exactly. Note that the use of such an expressive domain may hamper the scalability of the analysis. The cheaper $\mathbb{A}^\sharp$ numerical abstraction is also expressive enough for a conclusive analysis of Example 32. In addition, we will introduce in Chapter 6 an optimization of our memory model that allows for a successful analysis of Example 32 using only non-relational numerical domains, by maintaining cell equality inside the memory domain.

Example 33 (Pointer version of Example 32)**.** The alternate version of Example 32 shown on Fig. 5.19 makes the interest of the cell based memory model more obvious. Fields of struct `s` are read and written both directly and through pointers, and pointer arithmetic is used. Note that P_1 and P_2 use different pointer arithmetic expressions to access field `s.b`, as this field has different offsets in P_1 and P_2. In addition to the numeric cells synthesized for Example 32, the memory domain synthesizes pointer cells $\mathtt{pta}_i = \langle \mathtt{pa}_i, 0, \mathbf{ptr}\rangle$ and $\mathtt{ptb}_i = \langle \mathtt{pb}_i, 0, \mathbf{ptr}\rangle$, for $i \in \{1,2\}$. The memory domain

Related work origin	Benchmark	LOC	#P	Related time	Small proto. Nimp$_2$ Manual align.		Mopsa Simpl. C Manual align.		Mopsa Simpl. C Auto. align.	
[172]	Comp	13	2	539 ms	14 ms	✓	31 ms	✓	48 ms	✓
	Const	9	3	541 ms	7 ms	✓	25 ms	✓	28 ms	✓
	Fig. 2	14	1	–	4 ms	✓	19 ms	✓	31 ms	✓
	LoopMult[1]	14	2	49 s	33 ms	✓	136 ms	✓	166 ms	✓
	LoopSub	15	2	1.2 s	19 ms	✓	55 ms	✓	60 ms	✓
	UnchLoop	13	2	2.8 s	15 ms	✓	52 ms	✓	69 ms	✓
[72, 109]	loop	11	3	50 ms	12 ms	✓	32 ms	✓	43 ms	✓
	while-if	11	3	80 ms	15 ms	✓	55 ms	✓	66 ms	✓
[8, 72, 109]	digits10	24	19	1.12 s	47 ms	✓	325 ms	✓	312 ms	✓
[18, 72, 109]	barthe	13	2	120 ms	33 ms	✓	83 ms	✓	93 ms	✓
	Example 28	11	2	150 ms	40 ms	✓	71 ms	✓	81 ms	✓
[70, 20, 18]	iflow	79	0	–	–	–	808 ms	✓	808 ms	✓
[153]	sign	12	2	–	6 ms	✓	19 ms	✓	29 ms	✓
	sum	14	4	4 s	14 ms	✓	57 ms	✓	71 ms	✓
[153, 154]	copy[2]	37	1	2 s	23 ms	✓	78 ms	✓	132 ms	✓
	remove[2]	19	5	3 s	481 ms	✓[3]		✗		✗
	seq[2]	41	13	11 s	75 ms	✓	194 ms	✓	293 ms	✓
	pr[2]	111	8	1149 s	–	–	2.654 s	✓	2.686 s	✓
	test[2]	158	10	–	96 ms	✓	740 ms	✓	916 ms	✓

Figure 5.20: Polyhedral analyses of synthetic and simplified Coreutils benchmarks, with manual or automatic double program constructions.

synthesizes the same cells for the statement `*pb = *pa + x` of Example 33 and for the statement `s.b = s.a + x` of Example 32. Therefore the analysis is also successful.

5.4 Evaluation

We implemented the patch analysis for C programs presented in Sec. 5.3 on top of Mopsa. Our implementation is 3,300 lines of OCaml, 66% of which for automating double program construction. In this section, we give an experimental evaluation of this implementation.

5.4.1 From Nimp$_2$ to C

We evaluate our implementation on a set of C benchmarks from other authors [172, 153, 154], already introduced in Sec. 3.6 of Chapter 3. Unlike in Chapter 3, we analyze the C source code of all benchmarks directly – instead of encoding them into the Nimp$_2$ syntax. We add some additional benchmarks from related works on patch analysis of C programs [20, 18, 8, 72, 109]: `loop`, `while-if`, `digits10`, `barthe` and Example 28. We also include a benchmark related to information flow analysis: `iflow`, called "Joining Two Database Tables" in [70], and analyzed as "non-interference product" in [20, 18].

Fig. 5.20 summarizes the results of our analyses. For each example, we show the origin of the example (original publication), the number of lines (LOC: maximum number of lines between the two versions), the number of lines affected by the patch (#P), and the analysis time reported by the related works using their analysis – related works analyze their respective examples successfully. We reproduce the analysis times of our prototype for the NIMP$_2$ language, from Sec. 3.6 of Chapter 3. We also analyze NIMP$_2$ encodings of additional benchmarks for completeness, except for `iflow` and `pr` that need C pointers, arrays and structures. Then, we show analysis times with our analyzer for double C programs, implemented on top of MOPSA. We give two analysis results for each benchmark. For the first analysis, the double program is provided by hand, using the front-end described in Sec. 5.3.1. For the second analysis, the double program is constructed automatically, using the **merge_stmt** heuristic presented in Chapter 4 – see Sec. 5.3.1. All experiments were conducted on a Intel® Core-i7™ processor.

We only show analysis results with the polyhedra numerical abstract domain without partitioning. Indeed, we have not yet implemented into MOPSA the $\mathbb{A}^\sharp$ domain introduced in Chapter 3. Moreover, partitioning is not supported by MOPSA yet – thus the `remove` benchmark cannot be analyzed successfully. Finally, polyhedra outperform octagons on these benchmarks both in precision and efficiency, as already demonstrated in Sec. 3.6: octagons allow for successful analyzes of 30% of these benchmarks, while taking computing roughly four times longer than polyhedra. We will nonetheless discuss the result of the analysis with octagons of the synthetic C benchmarks from the related work in Sec. 6.4.2, as part of a comparison with the results obtained with a different memory model, which will be introduced in Chapter 6. We will additionally demonstrate the successful use of octagons on real-world patches from which the rest of the benchmarks originate in Sec. 5.4.2.

Fig. 5.20 shows the impact of moving from NIMP$_2$ to manually constructed double C programs on these small benchmarks: the global analysis time roughly triples. A large part of the extra cost is due to the use the Clang parser, the rest is due to powerful MOPSA features to support modularity, such as distributed iterators and dynamic expression rewriting. In constrast, the impact of automating the construction of the double program is relatively small. In particular, the impact of our **merge_stmt** algorithm, presented in Chapter 4, is negligible on these benchmarks, featuring only small C functions: from 0.04 ms for `UnchLoop` to 1.4 ms for `pr`. Note that this observation does not generalize to large C functions, as we will demonstrate in Sec. 5.4.3.

Though our patch analysis for C programs is less efficient than our analysis of NIMP$_2$ programs, it still outperforms the related works on their own examples. It is at least one order of magnitude faster than [72, 172, 153, 154], and slightly faster than [109]. Recall that [172, 153, 154] characterize differences, while we focus on inferring equivalences for now. Also note that moving from NIMP$_2$ to manually or automatically constructed double C programs does not affect the precision of our analyses on these benchmarks.

[1] only 20 loop iterations [2] Coreutils [3] with partitioning

		Simplified code					Original code				
Related origin	Bench.	Related time	MOPSA polyhedra		MOPSA octagon		LOC	#P	MOPSA polyhedra		MOPSA octagon	
[153, 154]	copy[1]	2 s	132 ms	✓	373 ms	✓	95	1	157 ms	✓	482 ms	✓
	seq[1]	11 s	293 ms	✓		✗	46	16	570 ms	✓		✗
	pr[1]	1149 s	2.686 s	✓	11.672 s	✓	114	8	1.421 s	✓	6.469s	✓
	test[1]		916 ms	✓	3.371 s	✓	352	10	9.188 s	✓		✗
	kvm[2]						248	1/11	2.707 s	✓	4.214 s	✓
	sched[2]						194	7/12	65 ms	✓		✗
	dma[2]						270	5/23	285 ms	✓	1.235 s	✓
	block[2]						324	22/6	80 ms	✓		✗
	iucv[2]						179	10/9	403 ms	✓	1.757 s	✓
	io_uring[2]						1569	10/14	868.701 s	✓		✗

Figure 5.21: Analyses of real patches from Coreutils and Linux

5.4.2 From simplified benchmarks to real code

Most of the benchmarks from the related works are synthetic C programs, written in a small subset of C featuring mostly well-typed scalar programs without pointers or pointer arithmetic. Nonetheless, the `copy`[2], `remove`[3], `seq`[4], `pr`[5] and `test`[6] benchmarks originate from real patches of the GNU core utilities. For these examples, we have started with analyzing the source codes used in the benchmarks of [153, 154]. Their tools DIZY [153] and SCORE [154] support smaller subsets of C than MOPSA does. The source codes of these examples are thus simplified to fit in the subsets of C supported by these analyzers: only scalar variables are used, function calls are inlined or replaced by uninterpreted functions, Booleans are replaced with integers, writes to memory through pointers are replaced by calls to instrumentation functions, etc.

In contrast, the large subset of C supported by MOPSA enables us to restore the original source codes, with the real data structures used in the GNU core utilities. For instance, the `set_owner` function of the `copy` benchmark features multiple calls to POSIX or helper functions; the `char_to_clump` function of the `pr` benchmark writes to a string buffer using pointer arithmetic and multiple local loops; and the `test` benchmark features bi-dimensional arrays, and multiple layers of function calls passing C structs by reference. Fig. 5.21 compares the analysis results obtained with the real code, with those

[1] Coreutils [2] Linux

[2] `https://github.com/coreutils/coreutils/commit/fc92148eac1cd2f8a5e99b3facc21e630e815bef`

[3] `https://github.com/coreutils/coreutils/commit/e73dfc3899c2a622c91c0948610a1c5c1e1972d1`

[4] `https://github.com/coreutils/coreutils/commit/32f31bad5e40f785488a325e114eaa19fd0e1fb4`

[5] `https://github.com/coreutils/coreutils/commit/6856089f7bfaca2709b303f01dae001a30930b61`

[6] `https://github.com/coreutils/coreutils/commit/7fd7709a7a5f3537f2f373dcc57e17001830591e`

obtained on the simplified C code. The `test` benchmark is analyzed less efficiently on the real code, as it features over 40 syntactic function call sites on 4 levels, which are inlined in the simplified code. In contrast, the `pr` benchmark is analyzed more efficiently on the real code, as MOPSA's memory domain allows for efficient analyses of writes to a string buffer through a pointer, as opposed to multiple calls to instrumentation functions on the simplified code. Note that we analyze the real code at least one order of magnitude faster than the related works [153, 154] analyze the simplified code. Also note that all benchmarks of Fig. 5.21 rely on the automatic construction of double programs, which has no impact on the precision on the analysis for these examples, and only negligible impact on performance.

Fig. 5.21 shows additional patches, handpicked from the Linux GitHub repository: `kvm`[7], `sched`[8], `dma`[9], `block`[10], `iucv`[11] and `io_uring`[12]. These patches mainly change the definitions of C structs, and were selected to test our memory model for double programs, which has no equivalent in the NIMP$_2$ language from previous chapters, nor in previous work. `dma` adds a field to a struct. `kvm` removes an unused field. `sched` changes the offset of a field by moving it out of an enclosed C struct to an enclosing one, and tunes the global layout with explicit alignment attributes. `block` changes the layout of a struct by moving multiple fields. `iucv` groups unions into a struct. `io_uring` flattens a struct and reduces padding, which changes the offsets of some fields. Such patches change the offsets of multiple fields in structs, and thus the effect of reads and writes of struct members, both directly and through pointers. For instance, `kvm` removes a single line in a header file (definition of an unused field), which changes the semantics of 11 statements in source files: this is the meaning of the notation "1/11" in the #P column of Fig. 5.21. Note that `kvm` features doubly-linked lists, while `block`, `iucv`, and `io_uring` feature C unions. `sched` and `io_uring` use explicit packing and alignment attributes to control the precise layout of structs or unions. `iucv` additionally features incompatible pointer casts, computed calls, and memcpys intentionally reading across neighboring fields. The equivalence (and correctness) of program versions can thus only be established by inferring precise information on the low-level representation of the memory. Thus this benchmark cannot be simplified by hand to be handled by other tools.

We analyze these original low-level programming constructs without modification. To the best of our knowledge, no related work on patch analysis is able to do so. Nonetheless, we analyze some bitwise computations of the original code imprecisely, as we rely on linear domains such as polyhedra. The successful analyses summed up on Fig. 5.21 have thus been obtained by replacing these bitwise computations with linear computations. Fig. 5.22 shows a code sample from the `io_uring` benchmark where we replaced a bitwise arithmetic expression (`sqe_flags & 2` in Fig. 5.22(a)) with a linear expression

[7] `https://github.com/torvalds/linux/commit/678a305b85d95f288c12e3d69a32d3351b34f2bb`
[8] `https://github.com/torvalds/linux/commit/ceeadb83aea28372e54857bf88ab7e17af48ab7b`
[9] `https://github.com/torvalds/linux/commit/da5a11d75d6837c9c5ef40810f66ce9d2db6ca5e`
[10] `https://github.com/torvalds/linux/commit/b60876296847e6cd7f1da4b8b7f0f31399d59aa1`
[11] `https://github.com/torvalds/linux/commit/5140aaa4604ba96685dc04b4d2dde3384bbaecef`
[12] `https://github.com/torvalds/linux/commit/9ba6a1c06279ce499fcf755d8134d679a1f3b4ed`

```
                                        IOSQE_FIXED_FILE_bit    = input(0,1);
                                        IOSQE_IO_DRAIN_bit      = input(0,1);
                                        IOSQE_IO_LINK_bit       = input(0,1);
                                        IOSQE_IO_HARDLINK_bit   = input(0,1);
                                        IOSQE_ASYNC_bit         = input(0,1);
                                        IOSQE_BUFFER_SELECT_bit = input(0,1);

sqe.flags = input(0,63);                sqe.flags = (IOSQE_FIXED_FILE_bit    * 1 +
                                                     IOSQE_IO_DRAIN_bit      * 2 +
                                                     IOSQE_IO_LINK_bit       * 4 +
                                                     IOSQE_IO_HARDLINK_bit   * 8 +
                                                     IOSQE_ASYNC_bit         * 16 +
                                                     IOSQE_BUFFER_SELECT_bit * 32);

if (unlikely(sqe_flags & 2))            if (unlikely(IOSQE_IO_DRAIN_bit * 2))
```

(a) original code (b) modified code

Figure 5.22: Stubbing bitwise computations in the `io_uring` benchmark

(`IOSQE_IO_DRAIN_bit * 2` in Fig. 5.22(b)). This limitation of our analysis will be addressed and solved in future chapters: Chapter 7 will introduce a symbolic predicate domain able to abstract some bitwise arithmetic computations, while Chapter 6 will introduce a memory domain able to represent some equalities symbolically, and infer equalities between expressions approximated imprecisely by the numerical abstraction.

Up to these slight modifications of bitwise expressions, we analyze these original Linux benchmarks precisely with the polyhedra domain. Half of them are also analyzed successfully with the octagon domain. We also analyze these benchmarks efficiently, except for the `io_uring` benchmark. This larger benchmark features indeed a large number of variables and scalar dereferences (over a thousand). The memory domain thus synthesizes accordingly many numerical variables to represent scalar cells. These numerous numerical variables are then handled by the polyhedra abstract domain, to represent linear relations (about a thousand). The polyhedra domain has exponential cost in the worst-case, which can hamper the scalability of the analysis. This limitation will be lifted in Chapter 6: the symbolic memory domain introduced in this chapter will allow for precise analyses of patches of data structures with less expressive, near-linear numerical domains.

5.4.3 Practical complexity of double program construction

Chapter 4 and Sec. 5.3.1 presented our **merge_stmt** heuristic for automating double program construction. As noted in Remark 24, the cost of this heuristic is expected to be polynomial in the size of functions, and linear in the number of functions. The practical complexity of **merge_stmt** should be evaluated on patches of C functions featuring a large number of statements. As noted in Sec. 5.4.1, the functions of previous

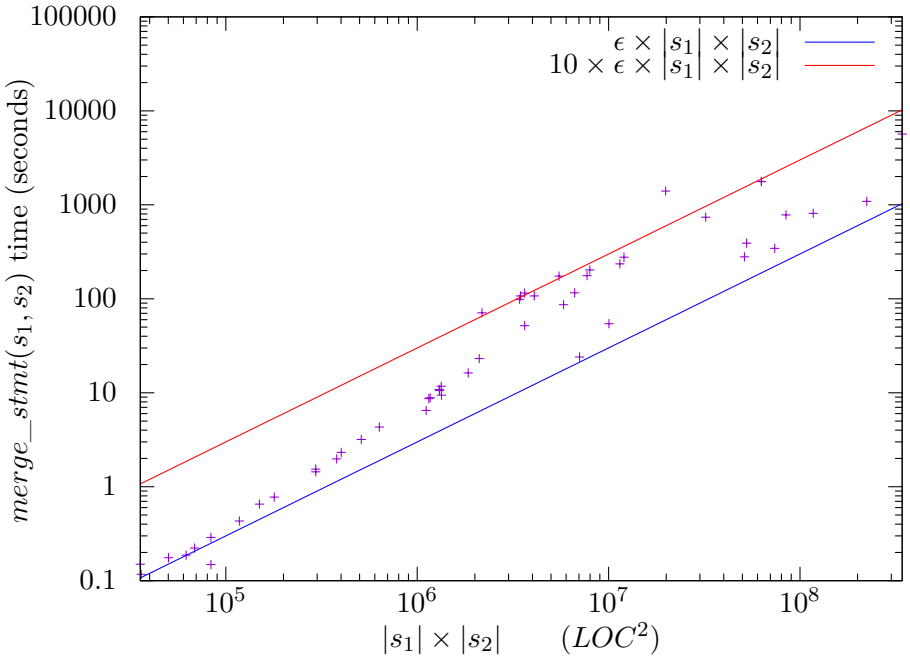


Figure 5.23: Quadratic complexity of **merge_stmt** on large C functions

benchmarks are too small for an experimental evaluation. In our experience, such large functions are not commonplace in open source software. Yet, they occur in industrial automatically generated code. We use a large avionics software benchmark which will be presented in detail in Chapter 7, as it is concerned with endian portability analysis rather than patch analysis. This benchmark features C functions of several thousand lines, each of which feature several hundred patches.

Fig. 5.23 shows the run-times of **merge_stmt** on these large functions, in log-log plot. Each dot denotes a patched function. The abscissa measures the product of lengths $|s_1| \times |s_2|$, where s_1 and s_2 denote the original and patched versions of its body, respectively. The ordinate measures the run-time t_{12} of **merge_stmt**(s_1, s_2). 94% of the measurements are between the blue and red lines, hence satisfy the relation $\epsilon \times |s_1| \times |s_2| \leq t_{12} \leq 10 \times \epsilon \times |s_1| \times |s_2|$, where $\epsilon = 3.6 \times 10^{-6} s/LOC^2$. These experiments are thus consistent with the quadratic asymptotic complexity stated in Chapter 4: $t_{12} = \mathcal{O}(|s_1| \times |s_2|)$.

5.5 Related works

We have already discussed related works on patch analysis in Chapter 3, and on product program construction in Chapter 4. In this section, we mainly focus on the support of the C language by related works on patch analysis.

[153, 154] address numerical programs in the C syntax, but their memory model does not allow tracking pointer equivalences, let alone analyzing patches of low-level C software abusing the weak type system of C.

[172] develop an implementation on top of the CPROVER framework, which supports a large subset of C. Programs are analyzed at the level of GOTO-LANGUAGE, the intermediate language used in the CProver framework. Yet their experimental evaluation focuses on small well-typed integer programs.

[72] use Horn constraint solving to infer coupling relations and relational procedure summaries, which works well for similarly structured integer programs. They rely on user-provided synchronization marks to help the alignment of program versions with dissimilar control structures, and user-provided relational invariants to enable proofs of equivalence. [109] adds support for pointer programs, as well as inference of invariants, in a CEGAR-based approach. Yet, this approach cannot handle union types and heterogeneous pointer casts, which our memory model supports.

[39] support some low-level C constructs, such as incompatible pointer casts. Yet the supported subset of C and the memory model are not described formally. In addition, they compare pairs of procedures, assuming freedom from aliases. Likewise, the SYMDIFF tool [111] operates at the level of the Boogie [17] intermediate language. The HAVOC [45] front-end is used to translate input C programs. HAVOC assumes that the input C programs are "field safe", *i.e.* different field names cannot alias, and maintains a map per word-valued (scalar or pointer) field and type. In contrast, we rely on the C memory model of MOPSA, based on the cell abstract domain, which allows for sound analyses despite the presence of aliases.

Like us, [59] rely on a C memory model based on [26], which gives a well-defined semantics to low-level programming constructs. Unlike us, they do not formalize this semantics and the related abstraction.

5.6 Conclusion

In this chapter, we presented an implementation of our patch analysis for C programs, on top of the MOPSA platform. This implementation benefits a lot from MOPSA's features to support modular development: the domains for the analysis of C programs are reused in a straightforward way. This includes the cell-based memory model [127], which enables precise analyses of architecture-dependent low-level C programs. We therefore needed to implement only the syntactic construction of double programs, a domain lifting the cell-based memory model to double programs, and iterators for the double program semantics. These iterators are quite lightweight (only 1,100 lines of OCaml), as they delegate to simple program iterators.

Our implementation competes with the state-of-the-art, both in efficiency and precision. It also extends the scope of the analysis to patches of data structures, which has no equivalent in the related works. Such patches may be viewed as a way to address a first kind of portability property: robustness to variations of the offsets of scalar fields,

such as those introduced by changes in ABI, compiler options or language extensions such as attributes of types or variables.

The scalability of the approach, in contrast, is hampered by the use of expressive relational numerical domains: the `io_uring` benchmark, for instance, is not analyzed efficiently. This can be improved by using less expressive domains, such as an equality domain and the $\triangle^\sharp$ domain introduced in Chapter 3. A complementary optimization of the memory model will be presented in Chapter 6. Finally, we will present an additional extension of the memory model in Chapter 7, which allows inferring an additional portability property in a scalable way.

Chapter 6

Sharing cells in the memory abstraction

Chapter 5 presented an analysis of double C programs on top of the Mopsa platform. This analysis reuses generic abstract domains, as well as specific domains developed for the analysis of simple C programs. In particular, it takes advantage of the C.cells memory abstraction, which implements the cell based memory model for C originally introduced in [127]. This memory model is key to allow sound and precise analyses of programs that abuse unions and pointers to bypass the type system of C, a commonplace practice in low-level programming known as *type punning*.

Most experiments of Sec. 5.4 were conducted with the polyhedra numerical domain, an approach that does not scale to large programs. In this chapter, we present an optimization of the memory model designed to reduce the burden of the numerical abstraction. We extend the memory model so that it can represent most relevant equalities symbolically, thus reducing the number of dimensions in the numerical abstraction. This new memory model will additionally allow successful analyses of some patches using only non-relational numerical domains.

This chapter is organized as follows. We first motivate the memory model optimization with idiomatic examples in Sec. 6.1. Then, we describe our extension of the memory abstraction in Sec. 6.2, and its implementation on top of Mopsa in Sec. 6.3. Finally, Sec. 6.4 provides an experimental evaluation of our implementation, and Sec. 6.5 concludes.

6.1 Motivating examples

In this section, we show the need for optimizing the cell based memory abstraction, and suggest directions for optimizations. To this aim, we give two motivating examples of double C programs: Example 32 and Example 34. Example 32 is analyzed successfully with the patch analysis presented in Chapter 5, at the cost of an expressive numerical abstraction such as polyhedra or affine equalities [102]. Example 34 cannot be analyzed successfully, even with polyhedra.

```
1  struct { u32 a; u32 b; } s; ‖ struct { u32 a; u32 x; u32 b; } s;
2  s.a = input_sync(0,1000); // ⟨s2, 0, u32⟩ = ⟨s1, 0, u32⟩ ∈ [0, 1000]
3  u32 x = input_sync(0,10); // ⟨x2, 0, u32⟩ = ⟨x1, 0, u32⟩ ∈ [0, 10]
4  s.b = s.a + x;            // ⟨s1, 4, u32⟩ = ⟨s1, 0, u32⟩ + ⟨x1, 0, u32⟩ ∧
5                            // ⟨s2, 8, u32⟩ = ⟨s2, 0, u32⟩ + ⟨x2, 0, u32⟩
6  assert_sync(s.b);         // ⟨s1, 4, u32⟩ ≟ ⟨s2, 8, u32⟩
```

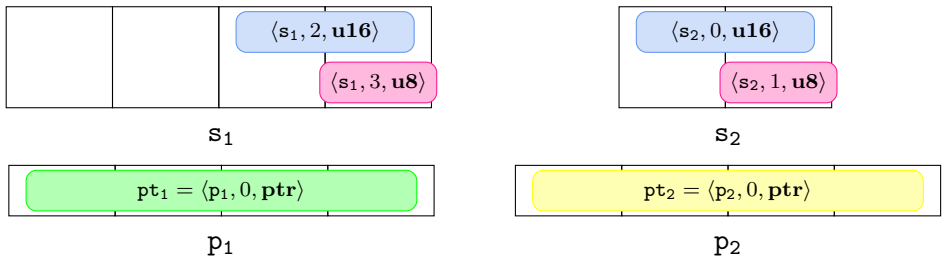
Figure 6.1: Sum of scalar fields (Example 32).

```
1  struct { u16 a; u16 b; } s; ‖ struct { u16 b; } s;
2  s.b = input_sync(0,1000); // ⟨s2, 0, u16⟩ = ⟨s1, 2, u16⟩ ∈ [0, 1000]
3  u8 *p = (u8 *) &s + 1;    // pt1 ↦ {s1} ∧ offset(pt1) = 1 ∧
4                            // pt2 ↦ {s2} ∧ offset(pt2) = 1
5  p+=sizeof(s.a) ‖ skip;    // offset(pt1) = 3
6  assert_sync(*p);          // ⟨s1, 3, u8⟩ = ⌊⟨s1, 2, u16⟩/2^8⌋ mod 2^8 ∧
7                            // ⟨s2, 1, u8⟩ = ⌊⟨s2, 0, u16⟩/2^8⌋ mod 2^8 ∧
8                            // ⟨s1, 3, u8⟩ ≟ ⟨s2, 1, u8⟩
```

Figure 6.2: Byte extraction, with numerical invariants over cells (Example 34).

To start with, let us come back to Example 32 from Sec. 5.3.5. The source code and invariants are reproduced on Fig. 6.1 for convenience. The goal of the patch analysis for C programs presented in Chapter 5 is to prove the assertion (line 6) by leveraging numerical domains to abstract the values of cells. However, such invariants require an expressive relational domain, such as polyhedra or linear equalities, which may hamper the scalability of the analysis. In particular, we note that we need to infer many equalities between the left and right versions of the same scalar fields. This is no surprise as we expect most variables to hold equal values in the left and right memories most of the time, with only local differences. Rather than relying completely on the expressiveness of the underlying numerical domain, we will optimize our memory model for this common case. In this chapter, we will present our approach to this optimization, which consists in sharing the representations of pairs of cells associated to pairs of "matching" scalar fields from P_1 and P_2. For instance, our optimized memory model will synthesize a single so-called *shared bi-cell* to hold the value of scalar field `s.a`, which is the same in both program version, instead the pair of equal cells $\langle \mathtt{s}_1, 0, \mathbf{u32}\rangle$ and $\langle \mathtt{s}_2, 0, \mathbf{u32}\rangle$. Cell sharing will act as a symbolic representation of cells equality. This optimization will allow a successful analysis of Example 32 using only non-relational numerical domains with linear cost.

Example 34 (Preserving cell equalities through modular computations from the memory abstraction)**.** While Example 32 featured well-typed, portable C code, Fig. 6.2 shows a related example relying on type-puning to read the second byte of field `s.b` (line 6). Note that this non portable code requires different pointer arithmetic expressions for P_1



(a) Program 1 (b) Program 2

Figure 6.3: Memory cells of Example 34: □ ∈ [0, 1000], □ = ⌊□/2^8⌋ mod 2^8.

and P_2 to reference the "same" byte of field `s.b` (line 5), due to the additional structure field in P_1 and the associated shift in field offets. Fig. 6.3 shows the cells synthesized at the end of the program. The invariants inferred by the analysis presented in Chapter 5 are displayed in blue on Fig. 6.2. The property to be proved is displayed in red at line 8. It holds indeed if P_1 and P_2 run on platforms with the same byte-order – a 32-bit little-endian System V ABI is assumed here. 1-byte cells $\langle \mathtt{s}_1, 3, \mathbf{u8} \rangle$ and $\langle \mathtt{s}_2, 1, \mathbf{u8} \rangle$ are synthesized by the function ϕ of the memory model to account for the dereference `*p` line 6. Function ϕ defines the values of these 1-byte cells as modular arithmetic functions of the values of 2-byte cells $\langle \mathtt{s}_1, 2, \mathbf{u16} \rangle$ and $\langle \mathtt{s}_2, 0, \mathbf{u16} \rangle$, respectively. Such non-linear relations are not represented precisely by linear abstract domains such as polyhedra. As consequence, this example is not analyzed successfully by the analysis presented in Chapter 5. Several options exist to improve the precision. For instance, a more expressive numerical abstract domain can be used, e.g. by combining polyhedra with a symbolic numerical abstraction. Yet this approach would not improve the scalability of the analysis. In this chapter, we present another solution: a symbolic abstraction of the memory model that allows a successful analysis of this example using only a non-relational numerical abstract domain with linear cost.

6.2 Memory model optimization

In this section, we optimize the cell-based memory model introduced in Sec. 5.2.4 for double program analysis. In Sec. 6.1 we noted that we expect most scalar fields to hold equal values in P_1 and P_2. We will thus let our new memory model represent all such equalities symbolically and implicitly, by merging the representations of equal cells in the representation of states.

6.2.1 Labeling cells with sides

We start with the domain of double cell-based memory states $\mathcal{D} \triangleq \mathcal{E}_1 \times \mathcal{E}_2$ introduced in Sec. 5.3.2, where $\mathcal{E}_k = \bigcup_{C \subseteq \mathcal{C}ell_k} \{ \langle C, \rho \rangle \mid \rho \in C \to \mathbb{V} \}$ is the set of states of simple

$$(\mathcal{P}(\mathcal{D}), \preccurlyeq_2) \underset{\tilde{\alpha}}{\overset{\tilde{\gamma}}{\leftrightarrows}} (\mathcal{P}(\tilde{\mathcal{D}}), \preccurlyeq)$$

$$\tilde{\alpha}(X) \triangleq \bigcup_{\langle\langle C_1,\rho_1\rangle,\langle C_2,\rho_2\rangle\rangle \in X} \{\, \langle \iota_1(C_1) \cup \iota_2(C_2), \langle c,k\rangle \mapsto \rho_k(c)\rangle \,\}$$
$$\tilde{\gamma}(Y) \triangleq \bigcup_{\langle C,\rho\rangle \in Y} \{\, \langle\langle \iota_1^{-1}(C), \rho \circ \iota_1\rangle, \langle \iota_2^{-1}(C), \rho \circ \iota_2\rangle\rangle \,\}$$

Figure 6.4: Environments on single cells

C program P_k. Then, we let the cell environment of every double state operate on the disjoint union of the cells of P_1 and the cells of P_2, by labeling every cell with its program version.

We introduce the set of *single cells*

$$\widetilde{\mathcal{C}ell} \triangleq \mathcal{C}ell_1 \uplus \mathcal{C}ell_2 = (\mathcal{C}ell_1 \times \{\, 1 \,\}) \cup (\mathcal{C}ell_2 \times \{\, 2 \,\})$$

to account for both versions. $\langle c, 1\rangle \in \widetilde{\mathcal{C}ell}$ denotes a cell c in the memory of P_1, while $\langle c, 2\rangle \in \widetilde{\mathcal{C}ell}$ denotes a cell c in the memory of P_2. Abstract double memory states are thus elements of the domain

$$\tilde{\mathcal{D}} \triangleq \bigcup_{C \subseteq \widetilde{\mathcal{C}ell}} \{\, \langle C, \rho\rangle \mid \rho \in C \to \mathbb{V} \,\}$$

The lattice of properties $(\mathcal{P}(\tilde{\mathcal{D}}), \preccurlyeq, \cup)$ is equipped with partial order

$$\tilde{X} \preccurlyeq \tilde{X}' \iff \forall (C,\rho) \in \tilde{X} : \exists (C',\rho') \in \tilde{X}' : C' \subseteq C \wedge \rho' = \rho_{|C'}$$

where $\rho_{|C'}$ denotes the restriction of ρ to C'. Note that we reuse the partial order $\preccurlyeq$ defined in Sec. 5.2.4 for simple C programs. Indeed the definition is identical, albeit on a different set. This abstraction does not lose information (Galois isomorphism). A formalisation is shown on Fig 6.4, where $\iota_k \in \mathcal{C}ell \to \widetilde{\mathcal{C}ell}$ labels cells with side $k \in \{\, 1, 2 \,\}$: $\iota_k(c) \triangleq \langle c, k\rangle$. Note that we write $\iota_k^{-1}(C)$ for the preimage of C under ι_k.

Proposition 12 (Isomorphic cell labeling)**.** *The pair $(\tilde{\alpha}, \tilde{\gamma})$ defined in Fig. 6.4 is a Galois isomorphism.*

Changes to transfer functions using this representation are straightforward. For instance, we demonstrate $\tilde{\mathbb{D}}[\![\, dstat \,]\!] \in \mathcal{P}(\tilde{\mathcal{D}}) \to \mathcal{P}(\tilde{\mathcal{D}})$ in the case of two syntactically different statements:

$$\tilde{\mathbb{D}}[\![\, s_1 \parallel s_2 \,]\!]\,(\tilde{X}) \triangleq \bigcup_{\langle C,\rho\rangle \in \tilde{X}} \left\{ \langle C', \rho'\rangle \;\middle|\; \begin{array}{l} C' = \iota_1(C_1) \cup \iota_2(C_2) \\ \rho' : \langle c,k\rangle \mapsto \rho_k(c) \\ (C_k, \rho_k) \in \mathbb{S}_k[\![\, s_k \,]\!]\ \{\, \langle \iota_k^{-1}(C), \rho \circ \iota_k\rangle \,\} \end{array} \right\}$$

6.2.2 Merging single cells

As a second step, we introduce additional cells in the representation of states, in order to represent equalities between cells of P_1 and cells of P_2.

$$(\mathcal{P}(\tilde{\mathcal{D}}), \preccurlyeq) \underset{\hat{\alpha}}{\overset{\hat{\gamma}}{\leftrightarrows}} (\mathcal{P}(\hat{\mathcal{D}}), \preccurlyeq)$$

$$\hat{\alpha}(X) \triangleq \bigcup_{\langle C,\rho\rangle \in X} \{\, \langle C \cup \left\{ \langle c_1, c_2\rangle \in C^2 \;\middle|\; \begin{array}{l} \mathbf{Bic}\langle c_1, c_2\rangle \wedge \\ \rho(c_1) = \rho(c_2) \end{array} \right\}, c \mapsto \begin{cases} \rho(c) & \textbf{if } c \in \widetilde{\mathcal{C}ell} \\ \rho(c_1) & \textbf{if } c = \langle c_1, c_2\rangle \in \widetilde{\mathcal{C}ell}^2 \end{cases} \rangle \,\}$$

$$\hat{\gamma}(Y) \triangleq \bigcup_{\substack{\langle C,\,\rho\rangle \,\in\, Y \\ \wedge \mathbf{Con}\langle C,\,\rho\rangle}} \{\, \langle (C \cap \widetilde{\mathcal{C}ell}) \cup \textstyle\bigcup_{\langle c_1,c_2\rangle \in C} \{\, c_1, c_2 \,\}, c \mapsto \begin{cases} \rho\langle c, c'\rangle & \textbf{if } \exists c' : \langle c, c'\rangle \in C \\ \rho\langle c', c\rangle & \textbf{else if } \exists c' : \langle c', c\rangle \in C \\ \rho(c) & \textbf{otherwise} \end{cases} \rangle \,\}$$

where

$$\mathbf{Bic}\langle c_1, c_2\rangle \overset{\triangle}{\iff} \exists \tau : \forall i \in \{\, 1, 2 \,\} : \exists V_i, o_i : c_i = \langle V_i, o_i, \tau, i\rangle$$

$$\mathbf{Con}\langle C, \rho\rangle \overset{\triangle}{\iff} \forall \langle c_1, c_2\rangle \in C : \mathbf{Bic}\langle c_1, c_2\rangle \wedge \forall c \in C \cap \{\, c_1, c_2 \,\} : \rho(c) = \rho\langle c_1, c_2\rangle$$

Figure 6.5: Representing equalities symbolically with shared bi-cells

We denote as $\mathcal{B}icell \triangleq \widetilde{\mathcal{C}ell} \cup (\widetilde{\mathcal{C}ell} \times \widetilde{\mathcal{C}ell})$ the set of so-called bi-cells. A bi-cell is either a single cell in $\widetilde{\mathcal{C}ell}$, or a pair of such cells in $\widetilde{\mathcal{C}ell} \times \widetilde{\mathcal{C}ell}$ assumed to hold equal value, called a *shared bi-cell*. Bi-cell sharing allows a single representation, in the memory environment, for two cells from different program versions and holding equal values. Abstract memory states are thus elements of

$$\hat{\mathcal{D}} \triangleq \bigcup_{C \subseteq \mathcal{B}icell} \{\, \langle C, \rho\rangle \,|\, \rho \in\, C \to \mathbb{V} \,\}$$

A formalization is shown on Fig 6.5. Note that we overload the partial order $\preccurlyeq$ again. $\hat{\alpha}$ uses shared bi-cells to represent pairs of single cells from different program versions and holding equal values, only if these single cells share the same type. This restriction is modeled by the predicate **Bic**. The reason for this restriction is that our bi-cell synthesis relies on the byte-representations of the memories of P_1 and P_2 to pattern-match the abstract memory state. In addition, $\hat{\gamma}$ uses the predicate **Cons** to filter away any inconsistent abstract state from the concretization. An abstract state is consistent if all its shared bi-cells hold the same values as their left or right projections, if any.

Proposition 13 (Connection between single cells and bi-cells of the same type)**.** *The pair $(\hat{\alpha}, \hat{\gamma})$ defined in Fig. 6.5 is a Galois connection.*

6.2.3 Bi-cell synthesis

A cornerstone of our optimization of the memory model is bi-cell synthesis. In order to read or write a scalar value to a given location of memory, we must create a suitable bi-cell, or retrieve an existing one from the environment. To guarantee the soundness of the analysis when adding a new bi-cell, it is necessary to ensure that values assigned to it are consistent with those of existing overlapping bi-cells. Our memory domain first attempts to synthesize shared bi-cells if an equality can be inferred from the environment,

```
1  struct { u16 a; u16 b; } s; || struct { u16 b; } s;
2  s.b = input_sync(0,1000); // ⟨c1, c2⟩ ∈ [0, 1000]
3  u8 *p = (u8 *) &s + 1;    // p1 ↦ {s1} ∧ offset(p1) = 1 ∧
4                            // p2 ↦ {s2} ∧ offset(p2) = 1
5  p+=sizeof(s.a) || skip;   // offset(p1) = 3
6  assert_sync(*p);          // ⟨c'1, c'2⟩ = ⌊⟨c1, c2⟩/2^8⌋ mod 2^8
7                            // c'1 ?= c'2
```

Figure 6.6: Byte extraction, with numerical invariants over bi-cells.

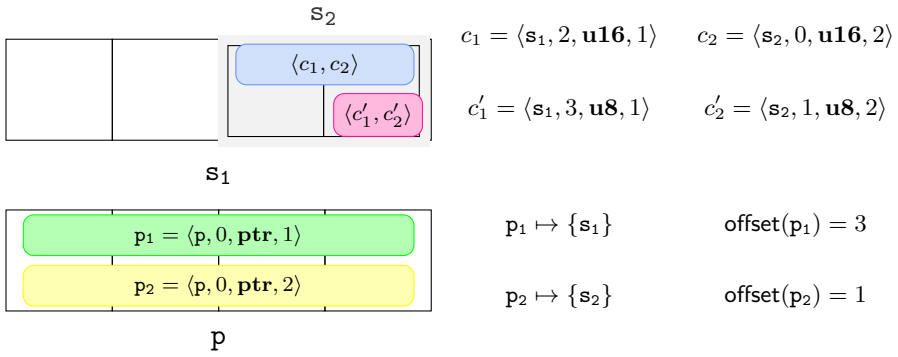


Figure 6.7: Bi-cells of Example 34: □ ∈ [0, 1000], ■ = ⌊□/2^8⌋ mod 2^8.

by pattern-matching. In case of failure, it safely defaults to a pair of single cells, the values of which are set according to those of existing overlapping bi-cells.

Let us illustrate bi-cell synthesis in the case of Example 34. Fig. 6.7 shows the bi-cells obtained after analyzing the program, and Fig. 6.6 shows the numerical invariants over bi-cells. Since the **input_sync** statement line 2 assigns equal values to $c_1 \triangleq \langle \mathtt{s}_1, 2, \mathbf{u16}, 1\rangle$ and $c_2 \triangleq \langle \mathtt{s}_2, 0, \mathbf{u16}, 2\rangle$, our memory domain synthesizes a shared bi-cell $\langle c_1, c_2\rangle$. In contrast, two single pointer bi-cells $\mathtt{p}_1 = \langle \mathtt{p}, 0, \mathbf{ptr}, 1\rangle$ and $\mathtt{p}_2 = \langle \mathtt{p}, 0, \mathbf{ptr}, 2\rangle$ are synthesized and updated separately lines 3 to 5, as pointer `p` has different bases and offsets in P_1 and P_2. Nonetheless, our memory domain attempts to synthesizes a shared 1-byte bi-cell $\langle c'_1, c'_2\rangle$ for the dereference `*p` line 6, where $c'_1 = \langle \mathtt{s}_1, 3, \mathbf{u8}, 1\rangle$ and $c'_2 = \langle \mathtt{s}_2, 1, \mathbf{u8}, 2\rangle$. To this aim, it attempts to match reachable abstract states with a set of predefined patterns. One of these patterns is as follows: it searches the set of previously synthesized bi-cells for any shared integer bi-cell x such that $\langle c'_1, c'_2\rangle$ extracts 1 byte of x. The match $x = \langle c_1, c_2\rangle$ is found. Therefore $\langle c'_1, c'_2\rangle$ is synthesized, as a proof of $c'_1 = c'_2$ (recall P_1 and P_2 are assumed to have the same endianness). The success of the synthesis relies solely on pattern-matching in this case: the assertion is proved even if the underlying numerical domain is unable represent the relation $\langle c'_1, c'_2\rangle = \lfloor \langle c_1, c_2\rangle / 2^8 \rfloor \bmod 2^8$ precisely. Our implementation proves it with the interval domain. The pattern we used in this case relies only on the set of synthesized bi-cells. Note that some other

$$
\begin{array}{l}
\mathit{equal}(\langle V, o, \tau, k\rangle, \langle V', o', \tau, k'\rangle)\langle C, \rho\rangle \triangleq \\
\quad \textbf{let } c = \langle V, o, \tau, k\rangle \textbf{ and } c' = \langle V', o', \tau, k'\rangle \textbf{ and } s = \mathit{sizeof}(\tau) \textbf{ in} \\
\quad \langle c, c'\rangle \in C \vee \langle c', c\rangle \in C \ \vee \\
\quad (\exists (x, x') \in \mathit{occ}(c, C) \times \mathit{occ}(c', C) : \ \rho(x) = \rho(x')) \ \vee \\
\quad (\forall 0 \leq w < s : \ \mathit{equal}(\langle V, o + w, \mathbf{u8}, k\rangle, \langle V', o' + w, \mathbf{u8}, k'\rangle)\langle C, \rho\rangle) \ \vee \\
\quad \left(\exists c_\star, c'_\star \in \mathit{flatten}(C) \setminus \{\, c, c' \,\} : \begin{cases} c_\star = \langle V, o_\star, \tau_\star, k\rangle \ \wedge \\ c'_\star = \langle V', o'_\star, \tau_\star, k'\rangle \ \wedge \\ o - o_\star = o' - o'_\star \ \wedge \\ o + s \leq o_\star + \mathit{sizeof}(\tau_\star) \ \wedge \\ o' + s \leq o'_\star + \mathit{sizeof}(\tau_\star) \ \wedge \\ \mathit{equal}(c_\star, c'_\star)\langle C, \rho\rangle \end{cases} \right)
\end{array}
$$

Figure 6.8: Equality test between single cells.

patterns involve the bi-cell environments, *e.g.* tests for bi-cell equalities.

Shared bi-cell synthesis

More generally, given an abstract memory state $\langle C, \rho\rangle \in \hat{\mathcal{D}}$ and a pair of scalar dereferences of the same type $c_1 \in \mathcal{C}ell_1$ from P_1 and $c_2 \in \mathcal{C}ell_2$ from P_2, the function $\hat{\phi} \in \mathcal{C}ell_1 \times \mathcal{C}ell_2 \to \hat{\mathcal{D}} \to \mathcal{B}icell \cup \{\, \top \,\}$ formalizes the patterns matched when attempting to synthesize a shared bi-cell for c_1 and c_2:

$$
\hat{\phi}(c_1, c_2)\langle C, \rho\rangle \triangleq \begin{cases} \langle\langle c_1, 1\rangle, \langle c_2, 2\rangle\rangle & \textbf{if } \ \mathit{equal}(\langle c_1, 1\rangle, \langle c_2, 2\rangle)\langle C, \rho\rangle \\ \top & \textbf{otherwise} \end{cases}
$$

$\hat{\phi}$ returns a shared bi-cell $\langle\langle c_1, 1\rangle, \langle c_2, 2\rangle\rangle$ if $\langle c_1, 1\rangle = \langle c_2, 2\rangle$ may be inferred from the environment. Otherwise, it returns a failure $\top$.

$\hat{\phi}$ relies on the predicate *equal* to compare two single cells of the same type. An implementation is shown on Fig. 6.8. *equal* returns *true* when compared single cells are part of a shared bi-cell, or when equality is ensured by the environment. In the formula, we denote the occurrences of a single cell in the environment as

$$
\mathit{occ}(c, C) \triangleq \{\, c' \in C \,|\, c' = c \vee \exists c'' : \ c' = \langle c, c''\rangle \vee c' = \langle c'', c\rangle \,\}
$$

Otherwise, *equal* compares individual 1-byte bi-cells of the same weights, *i.e.* at equal offsets in the compared single cells, as we assume for now the same endianness encoding in P_1 and P_2. Otherwise, *equal* searches for a pair of larger single cells in the environment, which are equal and contain the compared cells at equal offsets. Recall that we used this last pattern for Example 34, in order to synthesize $\langle\langle \mathtt{s}_1, 3, \mathbf{u8}, 1\rangle, \langle \mathtt{s}_2, 1, \mathbf{u8}, 2\rangle\rangle$ from $\langle\langle \mathtt{s}_1, 2, \mathbf{u16}, 1\rangle, \langle \mathtt{s}_2, 0, \mathbf{u16}, 2\rangle\rangle$. In the formula, we denote the set of single cells in the environment as

$$
\mathit{flatten}(C) \triangleq \{\, c \in \widetilde{\mathcal{C}ell} \,|\, c \in C \vee \exists c' \in C : \ \langle c, c'\rangle \in C \vee \langle c', c\rangle \in C \,\}
$$

$\widehat{add\text{-}cell}(c_1, c_2)\,\{\,\langle C, \rho\rangle\,\} \triangleq$
if $\hat{\phi}(c_1, c_2)\langle C, \rho\rangle = \langle\langle c_1, 1\rangle, \langle c_2, 2\rangle\rangle$ **then**
$\{\,\langle C \cup \{\,\langle\langle c_1, 1\rangle, \langle c_2, 2\rangle\rangle\,\}, \rho[\langle\langle c_1, 1\rangle, \langle c_2, 2\rangle\rangle \mapsto v]\rangle \mid v \in \mathbb{E}[\![\,\phi_1(\langle c_1, 1\rangle)(C)\,]\!]\rho\,\}$
else
$\{\,\langle C \cup \{\langle c_1, 1\rangle, \langle c_2, 2\rangle\}, \rho[\forall i :\ \langle c_i, i\rangle \mapsto v_i]\rangle \mid \forall i :\ v_i \in \mathbb{E}[\![\,\phi_i\langle c_i, i\rangle(C)\,]\!]\rho\,\}$

Figure 6.9: Bi-cell addition.

Finally, the predicate *equal* returns *true* in case of success, *false* otherwise.

Single cell synthesis

If all attempts to synthesize a shared bi-cell for the scalar dereferences c_1 and c_2 fail, our memory domain synthesizes the pair of single cells $\langle c_1, 1\rangle$ and $\langle c_2, 2\rangle$ instead. To set the values of $\langle c_1, 1\rangle$ and $\langle c_2, 2\rangle$ soundly, our memory domain calls $\phi_1\langle c_1, 1\rangle(C)$ and $\phi_2\langle c_2, 2\rangle(C)$, where $\phi_k\langle c, k\rangle(C)$ returns a syntactic expression denoting (an abstraction of) the value of $\langle c, k\rangle$ as a function of bi-cells existing in C. Functions $\phi_k \in \widetilde{\mathcal{C}ell} \to \mathcal{P}(\mathcal{B}icell) \to expr$ are defined as simple extensions of the cell synthesizing function ϕ for low-level C programs, which we introduced in Fig. 5.8 of Sec. 5.2.4. To define ϕ_1 and ϕ_2, we project bi-cells of the appropriate side onto cells, apply ϕ, and lift the resulting cell expression back to a bi-cell expression. More precisely, to compute $\phi_1\langle c, 1\rangle(C)$, we first project the bi-cell set C to the cells of $P_1 : C_1 \triangleq \{\,x \mid \langle x, 1\rangle \in C \ \vee\ \exists y :\ \langle\langle x, 1\rangle, \langle y, 2\rangle\rangle \in C\,\}$. Then, we retrieve the constraints on cell c by applying the generic cell synthesizing function: $e_1 \triangleq \phi(c)(C_1)$. Finally, $\phi_1\langle c, 1\rangle(C)$ is obtained by substituting every cell x occurring in e_1 with some bi-cell $x_1 \in occ(\langle x, 1\rangle, C)$.

Remark 28 (Choice of $x_1 \in occ(\langle x, 1\rangle, C)$). e_1 is a syntactic expression over cells in C_1, and $occ(\langle x, 1\rangle, C) \neq \emptyset$ for all $x \in C_1$. We can choose x_1 arbitrarily, as all elements of $occ(\langle x, 1\rangle, C)$ are assumed to hold equal values.

The definition of $\phi_2\langle c, 2\rangle(C)$ is analogue.

Bi-cell addition

Bi-cell addition, $\widehat{add\text{-}cell} \in \mathcal{C}ell_1 \times \mathcal{C}ell_2 \to \mathcal{P}(\hat{\mathcal{D}}) \to \mathcal{P}(\hat{\mathcal{D}})$, then simply adds the synthesized bi-cell(s) to the environment, and initializes their value(s), as shown on Fig. 6.9.

Remark 29 ($\widehat{add\text{-}cell}$ morphism). $\widehat{add\text{-}cell}(c_1, c_2)$ is a complete $\cup$-morphism for all pairs of cells c_1 and c_2.

6.2.4 Semantics of simple statements

Before defining the semantics for double statements in this domain, we need to define the semantics $\hat{\mathbb{E}}_k[\![\,*_t\,e\,]\!] \in \hat{\mathcal{D}} \to \mathcal{P}(\hat{\mathcal{D}}) \times \mathcal{P}(\mathbb{V})$ and $\hat{\mathbb{S}}_k[\![\,*_t\,e_1 \leftarrow e_2\,]\!] \in \mathcal{P}(\hat{\mathcal{D}}) \to \mathcal{P}(\hat{\mathcal{D}})$ for simple memory reads and writes, in program version $k \in \{\,1, 2\,\}$.

Bi-cell synthesis for simple programs.

The transfer functions of assignments and tests rely on bi-cell synthesis for the simple program P_k. As for double programs, the memory model attempts to synthesize shared bi-cells whenever possible for dereferences in the memory of P_k, safely defaulting to single cells on side k.

Shared bi-cell synthesis for simple program P_k. More precisely, the function $\hat{\phi}_k \in \mathcal{C}ell_k \to \hat{\mathcal{D}} \to \mathcal{B}icell \cup \{ \top \}$ formalizes the patterns matched when attempting to synthesize a shared bi-cell for a scalar dereference $c_k \in \mathcal{C}ell_k$ in the memory of P_k:

$$\hat{\phi}_k(c_k) \triangleq \begin{cases} \hat{\phi}(c_1, c_2) \textbf{ if } \exists \langle c_1, c_2 \rangle \in \mathfrak{B} \\ \quad \top \quad \textbf{otherwise} \end{cases}$$

$\hat{\phi}_k$ assumes a relation $\mathfrak{B} \in \mathcal{P}(\mathcal{C}ell_1 \times \mathcal{C}ell_2)$ given between the possible cells in the memory of P_1 and the possible cells in the memory of P_2. For instance, $\hat{\phi}_1(c_1)$ relies on $\mathfrak{B}$ to search for a candidate cell c_2 of P_2, such that $\langle c_1, 1 \rangle = \langle c_2, 2 \rangle$ may be inferred from the environment. It returns the shared bi-cell $\langle \langle c_1, 1 \rangle, \langle c_2, 2 \rangle \rangle$ in case of success, and a failure $\top$ otherwise. $\mathfrak{B}$ is a heuristic used as a hint for the memory domain. Its elements are pairs of cells that are likely to be equal most of the time during program execution. As a heuristic, $\mathfrak{B}$ can influence the precision of the memory abstraction, but not its soundness. Any definition of $\mathfrak{B}$ would be indeed sound: to choose $\mathfrak{B} = \emptyset$ would disable the cell sharing optimization, while $\mathfrak{B} = \mathcal{P}(\mathcal{C}ell_1 \times \mathcal{C}ell_2)$ would maximize cell sharing opportunities, at the cost of efficiency, as $\hat{\phi}_1$ and $\hat{\phi}_2$ would have to test a large number of possibilities.

In our implementation, $\mathfrak{B}$ defines a partial bijection between $\mathcal{C}ell_1$ and $\mathcal{C}ell_2$. It is computed by the analysis front-end, and contains pairs of cells likely to be equal most of the time during program execution, namely pairs of cells from different program versions that represent the "same" scalar variables or fields in P_1 and P_2. Indeed our implementation of $\mathfrak{B}$ matches scalar fields which enjoy the same names, types and scopes in the AST of P_1 and P_2. For instance, natural candidates of Example 34 are the cells inside field `s.b` and variable `p`, which occur in both program versions: $\langle \langle \mathtt{s}_1, 2, \mathbf{u16} \rangle, \langle \mathtt{s}_2, 0, \mathbf{u16} \rangle \rangle \in \mathfrak{B}$ and $\langle \langle \mathtt{p}, 0, \mathbf{ptr} \rangle, \langle \mathtt{p}, 0, \mathbf{ptr} \rangle \rangle \in \mathfrak{B}$. In contrast, we do not attempt to match the cells inside field `s.a` of P_1, as this field has no counterpart in P_2.

Remark 30 (Names of fields versus offsets of cells). Cells corresponding to similar path, s.b, can have different offsets. For instance, the cell corresponding to `s.b` has offset 2 in P_1 and 0 in P_2. Our heuristics favors cells with the same field name over cells with the same offset, to detect the likelihood of new fields being inserted in structs and shifting the position in memory of similar data between P_1 and P_2.

For each pair of matched cells $c_1 = \langle V_1, o_1, t \rangle$ and $c_2 = \langle V_2, o_2, t \rangle$, $\mathfrak{B}$ additionally matches any possible cells $c_1' = \langle V_1, o_1 + p, t' \rangle$ and $c_2' = \langle V_2, o_2 + p, t' \rangle$ such that $p + \mathit{sizeof}(t') \leq \mathit{sizeof}(t)$, *i.e.* c_1' and c_2' represent corresponding sequences of bytes included

$\widehat{add\text{-}cell}_k(c_k)\,\{\,\langle C,\rho\rangle\,\} \triangleq$
  if $\hat{\phi}(c_k)\langle C,\rho\rangle = \langle\langle c_1,1\rangle,\langle c_2,2\rangle\rangle$ **then**
    $\{\,\langle C \cup \{\,\langle\langle c_1,1\rangle,\langle c_2,2\rangle\rangle\,\}, \rho[\langle\langle c_1,1\rangle,\langle c_2,2\rangle\rangle \mapsto v]\rangle \mid v \in \mathbb{E}[\![\,\phi_1(\langle c_1,1\rangle)(C)\,]\!]\rho\,\}$
  else
    $\{\,\langle C \cup \{\langle c_k,k\rangle\}, \rho[\langle c_k,k\rangle \mapsto v]\rangle \mid v \in \mathbb{E}[\![\,\phi_k\langle c_k,k\rangle(C)\,]\!]\rho\,\}$

Figure 6.10: Bi-cell addition for simple program P_k.

in those of c_1 and c_2. For Example 34:

$$\begin{array}{rcl}\mathfrak{B} & \triangleq & \{\,\langle\langle \mathtt{s}_1,o,\tau\rangle,\langle \mathtt{s}_2,o-2,\tau\rangle\rangle \mid 2\le o \wedge o+\mathit{sizeof}(\tau)\le 4\,\}\\ & \cup & \{\,\langle\langle \mathtt{p},o,\tau\rangle,\langle \mathtt{p},o,\tau\rangle\rangle \mid 0\le o \wedge o+\mathit{sizeof}(\tau)\le 4\,\}\end{array}$$

$\mathfrak{B}$ matches offsets of variables defined in P_1 and P_2 with equal names but different types. It also matches offsets of versions of variables when P_1 and P_2 are compiled with different options or attributes, changing the layout of memory. The restriction of $\mathfrak{B}$ to the set of variables defined in P_1 and P_2 with identical names and types is the identity.

Single cell synthesis for simple program P_k. If all attempts to synthesize a shared bi-cell for the scalar dereference c_k fail, our memory domain synthesizes the single single cell $\langle c_k,k\rangle$ instead. As in Sec. 6.2.3, it uses $\phi_k\langle c_k,k\rangle(C)$ to set its value soundly.

Bi-cell addition for simple program P_k. Bi-cell addition, $\widehat{add\text{-}cell}_k \in \mathcal{C}ell_k \to \mathcal{P}(\hat{\mathcal{D}}) \to \mathcal{P}(\hat{\mathcal{D}})$, then simply adds the synthesized bi-cell to the environment, and initializes its value, as shown on Fig. 6.10.

Remark 31 ($\widehat{add\text{-}cell}_k$ morphism). As in Remark 29, $\widehat{add\text{-}cell}_k(c_k)$ is a complete $\cup$-morphism for all $c_k \in \mathcal{C}ell_k$.

Evaluations.

We describe the semantics of $\hat{\mathbb{E}}_k[\![\,*_t\,e\,]\!]\,\langle C,\rho\rangle$, assuming the expression e does not contain any dereference. This is not restrictive, as expressions can be transformed into purely scalar expressions by resolving left-values bottom up. To compute $\hat{\mathbb{E}}_k[\![\,*_t\,e\,]\!]\,\langle C,\rho\rangle$, we first resolve $*_t\,e$ into a set L_k of single cells on side k, by evaluating e into a set of pointer values, and gathering single cells corresponding to valid pointers:

$$L_k \triangleq \{\,\langle V,o,t,k\rangle \in \widetilde{\mathcal{C}ell} \mid \langle V,o\rangle \in \mathbb{E}_k[\![\,e\,]\!]\,\rho\,\}$$

Then, we call $\widehat{add\text{-}cell}_k$ to ensure that all the target bi-cells in L_k occur in the abstract environment, either directly or via a suitable shared bi-cell. This updates the singleton state $\{\,\langle C,\rho\rangle\,\}$ to a set of states $X_0 \in \mathcal{P}(\hat{\mathcal{D}})$:

$$X_0 = \Big(\widehat{add\text{-}cell}_k(c_n) \circ \cdots \circ \widehat{add\text{-}cell}_k(c_1)\Big)\,\{\,\langle C,\rho\rangle\,\}$$

where $\{ \langle c_1, k\rangle, \ldots, \langle c_n, k\rangle \} = L_k$. The semantics of $\widehat{add\text{-}cell}_k$ ensures that $occ(c, C_0) \neq \emptyset$ for all $\langle C_0, \rho_0\rangle \in X_0$ and $c \in L_k$.
Finally,

$$\hat{\mathbb{E}}_k[\![*_t e]\!] \langle C, \rho\rangle = \langle X_0, \{ \rho_0(c_0) \mid c_0 \in occ(c, C_0) \wedge c \in L_k \wedge \langle C_0, \rho_0\rangle \in X_0 \}\rangle$$

Assignments.

The semantics of assignments $\hat{\mathbb{S}}_k[\![*_t e_1 \leftarrow e_2]\!]$ involves more steps. We describe $\hat{\mathbb{S}}_k[\![*_t e_1 \leftarrow e_2]\!] \{ \langle C, \rho\rangle \}$ on a single state $\langle C, \rho\rangle$, as $\hat{\mathbb{S}}_k[\![*_t e_1 \leftarrow e_2]\!]$ is a complete $\cup$-morphism.

We first evaluate e_2 into a set of values $\mathbb{V}_{e_2} \in \mathcal{P}(\mathbb{V})$. This may involve synthesizing bi-cells for the dereferences of e_2, which updates the singleton state $\{ \langle C, \rho\rangle \}$ to a set of states $\hat{X} \in \mathcal{P}(\hat{\mathcal{D}})$: $\langle \hat{X}, \mathbb{V}_{e_2}\rangle = \hat{\mathbb{E}}_k[\![e_2]\!] \langle C, \rho\rangle$. and $\langle \hat{C}, \hat{\rho}\rangle \in \hat{X}$.

Like for evaluations, we assume the expression e_1 does not contain any dereference, and start with resolving $*_t e_1$ into a set L_k of single cells on side k. Then, we realize the bi-cells in L_k using $\widehat{add\text{-}cell}_k$, which updates the singleton state $\{ \langle \hat{C}, \hat{\rho}\rangle \}$ to a set of states $X_0 \in \mathcal{P}(\hat{\mathcal{D}})$. Let $\langle C_0, \rho_0\rangle \in X_0$. Some of the single cells in L_k may have been realized into shared bi-cells in C_0. Let $S_0 \triangleq (C_0 \setminus C) \cap \widetilde{Cell}^2$ be the set of such shared bi-cells. Elements of S_0 represent equalities between single cells on side k, and on side opposite to k. Such equalities may no longer hold, after assignment on side k. Therefore, we split shared bi-cells of S_0 into their left and right projections, in a copy-on-write strategy. Each state $\langle C_0, \rho_0\rangle \in X_0$ is thus updated to $\langle C_0', \rho_0'\rangle = split(S_0, \langle C_0, \rho_0\rangle) \in X_0'$, where $split \in \mathcal{P}(\widetilde{Cell}^2) \times \mathcal{P}(\hat{\mathcal{D}}) \to \mathcal{P}(\hat{\mathcal{D}})$ is defined as

$$C_0' \triangleq C_0 \cup \bigcup_{\langle c, c'\rangle \in S_0} \{ c, c' \} \qquad \text{and} \qquad \rho_0'(c) \triangleq \begin{cases} \rho_0(x) \textbf{ if } \exists x \in occ(c, S_0) \neq \emptyset \\ \rho_0(c) \textbf{ otherwise}. \end{cases} \tag{6.1}$$

Finally, we update the environment for the single cells written (elements of L_k), with the possible values of e_2. However, this is not sufficient: it is also necessary to update the environment for any overlapping bi-cells, including shared bi-cells that have been split into pairs of single cells.

Indeed, removing any bi-cell is always sound in our memory model: it amounts to losing information, as we lose constraints on the byte-representation of the memory. Let $\Omega_0' \subseteq C_0' \setminus L_k$ be the set of such bi-cells: elements of Ω_0' are shared bi-cells and single cells on side k, with base variables, offsets and sizes such that they overlap some element of L_k. The updated environment is:

$$\hat{\mathbb{S}}_k[\![*_t e_1 \leftarrow e_2]\!] \{ \langle C, \rho\rangle \} = \bigcup_{\langle C_0', \rho_0'\rangle \in X_0'} \{ \langle C_0' \setminus \Omega_0', \rho_{|C_0' \setminus \Omega_0'}[\forall c \in L_k : c \mapsto v]\rangle \mid v \in \mathbb{V}_{e_2} \}$$

Remark 32 (Recovering shared bi-cells after simple assignment). As explained above, removing all shared bi-cells that have been split in a copy-on-write strategy may result in loss of precision after numerical abstraction in non relational domains. A possible mitigation is to eagerly attempt to synthesize again the shared bi-cells that have been

split, immediately after the assignment on side k. An alternate strategy is to lazily let a subsequent memory reads from the program trigger such attempts.

In addition, an existing shared bi-cell such as $\langle\langle \mathtt{x}, 0, \mathbf{u32}, 1\rangle, \langle \mathtt{x}, 0, \mathbf{u32}, 2\rangle\rangle$ should not be removed when both program versions execute jointly commonplace statements such as `x=42;` and `x=x+1;`. This is ensured by the transfer function of double assignment statements, which will be presented in Sec. 6.2.5.

6.2.5 Semantics of double statements

We are now ready to define the semantics $\hat{\mathbb{D}}[\![\, dstat \,]\!] \in \mathcal{P}(\hat{\mathcal{D}}) \to \mathcal{P}(\hat{\mathcal{D}})$ of double statements in this domain. Like $\mathbb{D}$, $\hat{\mathbb{D}}$ is defined by induction on the syntax. We focus on base cases, as inductive cases are unchanged.

The semantics for **input_sync**, **assert_sync**$(\cdot)$, and $\hat{\mathbb{F}}[\![\, e_1 \bowtie 0 \parallel e_2 \bowtie 0 \,]\!]$ are mostly unchanged, but for symbolic simplifications taking advantage of symbolic representations of equalities in our memory domain, in order to maximize efficiency and precision after numerical abstraction in non relational domains. In particular, when l is a deterministic left-value expression $*_\tau e$ containing a single dereference, then $\hat{\mathbb{D}}[\![\, l \leftarrow \mathbf{input\_sync}(a, b) \,]\!]$ adds a shared bi-cell for this dereference to every abstract state. Consistently, $\hat{\mathbb{D}}[\![\, \mathbf{assert\_sync}(l) \,]\!]$ first tests whether l is a left-value expression $*_\tau e$ that evaluates to a single shared bi-cell. If such is the case, then l is guaranteed to evaluate to equal values in all double memory states, so $\hat{\mathbb{D}}[\![\, \mathbf{assert\_sync}(l) \,]\!]$ raises no alarm. Otherwise, the semantics uses environment functions ρ to test equalities of bi-cell values in every state, as for $\mathbb{D}$. A similar symbolic simplification is used for the $\hat{\mathbb{F}}$ filter: $\hat{\mathbb{F}}[\![\, e \bowtie 0 \parallel e \not\bowtie 0 \,]\!]\langle C, \rho\rangle = \emptyset$ (hence the test $e \bowtie 0$ is stable) when e is deterministic and all dereferences evaluate to shared bi-cells, which is the common case. For instance, when evaluating $\hat{\mathbb{D}}[\![\, \mathbf{if}\ (x < y)\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!]$, if the dereferences for variables `x` and `y` evaluate to shared bi-cells in every state, then the two unstable tests cases are $\bot$.

Assignments.

In an assignment $\hat{\mathbb{D}}[\![\, *_t e_1 \leftarrow e_2 \,]\!]$, although both programs execute the same syntactic assignment, their semantics may differ, as the memory layouts may be different. For instance, recall Example 3 from Sec. 1.1.4. The value assigned to field `s.x` by the statement `p[4]=1` depends on the alignments of types defined by the ABI, or compiler options and directives. In addition, available bi-cells may be different. By default, double assignments are straightforward extensions of simple assignments: $\hat{\mathbb{D}}[\![\, *_t e_1 \leftarrow e_2 \,]\!] = \hat{\mathbb{S}}_2[\![\, *_t e_1 \leftarrow e_2 \,]\!] \circ \hat{\mathbb{S}}_1[\![\, *_t e_1 \leftarrow e_2 \,]\!]$. We introduce two precision optimizations, taking advantage of implicit equalities represented by shared bi-cells. We discuss these precision optimizations on a singleton state $\{\, \langle C, \rho\rangle \,\}$, as the transfer function is a complete $\cup$-morphism. We first transform $*_t e_1$ and the dereferences in e_2 into sets of bi-cells L and R, respectively. Note that R may be empty, as e_2 may be a constant expression. Then, we realize the cells in L and R, using $\widehat{add\text{-}cell}$, which updates the singleton state $\{\, \langle C, \rho\rangle \,\}$ to a set of states $X_0 \in \mathcal{P}(\hat{\mathcal{D}})$. Two optimizations are possible, depending on e_1, e_2, L, and R.

Optimization 1: Assignment of shared bi-cells. If $*_t\, e_1$ and e_2 are deterministic expressions, and if they evaluate to bi-cells that are all shared ($L \cup R \subseteq \widetilde{\mathcal{C}ell}^2$) in every state $\langle C_0, \rho_0\rangle \in X_0$, then P_1 and P_2 write the same value to the same destination. We thus update shared destination bi-cells (in L), and remove any overlapping bi-cells. Formally:

$$\hat{\mathbb{D}}[\![*_t\, e_1 \leftarrow e_2]\!]\, \{\, \langle C, \rho\rangle \,\} = \bigcup_{\langle C_0,\rho_0\rangle \in X_0} \{\, \langle C_0 \backslash \Omega_0, \rho_{|C_0\backslash\Omega}[\forall c \in L :\, c \mapsto v]\rangle \mid v \in \mathit{snd}(\hat{\mathbb{E}}_1[\![e_2]\!]\, \langle C_0, \rho_0\rangle) \,\}$$

where $\Omega_0 \subseteq \mathcal{C}_0 \setminus L$ is the set of (shared or single) bi-cells overlapping elements of L.

Remark 33 (Evaluation on arbitrary side after bi-cell synthesis)*.* The choice of evaluating $\hat{\mathbb{E}}_1[\![e_2]\!]$ (rather than $\hat{\mathbb{E}}_2[\![e_2]\!]$) is arbitrary, as they are equal. Indeed, all the necessary cells are materialized before evaluating expression e_2.

Example 35 (Assignment of shared bi-cells)**.** The case occurs typically in commonplace assignments such as `x=x+1;`. Fig. 6.11 shows such an example, where a shared bi-cell represents the bytes of `x` for all states before the assignment, and a counterexample where this is not the case.

This optimization can be extended to the case where $*_t\, e_1$ is not resolved to a shared bi-cell, but to a pair of single cells: c_1 for P_1, and c_2 for P_2. In this case, we replace c_1 and c_2 by $\langle c_1, c_2\rangle$ in L, and apply the same transfer function.

Example 36 (Extended assignment of shared bi-cells)**.** The case occurs in assignments such as `x=42;` Fig. 6.11 shows such an example.

Optimization 2: Copy assignment. If the conditions for optimization *1* are satisfied, and if, in addition, $e_2 = *_t\, e_2'$, and both $*_t\, e_1$ and $*_t\, e_2'$ evaluate to single bi-cells ($|L| = |R| = 1$) in every state $\langle C_0, \rho_0\rangle \in X_0$, then we are dealing with a copy assignment, as in `y=x;`. We may thus soundly copy any memory information from the source $\{r\} = R$, to the destination $\{l\} = L$, so as to further improve precision. We therefore remove l, and create a copy of r, and of any smaller bi-cell $r' \in C_0$ for the same bytes, to a corresponding bi-cell for the bytes of l. Newly created destination bi-cells have the sides of their sources. The environment is updated accordingly, to reflect equalities between sources and destinations.

Example 37 (Copy assignment)**.** The case occurs copies between variables, such as `y=x;`. Fig. 6.11 shows such an example, where the bi-cell for `y` is preserved thanks to that for `x`, and a counterexample with the bi-cell for `y` is removed as there is no bi-cell for `x`.

This optimization can be extended to the case where $*_t\, e_1$ is not resolved to a shared bi-cell, but to a pair of single cells: c_1 for P_1, and c_2 for P_2. In this case, we replace c_1 and c_2 by $\langle c_1, c_2\rangle$ in L, and apply the same transfer function.

Example 38 (Extended copy assignment)**.** The last equality in Fig. 6.11 shows such an example. `y=x;` creates a shared bi-cell for `y`.

$\hat{\mathbb{D}}[\![x \leftarrow x+1]\!] \{ \langle \{ x_1^0, x_2^0, x_{12} \}, [x_1^0 \mapsto 0, x_2^0 \mapsto 0, x_{12} \mapsto 0] \rangle, \langle \{ x_1^3, x_2^0, x_{12} \}, [x_1^3 \mapsto 0, x_2^0 \mapsto 1, x_{12} \mapsto 1] \rangle \}$
$= \{ \langle \{ x_{12} \}, [x_{12} \mapsto 1] \rangle, \langle \{ x_{12} \}, [x_{12} \mapsto 2] \rangle \}$

$\hat{\mathbb{D}}[\![x \leftarrow x+1]\!] \{ \langle \{ x_1^0, x_2^0, x_{12} \}, [x_1^0 \mapsto 0, x_2^0 \mapsto 0, x_{12} \mapsto 0] \rangle, \langle \{ x_1, x_2 \}, [x_1 \mapsto 1, x_2 \mapsto 2] \rangle \}$
$= \{ \langle \{ x_{12} \}, [x_{12} \mapsto 1] \rangle, \langle \{ x_1, x_2 \}, [x_1 \mapsto 2, x_2 \mapsto 3] \rangle \}$

$\hat{\mathbb{D}}[\![x \leftarrow 42]\!] \{ \langle \{ x_1^1, x_2^0, x_1 \}, [x_1^1 \mapsto 1, x_2^0 \mapsto 0, x_1 \mapsto 256] \rangle, \langle \{ x_1 \}, [x_1 \mapsto 1] \rangle \}$
$= \{ \langle \{ x_{12} \}, [x_{12} \mapsto 42] \rangle \}$

$\hat{\mathbb{D}}[\![y \leftarrow x]\!] \{ \langle \{ x_1^0, x_2^3, x_{12}, y_1^1, y_{12} \}, [x_1^0 \mapsto 8, x_2^3 \mapsto 0, x_{12} \mapsto 8, y_1^1 \mapsto 1, y_{12} \mapsto 256] \rangle \}$
$= \{ \langle \{ x_1^0, x_2^3, x_{12}, y_1^0, y_2^3, y_{12} \}, [x_1^0 \mapsto 8, x_2^3 \mapsto 0, x_{12} \mapsto 8, y_1^0 \mapsto 8, y_2^3 \mapsto 0, y_{12} \mapsto 8] \rangle \}$

$\hat{\mathbb{D}}[\![y \leftarrow x]\!] \{ \langle \{ x_1^0, x_2^3, y_1^1, y_{12} \}, [x_1^0 \mapsto 8, x_2^3 \mapsto 0, x_{12} \mapsto 8, y_1^1 \mapsto 1, y_{12} \mapsto 256] \rangle \}$
$= \{ \langle \{ x_1^0, x_2^3, y_1^0, y_2^3 \}, [x_1^0 \mapsto 8, x_2^3 \mapsto 0, y_1^0 \mapsto 8, y_2^3 \mapsto 0] \rangle \}$

$\hat{\mathbb{D}}[\![y \leftarrow x]\!] \{ \langle \{ x_1^0, x_2^3, x_{12}, y_1^1 \}, [x_1^0 \mapsto 8, x_2^3 \mapsto 0, x_{12} \mapsto 8, y_1^1 \mapsto 1] \rangle \}$
$= \{ \langle \{ x_1^0, x_2^3, x_{12}, y_1^0, y_2^3, y_{12} \}, [x_1^0 \mapsto 8, x_2^3 \mapsto 0, x_{12} \mapsto 8, y_1^0 \mapsto 8, y_2^3 \mapsto 0, y_{12} \mapsto 8] \rangle \}$

where

$x_k = \langle \mathtt{x}, 0, \mathbf{int}, k \rangle \qquad x_k^o = \langle \mathtt{x}, o, \mathbf{u8}, k \rangle \qquad (o,k) \in \{0,1,2,3\} \times \{1,2\}$
$y_k = \langle \mathtt{y}, 0, \mathbf{int}, k \rangle \qquad x_{12} = \langle x_1, x_2 \rangle \qquad y_{12} = \langle y_1, y_2 \rangle$

Figure 6.11: Assignments to shared bi-cells (Examples 35, 36, 37 and 38)

$$(\mathcal{P}(\hat{\mathcal{D}}), \preccurlyeq) \underset{\alpha^\flat}{\overset{\gamma^\flat}{\leftrightarrows}} (\hat{\mathcal{D}}^\flat, \preccurlyeq^\flat)$$

$\alpha^\flat(X) \triangleq \langle \bar{C}, \{ \rho_{|\bar{C}} \mid \langle C, \rho \rangle \in X \} \rangle$ where $\bar{C} = \bigcap \{ C \mid \langle C, \rho \rangle \in X \}$

$\gamma^\flat \langle C, R \rangle \triangleq \{ \langle C, \rho \rangle \mid \rho \in R \}$ and $\langle C, R \rangle \preccurlyeq^\flat \langle C', R' \rangle \overset{\triangle}{\iff} C' \subseteq C \wedge \{ \rho_{|C'} \mid \rho \in R \} \subseteq R'$

Figure 6.12: Unified bi-cell environments

6.2.6 Unification

Like for $\mathbb{S}$ in Sec. 5.2.4 and $\mathbb{D}$ in Sec. 5.3.3, we aim at abstracting $\hat{\mathbb{D}}$ using numerical domains, which naturally represent sets of environments with homogenous support. Yet states in $\hat{\mathcal{D}}$ have heterogeneous bi-cell support, so we first unify sets of bi-cells. To this aim, we only retain bi-cells that are part of every state in a set, soundly disregarding all others. Our new domain is thus a choice of a set of bi-cells C and a set of scalar environments on C:

$$\hat{\mathcal{D}}^\flat \triangleq \bigcup_{C \subseteq \mathcal{B}icell} \{ \langle C, R \rangle \mid R \in \mathcal{P}(C \to \mathbb{V}) \}$$

A formalization is shown on Fig. 6.12. Note that we reuse the partial order $\preccurlyeq^\flat$ and the abstraction and concretization $\alpha^\flat$ and $\gamma^\flat$ defined Fig. 5.9 of Sec. 5.2.4 for simple C

programs. The definitions are indeed identical, albeit for different sets.

Proposition 14. *The pair $(\alpha^\flat, \gamma^\flat)$ defined in Fig. 6.12 is a Galois connection.*

The adaptation of $\widehat{add\text{-}cell}^\flat \in \mathcal{C}ell \to \hat{\mathcal{D}}^\flat \to \hat{\mathcal{D}}^\flat$ to this new domain is straightforward. Adaptations of transfer functions $\hat{\mathbb{D}}^\flat[\![stat]\!] \in \mathcal{D}^\flat \to \mathcal{D}^\flat$ are also straightforward, assuming a sound abstract join $\hat{\sqcup}^\flat$ is provided. $\hat{\sqcup}^\flat$ must merge environment sets defined on heterogeneous bi-cell sets. We therefore define a unification function $\widehat{unify}^\flat \in (\hat{\mathcal{D}}^\flat)^2 \to (\hat{\mathcal{D}}^\flat)^2$. $\widehat{unify}^\flat(\langle C_1, R_1\rangle, \langle C_2, R_2\rangle)$ adds, with $\widehat{add\text{-}cell}^\flat$, any missing cells to $\langle C_1, R_1\rangle$ and $\langle C_2, R_2\rangle$: respectively $C_2 \setminus C_1$ and $C_1 \setminus C_2$. Let $\langle C_1', R_1'\rangle$ and $\langle C_2', R_2'\rangle$ be the resulting abstract states. C_1' and C_2' may include both single cells and shared bi-cells. A shared bi-cell that does not occur in both C_1' and C_2' cannot be soundly included in the unified state, as it conveys equality information that holds for one abstract state only. Such shared bi-cells are elements of $\Omega = ((C_1' \cup C_2') \setminus \widetilde{\mathcal{C}ell}) \setminus (C_1' \cap C_2')$. Elements of Ω are first split into their left and right projections, and then removed before unification.

Formally, let $\Omega_k = \Omega \cap C_k$ and $\langle C_k'', R_k''\rangle = split^\flat(\Omega_k, \langle C_k', R_k'\rangle)$, for $k \in \{1, 2\}$, where $split^\flat \in \mathcal{P}(\widetilde{\mathcal{C}ell}^2) \times \hat{\mathcal{D}}^\flat \to \hat{\mathcal{D}}^\flat$ is defined as the function *split* defined by Equation 6.1.

$$\widehat{unify}^\flat(\langle C_1, R_1\rangle, \langle C_2, R_2\rangle) = (\langle C_{12}, R_1'''\rangle, \langle C_{12}, R_2'''\rangle)$$

where $C_{12} = (C_1'' \cup C_2'') \setminus \Omega$, and $R_k''' = \{ \rho_{|C_{12}} \mid \rho \in R_k'' \}$.

The abstract join may now be defined as

$$\langle C_1, R_1\rangle \hat{\sqcup}^\flat \langle C_2, R_2\rangle \triangleq \langle C_{12}, R_1''' \cup R_2'''\rangle$$

Example 39 (Comparing $\hat{\mathbb{D}}$ and $\hat{\mathbb{D}}^\flat$)**.** The slightly more abstract $\hat{\mathbb{D}}^\flat$ is oblivious of some symbolic information expressed by $\hat{\mathbb{D}}$. Consider, for instance, an assignment `*p =33;` in a environment where the pointer `p` may point either to variable `a`, or to variable `b`. As shown by Fig. 6.13, $\hat{\mathbb{D}}$ synthesizes only shared bi-cells, and distinguishes states where a bi-cell is synthesized for `b`, from states where no bi-cell is synthesized for `b`. In contrast, unification always synthesizes single bi-cells for `b`, with arbitrary values in most environments.

6.2.7 Value abstraction

We finally rely on numerical abstractions to abstract further $\hat{\mathbb{D}}^\flat$ into a computable abstract semantics $\hat{\mathbb{D}}^\sharp$, resulting in an effective static analysis. As in Sec. 5.3.4, our memory domain translates memory reads and writes into purely numerical operations on synthetic bi-cells, that are oblivious to the double semantics of double programs: each bi-cell is viewed as an independent numeric variable, and each numeric operation is carried out on a single bi-cell store, as if emanated from a single program. In particular, we notice that the transfer function for simple assignments $\hat{\mathbb{S}}_k[\![*_t e_1 \leftarrow e_2]\!]$ described in Sect. 6.2.4 has

$$\hat{\mathbb{D}}[\![*p \leftarrow 33]\!] \{ \langle \{ a_{12}, p_{12} \}, [a_{12} \mapsto 0, p_{12} \mapsto \langle a, 0 \rangle] \rangle, \langle \{ a_{12}, p_{12} \}, [a_{12} \mapsto 0, p_{12} \mapsto \langle b, 0 \rangle] \rangle \}$$
$$= \{ \langle \{ a_{12}, p_{12} \}, [a_{12} \mapsto 33, p_{12} \mapsto \langle a, 0 \rangle] \rangle, \langle \{ a_{12}, b_{12}, p_{12} \}, [a_{12} \mapsto 0, b_{12} \mapsto 33, p_{12} \mapsto \langle b, 0 \rangle] \rangle \}$$

$$\hat{\mathbb{D}}^\flat[\![*p \leftarrow 33]\!] \langle \{ a_{12}, p_{12} \}, \{ [a_{12} \mapsto 0, p_{12} \mapsto \langle a, 0 \rangle], [a_{12} \mapsto 0, p_{12} \mapsto \langle b, 0 \rangle] \} \rangle =$$
$$\langle \{ a_{12}, b_1, b_2, p_{12} \}, \bigcup_{v_1, v_2 \in \mathbb{Z}} \{ [a_{12} \mapsto 33, p_{12} \mapsto \langle a, 0 \rangle, b_1 \mapsto v_1, b_2 \mapsto v_2] \} \cup$$
$$\{ [a_{12} \mapsto 0, p_{12} \mapsto \langle b, 0 \rangle, b_1 \mapsto 33, b_2 \mapsto 33] \} \rangle$$

where

$a_{12} = \langle \langle \mathtt{a}, 0, \mathbf{int}, 1 \rangle, \langle \mathtt{a}, 0, \mathbf{int}, 2 \rangle \rangle$ $\quad b_{12} = \langle b_1, b_2 \rangle$

$p_{12} = \langle \langle \mathtt{p}, 0, \mathbf{ptr}, 1 \rangle, \langle \mathtt{p}, 0, \mathbf{ptr}, 2 \rangle \rangle$ $\quad b_k = \langle \mathtt{b}, 0, \mathbf{int}, k \rangle$ $\quad k \in \{ 1, 2 \}$

Figure 6.13: Comparing $\hat{\mathbb{D}}$ and $\mathbb{D}^\flat$ (Example 39)

the form of that of an assignment in a purely numerical language, where bi-cells play the roles of the numeric variables. This property is a key motivation for the Cell domain and the extension presented in this chapter. Bi-cells may thus be fed, as variables, to a numerical abstract domain for environment abstraction. Any standard numerical domain may be used, such as intervals, congruences and polyhedra.

We thus assume an abstract domain $\hat{\mathcal{D}}^\sharp_C$ given, with concretization $\hat{\gamma}_C$, for each bi-cell set $C \subseteq \mathcal{B}icell$. It abstracts $\mathcal{P}(C \to \mathbb{Z}) \simeq \mathcal{P}(\mathbb{Z}^{|C|})$, i.e., sets of points in a $|C|-$dimensional vector space. A cell of integer type naturally corresponds to a dimension in an abstract element. We also associate a distinct dimension to each bi-cell with pointer type; it corresponds to the offset o of a symbolic pointer $\langle V, o \rangle \in \mathcal{P}tr$. In order to abstract fully pointer values, we enrich the abstract numerical environment with a map P associating to each pointer bi-cell the set of variables it may point to. Hence, the abstract domain becomes: $\hat{\mathcal{D}}^\sharp \triangleq \{ \langle C, R^\sharp, P \rangle \mid C \subseteq \mathcal{B}icell,\ R^\sharp \in \hat{\mathcal{D}}^\sharp_C,\ P \in P_C \to \mathcal{P}(\mathcal{V} \cup \{ \mathbf{NULL}, \mathbf{invalid} \}) \}$, where $P_C \subseteq C$ is the subset of bi-cells of pointer type.

6.3 Implementation

We implemented a static analysis based on the $\hat{\mathbb{D}}^\sharp$ abstract semantics on top of Mopsa. The only difference, with respect to the $\tilde{\mathbb{D}}^\sharp$ semantics presented in Sec. 5.3.3, is our optimized bi-cell based memory model. This change is reflected in a modular way in the configuration of the analysis shown on Fig. 6.14. The only change, with respect to Fig. 5.14 from Sec. 5.3.5, is the memory abstraction, composed of domain-modules D.cells, C.machineNum, and C.pointers. In contrast, all iterator and numerical domains-modules are unchanged in the configuration of the analysis.

The memory abstraction is updated by replacing the composition D.patch ∘ C.cells from Fig. 5.14 with the domain D.cells on Fig. 6.14. D.cells stands for the bi-cell abstract domain. It handles atomic statements and tests. To this aim, it decomposes program variables into a set of synthetic scalar variables representing bi-cells. It implements in particular $\hat{\phi}^\flat$, a straightforward adaptation of the bi-cell synthesize function $\hat{\phi}$ from

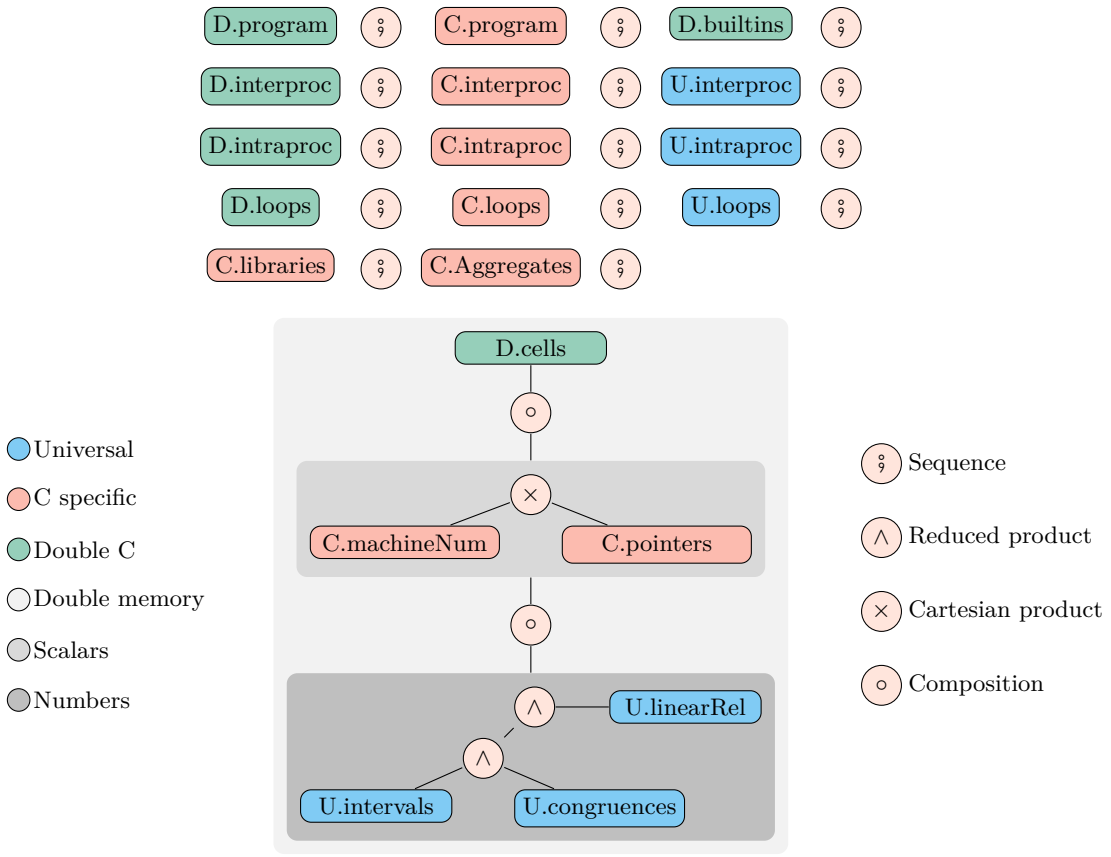


Figure 6.14: Analysis of double C programs with bi-cells

Sec. 6.2.3 to the $\hat{\mathcal{D}}^b$ domain of Sec. 6.2.6.

Synthetic bi-cells are then handled by a Cartesian product: C.machineNum × C.pointers rewrites scalar expressions into mathematical expressions on unbounded integers, which are then handled natively by classic Universal numerical abstract domains.

In our implementation, the new D.cells domain accounts for 3,400 lines of OCaml.

6.4 Evaluation

In this section, we give an experimental evaluation of our implementation. We evaluate it against a set of C benchmarks already presented in Sec. 5.4.

6.4.1 Real-world patches

The first set of benchmarks has been introduced in Fig. 5.21 of Sec. 5.4.2. It is composed of real patches of the GNU core utilities and the Linux GitHub repository. Benchmarks

Bench.	LOC	#P	cell based $\tilde{\mathcal{D}}^\sharp$				bi-cell based $\hat{\mathcal{D}}^\sharp$					
			polyhedra		octagon		polyhedra		octagon		interval	
copy[1]	95	1	157 ms	✓	482 ms	✓	113 ms	✓	156 ms	✓	41 ms	✓
seq[1]	46	16	570 ms	✓		✗	442 ms	✓		✗		✗
pr[1]	114	8	1.421 s	✓	6.469 s	✓	4.642 s	✓	3.723	✓	88 ms	✓
test[1]	352	10	9.188 s	✓		✗	440 ms	✓	1.163 s	✓	96 ms	✓
kvm[2]	248	1/11	2.707 s	✓	4.214 s	✓	1.426 s	✓	1.568 s	✓	96 ms	✓
sched[2]	194	7/12	65 ms	✓		✗	63 ms	✓	104 ms	✓	38 ms	✓
dma[2]	270	5/23	285 ms	✓	1.235 s	✓	216 ms	✓	584 ms	✓	76 ms	✓
block[2]	324	22/6	80 ms	✓		✗	67 ms	✓	121 ms	✓	31 ms	✓
iucv[2]	179	10/9	403 ms	✓	1.757 s	✓	7.721 s	✓	14.423 s	✓	426 ms	✓
io_uring[2]	1569	10/14	868.701 s	✓		✗	594.481 s	✓	4170.295s	✓	288 ms	✓

Figure 6.15: Analyses of real patches from Coreutils and Linux

`copy` to `test` feature algorithmic patches, while benchmarks `kvm` to `io_uring` feature changes in data structures.

We analyze these benchmarks with the bi-cell based memory model introduced in this chapter, together with three numerical domains: polyhedra, octagons and intervals. The results are shown on Fig. 6.15. We compare these results with the results of reference analyses of the same benchmarks with the cell based memory model introduced in Chapter 5. We only show results with the polyhedra and octagon numerical domains for these reference analyses, as the interval domain allows no successful patch analysis of these benchmarks with the cell based memory model. Note nonetheless that the running times of analyses with the interval domain are very similar with both memory models, which suggests that the bi-cell based memory model does not significantly increase the cost of the memory abstraction. Fig. 6.15 shows that the bi-cell based memory model does not significantly improve performance of successful analyses with the polyhedra domain. This suggests that the reduction of the number of dimensions in the numerical abstraction allowed by bi-cell sharing is compensated by the polyhedral operations triggered by bi-cell synthesis. However, our bi-cell based memory model allows improving the precision of analyses with numerical domains less expressive than polyhedra. In particular, it allows successful analyses of most benchmarks with the interval domain, which dramatically improves the scalability of the analysis. Consider for instance the `io_uring` benchmark, composed of 1,500 lines of C. This benchmark is analyzed successfully with bi-cells over the interval domain, whereas the cell based memory model requires the polyhedra domain. The bi-cell based model allows thus a faster analysis, by three orders of magnitude.

6.4.2 Synthetic benchmarks

The second set of benchmarks is composed of patches of small synthetic C programs from the related works. These benchmarks have already been presented as part of

[1] Coreutils

[2] Linux

Benchmark	LOC	#P	cell based $\tilde{\mathcal{D}}^\sharp$				bi-cell based $\hat{\mathcal{D}}^\sharp$					
			polyhedra		octagon		polyhedra		octagon		interval	
Comp	13	2	48 ms	✓		✗	107 ms	✓	209 ms	✓		✗
Const	9	3	28 ms	✓		✗	38 ms	✓	49 ms	✓		✗
Fig. 2	14	1	31 ms	✓	39 ms	✓	40 ms	✓	47 ms	✓	25 ms	✓
LoopMult	14	2	166 ms	✓		✗	367 ms	✓		✗		✗
LoopSub	15	2	60 ms	✓		✗	74 ms	✓		✗		✗
UnchLoop	13	2	69 ms	✓		✗	71 ms	✓		✗		✗
loop	11	3	43 ms	✓		✗	52 ms	✓		✗		✗
while-if	11	3	66 ms	✓	156 ms	✓	66 ms	✓	97 ms	✓		✗
digits10	24	19	312 ms	✓		✗	207 ms	✓	313 ms	✓	47 ms	✓
barthe	13	2	93 ms	✓		✗	69 ms	✓		✗		✗
Example 28	11	2	81 ms	✓		✗	79 ms	✓		✗		✗
iflow	79	0	808 ms	✓		✗	7.259 s	✓		✗		✗
sign	12	2	29 ms	✓		✗	33 ms	✓		✗		✗
sum	14	4	71 ms	✓		✗	162 ms	✓	349 ms	✓		✗

Figure 6.16: Analyses of synthetic benchmarks from the related works.

the evaluation of our patch analysis with the cell-based memory model, in Fig. 5.20 of Sec. 5.4.1. Fig. 6.16 shows the results of analyses with the cell-based and with the bi-cell based memory models. The polyhedra numerical domain is required for successful analyses of most benchmarks with the cell-based memory domain, as these benchmarks are designed to feature expressive relational invariants. Indeed, only two analyses are successful with octagons with the cell-based memory domain, and none with intervals (hence we do not show results with intervals for this memory model on Fig. 6.16). In contrast, the bi-cell based memory domain allows for six successful analyses with octagons, and two with intervals. Performances are similar with either memory domain.

This experimental evaluation shows that our bi-cell based memory domain allows successful analyses of two kinds of patches.

1. Patches of small programs that require linear invariants can be analyzed successfully with an expressive numerical abstraction such as polyhedra. This also the case with our cell-based memory domain;
2. Patches of large programs that mainly require equalities between related fields of data structures in the two versions can be analyzed successfully with less expressive domains. In the case of patches of type definitions, the interval domain is often sufficient.

6.5 Conclusion

In this chapter, we presented an optimized implementation of our patch analysis for C programs, on top of the Mopsa platform. Starting from the implementation of Chapter 5, which reused Mopsa's cell based memory model directly, we proposed an extension of this memory model that allows sharing the representations of cells to represent equal-

ities symbolically. We implemented our new bi-cell based memory model on top of Mopsa, which resulted in an alternate analysis reusing most of that of Chapter 5, as Mopsa allows plugging alternate memory domains in a modular way.

While the analysis of Chapter 5 required expressive numerical abstractions, our optimized memory domain allows successful analyses of multiple patches of real-world C programs using only non relational numerical domains, which improves scalability dramatically. This is in particular the case for patches of C data structures, where we want to verify the portability of the user program against changes of the offsets of scalar fields of C structs.

In Chapter 7, we will extend our memory model further, in order to enable the inference of another portability property: endian portability, *i.e.* portability against the order of bytes in the representation of scalars on the platform.

Chapter 7

Endian portability analysis

Previous chapters have presented patch analyses of NIMP$_2$ and C programs, as well as a related portability analysis: robustness to variations of the memory layout of C programs (offsets of fields in C structures). Along the way, we have defined the elementary components of such analyses. Chapter 4 proposes an algorithm for constructing a double program from a pair of program versions. Chapter 3 defines a concrete collecting semantics for double programs, and tailors an abstract semantics allowing for efficient patch analysis of numerical programs. Chapter 5 lifts this abstraction to the precise analysis of double C programs, leveraging a memory model for low-level C. Chapter 6 optimizes the memory model to enable scalable analyses of some patches of low-level C programs, among which patches changing the memory layout in data structures.

In this chapter, we leverage the results of all previous chapters to define a new portability analysis, coined *endian portability analysis*. Our analysis can infer that a given program, or two syntactically close versions thereof, compute the same outputs when run with the same inputs on platforms with different byte-orders, *a.k.a.* endiannesses. We target low-level C programs that abuse C pointers and unions, hence rely on implementation-specific behaviors undefined in the C standard [91].

7.1 Introduction

There is no consensus on the representation of a multi-byte scalar value in computer memory [43]. Some systems store the least-significant byte at the lowest address, while others do the opposite. The former are called little-endian, the latter big-endian. Such systems include processor architectures, network protocols and data storage formats. For instance, Intel processors are little-endian, while internet protocols and some legacy and embedded processors are big-endian (*e.g.* SPARC and PowerPC). As a consequence, programs relying on assumptions on the encoding of scalar types may exhibit different behaviors when run on platforms with different byte-orders, *a.k.a.* endiannesses. The case occurs typically with low-level C software, such as device drivers or embedded software. Indeed, the C standard [91] leaves the encoding of scalar types partly unspecified. The precise representation of types is standardized in implementation-specific

```
1   read_from_network((uint8_t *)&x, sizeof(x));
2 # if __BYTE_ORDER == __LITTLE_ENDIAN
3     uint8_t *px = (uint8_t *)&x, *py = (uint8_t *)&y;
4     for (int i=0; i<sizeof(x); i++) py[i] = px[sizeof(x)-i-1];
5 # else
6     y = x;
7 # endif
8   assert_sync(y);
```

Figure 7.1: Reading input in network byte-order (Example 40)

Application Binary Interfaces (ABI), such as [11], to ensure the interoperability of compiled programs, libraries, and operating systems. Although it is possible to write fully portable, ABI-neutral C code, the vast majority of C programs rely on assumptions on the ABI of the platform, such as endianness. Therefore, the typical approach used, when porting a low-level C program to a new platform with opposite endianness, is to eliminate most of the byte-order-dependent code, and to wrap the remainder, if any, in conditional inclusion directives. The result is a pair of syntactically close endian-specific variants of the same program. A desirable property, which we call endian portability, is that a program computes the same outputs when run with the same inputs on the little- and big-endian platforms. By extension, we also say that a program is endian portable if two endian-specific variants thereof compute the same outputs when run with the same inputs on their respective platforms. In this chapter, we describe a static analysis which aims at inferring the endian portability of large real-world low-level C programs.

Motivating example

Example 40 (Reading input in network byte-order)**.** Fig. 7.1 shows a snippet of code for reading network input. This snippet was previously introduced as Example 2 in Sec. 1.1.4. We study it in more detail in this chapter. Recall that the order of bytes in the representation of network data is standardized, platform-independent: it follows the big-endian byte-order. Assume variable `x` has some integer type, of arbitrary byte-size. The sequence of bytes read from the network is first stored into `x`. Assume variable `y` has the same type. `x` is then either copied, or byte-swapped into `y`, depending on whether the endianness of the platform matches the (big-endian) network byte-order, or is the opposite endianness. Our analysis is able to infer that this snippet is endian portable, *i.e.* both endian-specific variants compute the same value for `y`, whatever the values of the bytes read from the network. This property is expressed by the assertion at line 8.

Example 40 abuses pointers to bypass the C type system, a common practice in low-level programming known as *type punning*. Alternatively, some implementations rely on bitwise arithmetics. For instance, if `x` and `y` have type `uint32_t`, the little-endian case may be rewritten as `((x & 0xff000000) >> 24) | ((x & 0xff0000) >> 8) | ((x & 0xff00) << 8) | ((x & 0xff) << 24)`. Other implementations rely on compiler built-in

functions, or assembly code, possibly using dedicated processor instructions. Examples can be seen in the Linux implementations of the POSIX `htons` and `htonl` functions, converting values between host and network byte-order. Our analysis is able to analyze all the above C implementations successfully, as well as alternative implementations (with stubs for assembly code). In the rest of the chapter, unless otherwise stated, we will implicitly refer to a version of Example 40 where variables have type `uint16_t` for illustration purposes, although our method works for integers of arbitrary byte-size.

Approach

Low-level C programs exhibit different semantics when run on platforms with different endiannesses. We thus model them as double programs. The little-endian program is the first (or left, or little-endian) version of the double program, while the big-endian program is the second (or right, or big-endian) version. Both versions may share the same source code, or present syntactic differences (if conditional inclusion is used). Recall that syntactic differences between program versions P_1 and P_2 may be distinct from semantic differences. Syntactically different statements may exhibit the same semantics in P_1 and P_2, like in Example 40, while syntactically equal statements may exhibit different semantics, like with the C statement `*((char*)&x)=1` when integer variable `x` is such that `sizeof(x)>1`. Our approach to endian portability is to devise a joint, whole-program static analysis of a double program able to infer equivalences between the input-output relations of its versions. To this aim, we define a memory model able to represent a joint abstraction of their memories. We first parameterize the memory domain for low-level C programs presented in Sec. 5.2.4 with an explicit endianness parameter. Then, we lift it to double programs, and tailor it to infer, and represent symbolically, relevant equalities between little- and big-endian memories. We do this by adapting the construction of the bi-cell based memory model of Chapter 6 to represent equalitites modulo byte-swapping. We additionally rely on a dedicated numerical domain based on symbolic predicates, to infer complementarity relations between individual bytes of program variables, such as those established by bitwise arithmetic operations. We validate our approach by analyzing large industrial low-level embedded C programs designed to be endian portable.

Related work

Several approaches to endian portability are developed in the literature. [146] relies on a source-to-source translation, which is only sound with respect to annotations provided by the programmer, whereas we require no annotations. [31] extends a compiler to generate code that executes with the opposite byte order semantics as the underlying architecture, at the cost of a performance penalty. Annotations are also required for soundness in some cases. [101] relies on dynamic analysis, which can find portability errors, but cannot prove endian portability formally, unlike our method. The Sparse [33] static analysis tool used by Linux kernel developers relies on pervasive type annotations to detect endiannesses issues, but comes with no formal guarantee.

To our knowledge, no prior work uses sound static analysis to infer endian portability. Neither are we aware of prior work leveraging patch analysis techniques to address portability properties. From the perspective of this thesis, patch analysis is viewed as a particular case of portability analysis, where program versions run in the same environments. Nonetheless, our approach leverages prior work. We build on a memory abstract domain [127],[131, Sect. 5.2] developed for run-time error analysis of low-level C programs able to expose endian-dependent behaviors, and on double program semantics developed for patch analysis [65, 66]. Our symbolic predicate domain is based on previous work on predicate domains [130], and symbolic constant propagation [129]. Our domain is also reminiscent of the Slice domain introduced in [37, 36] for another purpose, and implemented differently.

Outline of the chapter

The rest of the chapter is organized as follows. Sec. 7.2 parameterizes the concrete semantics of low-level C programs with an explicit endianness parameter. Sec. 7.3 describes the memory abstraction. Sec. 7.4 describes the numerical abstraction and introduces a novel numerical domain. Sec. 7.5 presents our prototype implementation on top of Mopsa. Sec. 7.6 presents experimental results. Sec. 7.7 concludes.

7.2 Concrete semantics

Syntactically identical programs running on platforms with different endiannesses may exhibit different semantics. Recall Example 3 from Sec. 1.1.4. Assuming a 32-bit *Application Binary Interface* (ABI) with System V alignments, the statement `p[4]=1` assigns different values to field `s.x`, depending on the endianness of the platform. The value is 1 on a little-endian platform, and 2^{24} on a big-endian one (see Fig. 5.5).

In this section, we define a joint semantics for two versions of a C program, running on platforms with different endiannesses. We start by parameterizing the simple program semantics with an explicit endianness parameter. Then, we define a double program semantics able to express jointly the behaviors of the little-endian and the big-endian versions of a C program.

7.2.1 Semantics of simple endian-aware low-level C programs

The semantics of simple low-level C programs is parameterized by an Application Binary Interface (ABI). In Sec. 5.2.3, we formalized a semantics for low-level simple C programs, assuming a 32-bit little-endian System V ABI. In this chapter, we assume program versions P_1 and P_2 run on platforms with opposite endiannesses. As in Sec. 5.3.2, C structs may additionally exhibit different layouts (offset of fields). Besides endianness and layout, the platforms share the same 32-bit ABI.

Let $\mathcal{A} \triangleq \{\mathcal{L}, \mathcal{B}\}$ denote the possible endiannesses (little- and big-endian). The sizes of types, in contrast, are the same for both program versions. As in Sec. 5.3.2, we thus assume a unique function $\mathit{sizeof} \in \mathit{type} \to \mathbb{N}$ given, which provides these sizes (in bytes).

$$\begin{array}{l}\mathbb{E}[\![*_\tau e]\!]_\alpha\mu \triangleq \{ v \mid \langle V, o\rangle \in \mathbb{E}[\![e]\!]_\alpha\mu \wedge 0 \leq o \leq \mathit{sizeof}(V) - \mathit{sizeof}(\tau) \\ \qquad\qquad \wedge v \in \mathit{bdec}_{\tau,\alpha}(\mu\langle V, o\rangle, \ldots, \mu\langle V, o + \mathit{sizeof}(\tau) - 1\rangle) \} \\ \mathbb{S}[\![*_\tau e_1 \leftarrow e_2]\!]_\alpha M \triangleq \\ \quad \bigcup_{\mu\in M} \{ \mu[\forall i < \mathit{sizeof}(\tau) : \langle V, o + i\rangle \mapsto b_i] \mid \langle V, o\rangle \in \mathbb{E}[\![e_1]\!]_\alpha\mu \\ \qquad\qquad \wedge 0 \leq o \leq \mathit{sizeof}(V) - \mathit{sizeof}(\tau) \wedge (b_0, \ldots, b_{\mathit{sizeof}(\tau)-1}) \in \mathit{benc}_{\tau,\alpha}(\mathbb{E}[\![e_2]\!]_\alpha\mu) \}\end{array}$$

Figure 7.2: Concrete endian-aware semantics of memory reads and writes with endianness α.

We also reuse notations from Sec. 5.2.3 for the semantics of simple C programs, namely the sets of:

- pointer values $\mathcal{P}tr = \mathcal{V} \times \mathbb{Z} \cup \{\mathbf{NULL}, \mathbf{invalid}\}$;
- pointers to addressable memory bytes $\mathcal{A}addr \subseteq \mathcal{P}tr$;
- numeric byte values and symbolic pointer bytes $\mathbb{B} = [0, 255] \cup (\mathcal{P}tr \times \mathbb{N})$;
- scalar values $\mathbb{V} = \mathbb{Z} \cup \mathcal{P}tr$.

The definition of the most concrete endian-aware semantics requires a family of representation functions $\mathit{benc}_{\tau,\alpha} \in \mathbb{V} \to \mathcal{P}(\mathbb{B}^*)$, that convert a scalar value of given type $\tau \in \mathit{scalar\text{-}type}$ and endianness $\alpha \in \mathcal{A}$ into a sequence of $\mathit{sizeof}(\tau)$ byte values. We denote as $\mathit{bdec}_{\tau,\alpha} \in \mathbb{B}^* \to \mathcal{P}(\mathbb{V})$ the reverse conversion. These conversion functions are parametrized by the type and the endianness, which define the binary representation of scalars. They reflect the endianness-dependent ordering of bytes in integers. For instance, $\mathit{benc}_{\mathbf{unsigned\ int},\mathcal{L}}(1) = \{ (1, 0, 0, 0) \}$ and $\mathit{bdec}_{\mathbf{unsigned\ short},\mathcal{B}}(0, 1) = \{ 1 \}$ on a 32-bit platform. In contrast, the memory representation $b = (\langle p, 0\rangle, \langle p, 1\rangle, \langle p, 2\rangle, \langle p, 3\rangle)$ of pointers p is kept symbolic: $\mathit{benc}_{\mathbf{ptr},\mathcal{B}}(p) = \mathit{benc}_{\mathbf{ptr},\mathcal{L}}(p) = \{ b \}$ and $\mathit{bdec}_{\mathbf{ptr},\mathcal{B}}(b) = \mathit{bdec}_{\mathbf{ptr},\mathcal{L}}(b) = \{ p \}$. The benc_τ and bdec_τ functions introduced in Sec. 5.2.3 are the little-endian versions $\mathit{benc}_{\tau,\mathcal{L}}$ and $\mathit{bdec}_{\tau,\mathcal{L}}$ of these representation functions. They are presented in full detail in [131, Sec. 5.2]. The big-endian versions $\mathit{benc}_{\tau,\mathcal{B}}$ and $\mathit{bdec}_{\tau,\mathcal{B}}$ are the same, up to the order of bytes in the representation of integers: $\mathit{benc}_{\tau,\mathcal{B}}(v) = \{ \mathit{bswap}(b) \mid b \in \mathit{benc}_{\tau,\mathcal{L}}(v) \}$ and $\mathit{bdec}_{\tau,\mathcal{B}} = \mathit{bdec}_{\tau,\mathcal{L}} \circ \mathit{bswap}$ if $\tau \neq \mathbf{ptr}$, with the byte-swapping function $\mathit{bswap} \in \mathbb{B}^* \to \mathbb{B}^*$ defined as $\mathit{bswap}(b_0, \ldots, b_{n-1}) = (b_{n-1}, \ldots, b_0)$. In contrast, we keep the same symbolic representation of pointers for big-endian: $\mathit{benc}_{\mathbf{ptr},\mathcal{B}} = \mathit{benc}_{\mathbf{ptr},\mathcal{L}}$ and $\mathit{bdec}_{\mathbf{ptr},\mathcal{B}} = \mathit{bdec}_{\mathbf{ptr},\mathcal{L}}$. Reflecting byte-swaps in the machine representation of pointers is indeed irrelevant here, as our memory model does not allow extracting numeric bytes from pointer values.

Remark 34 (Endian-neutrality of single-byte values). The endianness encoding $\alpha \in \mathcal{A}$ only affects the representation of multibyte scalar values. Indeed, $\mathit{benc}_{\tau,\mathcal{B}} = \mathit{benc}_{\tau,\mathcal{L}}$ and $\mathit{bdec}_{\tau,\mathcal{B}} = \mathit{bdec}_{\tau,\mathcal{L}}$ if $\mathit{sizeof}(\tau) = 1$.

Concrete byte-level memory states are elements of $\mathcal{M} \triangleq \mathcal{A}addr \to \mathbb{B}$. The semantics $\mathbb{E}[\![\mathit{expr}]\!] \in \mathcal{A} \to \mathcal{E} \to \mathcal{P}(\mathbb{V})$ and $\mathbb{S}[\![\mathit{stat}]\!] \in \mathcal{A} \to \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ for simple expressions and statements is defined by standard induction on the syntax. We therefore only show, on Fig. 7.2, the semantics $\mathbb{E}[\![*_\tau e]\!]_\alpha$ and $\mathbb{S}[\![*_\tau e_1 \leftarrow e_2]\!]_\alpha$ for memory reads and writes, given endianness $\alpha \in \mathcal{A}$. Bytes are fetched and decoded with $\mathit{bdec}_{\tau,\alpha}$ when reading from

memory in expression $*_\tau e$, while values computed by expression e_2 are encoded into bytes with $benc_{\tau,\alpha}$ when writing to memory in assignment $*_\tau e_1 \leftarrow e_2$.

7.2.2 Endian-aware cell-based memory model

The most concrete endian-aware semantics is not suitable for numerical abstraction, as it would require expressive domains. As in Sec. 5.2.4, we rely instead on a more abstract semantics, based on an extension of the Cells memory model.

Endian-aware memory abstraction

We first extend the definition of cells to take endianness encoding into account. Let $\mathcal{C}ell \subseteq \mathcal{V} \times \mathbb{N} \times \mathit{scalar\text{-}type} \times \mathcal{A}$ denote the finite set of possible endian-aware scalar dereferences. Each cell $\langle V, o, \tau, \varepsilon\rangle \in \mathcal{C}ell$ is denoted as a variable V, an offset o, and information specifying the encoding of values: a scalar type τ and endianness ε:

$$\mathcal{C}ell \triangleq \{\, \langle V, o, \tau, \varepsilon\rangle \mid V \in \mathcal{V},\, \tau \in \mathit{scalar\text{-}type},\, 0 \leq o \leq \mathit{sizeof}(\mathit{typeof}(V)) - \mathit{sizeof}(\tau),\, \varepsilon \in \mathcal{A} \,\}$$

All bytes in a cell are addressable by construction.

Remark 35 (Endian-neutrality of single-byte cells). Following Remark 34, the endianness encoding $\varepsilon \in \mathcal{A}$ of a cell $c = \langle V, o, \tau, \varepsilon\rangle$ is only meaningful if c is a multi-byte cell, *i.e.* $\mathit{sizeof}(\tau) > 1$.

Domain. As in Sec. 5.2.4, we define an abstract memory state as a set of pairs consisting of a set of cells $C \subseteq \mathcal{C}ell$ and a scalar environment over C. The associated memory domain is:

$$\mathcal{E} \triangleq \bigcup_{C \subseteq \mathcal{C}ell} \{\, \langle C, \rho\rangle \mid \rho \in\, C \to \mathbb{V} \,\}$$

A property $X \in \mathcal{P}(\mathcal{E})$ represents the set of endian-aware byte-level memory states $\gamma_{\mathcal{C}ell}(X) \in \mathcal{P}(\mathcal{M})$ satisfying environment constraints over cells. The values of the bytes of these memories must satisfy all of the numerical constraints implied by any memory state:

$$\gamma_{\mathcal{C}ell}(X) \triangleq \left\{ \mu \in \mathcal{M} \;\middle|\; \begin{array}{l} \exists (C,\rho) \in X : \forall \langle V, o, \tau, \varepsilon\rangle \in C : \\ \exists (b_0, \ldots, b_{\mathit{sizeof}(\tau)-1}) \in \mathit{benc}_{\tau,\varepsilon}(\rho\langle V, o, \tau, \varepsilon\rangle) : \\ \forall i < \mathit{sizeof}(\tau) : \mu\langle V, o+i\rangle = b_i \end{array} \right\}$$

As in Sec. 5.2.4, the lattice of properties $(\mathcal{P}(\mathcal{E}), \preccurlyeq, \cup)$ is equipped with partial order

$$X \preccurlyeq X' \iff \forall (C,\rho) \in X : \exists (C',\rho') \in X' : C' \subseteq C \wedge \rho' = \rho_{|C'}$$

$$
\begin{array}{l}
\phi\langle V,o,t,\varepsilon\rangle(C) \triangleq \\
\quad \begin{cases}
\langle V,o,t,\varepsilon\rangle & \textbf{if } \langle V,o,t,\varepsilon\rangle \in C \\
\mathit{wrap}(\langle V,o,t',\varepsilon\rangle, \mathit{range}(t)) & \textbf{else if } \langle V,o,t',\varepsilon\rangle \in C \wedge t,t' \in \mathit{int\text{-}type} \wedge \mathit{sizeof}(t) = \mathit{sizeof}(t') \\
\mathit{byte}(\langle V,o-b,t',\varepsilon'\rangle, w(\varepsilon',b,\mathit{sizeof}(t'))) & \\
\qquad\qquad & \textbf{else if } \langle V,o-b,t',\varepsilon'\rangle \in C \wedge\ t = \mathbf{u8} \wedge t' \in \mathit{int\text{-}type} \wedge b < \mathit{sizeof}(t') \\
\mathit{wrap}(\sum_{i=0}^{\mathit{sizeof}(t)-1} 2^{8\times w(\varepsilon,i,\mathit{sizeof}(t))} \times \langle V,o+i,\mathbf{u8},\varepsilon_i\rangle, \mathit{range}(t)) & \\
\qquad\qquad & \textbf{else if } \forall i < \mathit{sizeof}(t) : \langle V,o+i,\mathbf{u8},\varepsilon_i\rangle \in C \wedge t \in \mathit{int\text{-}type} \\
\mathit{range}(t) & \textbf{else if } t \in \mathit{scalar\text{-}type} \\
\textbf{invalid} & \textbf{else if } t = \mathbf{ptr}
\end{cases} \\
\text{where } w(\varepsilon,i,s) = \begin{cases} i & \textbf{if } \varepsilon = \mathcal{L} \\ s-i-1 & \textbf{if } \varepsilon = \mathcal{B} \end{cases} \\
\text{and } \mathit{byte}(x,w) = \lfloor x/2^{8w} \rfloor \mod 2^8
\end{array}
$$

Figure 7.3: Endian-aware generic cell synthesizing function.

Transfer functions: cell addition and removal. Removing any cell is always sound: it amounts to losing information. It is also possible to add new cells, provided the values assigned to them are consistent with those of existing overlapping cells. As in Sec. 5.2.4, this consistency is ensured by a value synthesize function $\phi \in \mathcal{C}ell \to \mathcal{P}(\mathcal{C}ell) \to \mathit{expr}$ such that $\phi(c)(C)$ returns a syntactic expression denoting (an abstraction of) the value of the cell c as a function of cells in C.

An example implementation is proposed in Fig. 7.3. ϕ is designed as an extension to multiple endianness encodings of the cell synthesize function originally proposed in [131, sec. 5.2], and shown on Fig. 5.8 of Sec. 5.2.4. The two implementations of ϕ are identical, up to the endiannesss encoding $\varepsilon \in \mathcal{A}$ used to compute the weights $w(\varepsilon, i, \mathit{sizeof}(t))$ of bytes at offsets $i < \mathit{sizeof}(t)$ in the value of a cell of type t. Recall from Sec. 5.2.4 that

- function $w \in \{\mathcal{L},\mathcal{B}\} \times \mathbb{N}^2 \to \mathbb{N}$ models the endianness-dependent weights of bytes in integers: $w(\mathcal{L},i,s) = i$ for little-endian encoding, and $w(\mathcal{B},i,s) = s-i-1$ for big-endian encoding;
- $\mathit{byte}(x,w)$ models the value of the byte of weight 2^{8w} in an unsigned integer x.

The original cell synthesize function from Sec. 5.2.4 only considered a little-endian ABI. It thus always used little-endian encoding $\alpha = \mathcal{L}$. In contrast, the endian-aware ϕ shown on Fig. 7.3. uses the endianness encoding ε of a given cell $\langle V,o,t,\varepsilon\rangle$ to compute the weights $w(\varepsilon,i,\mathit{sizeof}(t))$ of bytes at offsets $i < \mathit{sizeof}(t)$ in the value of $\langle V,o,t,\varepsilon\rangle$.

Cell addition, $\mathit{add\text{-}cell} \in \mathcal{C}ell \to \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$, then simply adds the cell and initializes its value using the ϕ function, as in Sec. 5.2.4.

Remark 36 (Alternate sound definitions of ϕ). Multiple definitions can be proposed for ϕ, provided *add-cell* soundly over-approximates the identity. For instance, an integer cell could be synthesized from an existing integer cell of the same size, but opposite endianness encoding, using a byte-swap expression. We use the definition shown on Fig. 7.3 because is the simplest that enables successful analyses of examples of interest.

Transfer functions: expressions and statements. Simple programs have concrete states in $\mathcal{E}$, and their semantics is parameterized by the endianness $\alpha \in \mathcal{A}$ of the platform. Thus $\mathbb{E}[\![\, expr \,]\!] \in \mathcal{A} \to \mathcal{E} \to \mathcal{P}(\mathbb{V})$, and $\mathbb{S}[\![\, stat \,]\!] \in \mathcal{A} \to \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$. Given a platform of fixed endianness $\alpha \in \mathcal{A}$, the transfer functions $\mathbb{E}[\![\, e \,]\!]_\alpha$ and $\mathbb{S}[\![\, s \,]\!]_\alpha$ for simple C programs are defined as in Sec. 5.2.4, up to the fact that all cells are synthesized with the native encoding α of the platform.

Remark 37 (Encoding of cells versus native endianness of the platform). With this setting, all cells of a simple program share the same endianness $\varepsilon = \alpha$, where α is the native endianness of the platform. A natural extension is to support C programs computing with data encoded with multiple byte-orders. Such variables are typically introduced using non-portable C extensions for specifying a fixed, platform-independent storage order. For instance, GCC supports `#pragma scalar_storage_order <e>`, such that `<e>` is `default`, `little-endian` or `big-endian`. This scheme assigns a byte-order $\varepsilon \in \mathcal{V} \to \mathcal{A}$ to every program variable, using the native endianness α of the platform as the default value. In that case, transfer function use the specified endiannesses to synthesize cells $\langle V, o, t, \varepsilon(V) \rangle$.

7.2.3 Semantics of endian-diverse double C programs

We now lift the simple endian-aware program semantics $\mathbb{S}$ to the double program semantics $\mathbb{D}$. Let P be a double C program, and $k \in \{\, 1, 2 \,\}$. As in Sec. 5.3.2, the simple program version $P_k = \pi_k(P)$ has variables in $\mathcal{V}_k$, cells in $\mathcal{C}ell_k$, and concrete states in $\mathcal{E}_k$. As described in Sec. 7.2.2, the semantics of P_k is parameterized by its endianness $\alpha_k \in \mathcal{A}$: $\mathbb{S}_k[\![\, s \,]\!] = \mathbb{S}[\![\, s \,]\!]_{\alpha_k}$. In Chapter 6, we have presented an analysis for one endian-homogenous case, where both program versions run on a little-endian platform: $\alpha_1 = \alpha_2 = \mathcal{L}$. Our implementation also supports the other endian-homogenous case, where both program versions run on a big-endian platform: $\alpha_1 = \alpha_2 = \mathcal{B}$. This analysis could be formalized in the same way as the little-endian case. In this chapter, we are interested in heterogeneous endian-diverse cases, where program versions run on platforms with opposite endiannesses: $\alpha_1 \neq \alpha_2$. We assume, without loss of generality, that P_1 is the little-endian version, and P_2 the big-endian one, *i.e.* $\alpha_1 = \mathcal{L}$ and $\alpha_2 = \mathcal{B}$. We also identify the program version number $k \in \{\, 1, 2 \,\}$ with its endianness $\alpha_k \in \{\, \mathcal{L}, \mathcal{B} \,\}$ to simplify notations. For instance, $P_\mathcal{L} = \pi_\mathcal{L}(P)$ denotes the first, or left, or little-endian version of P, with states in $\mathcal{E}_\mathcal{L}$, and semantics $\mathbb{S}_\mathcal{L}$, while $P_\mathcal{B} = \pi_\mathcal{B}(P)$ denotes the second, or right, or big-endian version of P, with states in $\mathcal{E}_\mathcal{B}$ and semantics $\mathbb{S}_\mathcal{B}$. With these notations, the double program P has concrete states in $\mathcal{D} \triangleq \mathcal{E}_\mathcal{L} \times \mathcal{E}_\mathcal{B}$. $\mathbb{D}[\![\, s \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$ describes the relation between input and output states of s, which are pairs of states of simple programs. The definition for $\mathbb{D}[\![\, s \,]\!]$ is shown on Fig. 7.4. It is the same as that of Fig. 5.12 of Sec. 5.3.2, up to the straightforward replacement of 1 with $\mathcal{L}$ and 2 with $\mathcal{B}$. As in Fig. 5.12 of Sec. 5.3.2, the semantics of **input_sync** and **assert_sync** requires program versions to execute I/Os in lockstep. The semantics $\mathbb{D}[\![\, s_\mathcal{L} \parallel s_\mathcal{B} \,]\!]$ for the composition of two syntactically different statements is thus only valid if neither $s_\mathcal{L}$ nor $s_\mathcal{B}$ reads input or writes output. In this case, $\mathbb{D}[\![\, s_\mathcal{L} \parallel s_\mathcal{B} \,]\!]$

$\mathbb{D}[\![\, dstat \,]\!] \in \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$

$$
\begin{array}{ll}
\mathbb{D}[\![\, \textbf{skip} \,]\!] & \triangleq \lambda X.\, X \\
\mathbb{D}[\![\, s_{\mathcal{L}} \parallel s_{\mathcal{B}} \,]\!] X & \triangleq \bigcup_{(\langle C_{\mathcal{L}}, \rho_{\mathcal{L}} \rangle, \langle C_{\mathcal{B}}, \rho_{\mathcal{B}} \rangle) \in X} (\mathbb{S}_{\mathcal{L}}[\![\, s_{\mathcal{L}} \,]\!]\, \{\, \langle C_{\mathcal{L}}, \rho_{\mathcal{L}} \rangle \,\} \times \mathbb{S}_{\mathcal{B}}[\![\, s_{\mathcal{B}} \,]\!]\, \{\, \langle C_{\mathcal{B}}, \rho_{\mathcal{B}} \rangle \,\}) \\
\mathbb{D}[\![\, l \leftarrow e \,]\!] & \triangleq \mathbb{D}[\![\, l \leftarrow e \parallel l \leftarrow e \,]\!] \\
\mathbb{D}[\![\, \textbf{assert}(c) \,]\!] & \triangleq \mathbb{D}[\![\, \textbf{assert}(c) \parallel \textbf{assert}(c) \,]\!] \\
\mathbb{D}[\![\, l \leftarrow \textbf{input\_sync}(a, b) \,]\!] & \triangleq \dot{\bigcup}_{v \in [a,b]} \mathbb{D}[\![\, l \leftarrow v \,]\!] \\
\mathbb{D}[\![\, \textbf{assert\_sync}(l) \,]\!] X & \triangleq \bigcup_{v \in \mathbb{V}} \{\, (\langle C_{\mathcal{L}}, \rho_{\mathcal{L}} \rangle, \langle C_{\mathcal{B}}, \rho_{\mathcal{B}} \rangle) \in X \mid \mathbb{E}_{\mathcal{L}}[\![\, l \,]\!]\, \langle C_{\mathcal{L}}, \rho_{\mathcal{L}} \rangle = \mathbb{E}_{\mathcal{B}}[\![\, l \,]\!]\, \langle C_{\mathcal{B}}, \rho_{\mathcal{B}} \rangle = \{\, v \,\} \,\} \\
\mathbb{D}[\![\, s \,;\, t \,]\!] & \triangleq \mathbb{D}[\![\, t \,]\!] \circ \mathbb{D}[\![\, s \,]\!] \\
\mathbb{D}[\![\, \textbf{if } e_{\mathcal{L}} \bowtie 0 \parallel e_{\mathcal{B}} \bowtie 0 \textbf{ then } s \textbf{ else } t \,]\!] & \triangleq \mathbb{D}[\![\, s \,]\!] \circ \mathbb{F}[\![\, e_{\mathcal{L}} \bowtie 0 \parallel e_{\mathcal{B}} \bowtie 0 \,]\!] \\
& \dot{\cup}\ \mathbb{D}[\![\, \pi_{\mathcal{L}}(s) \parallel \pi_{\mathcal{B}}(t) \,]\!] \circ \mathbb{F}[\![\, e_{\mathcal{L}} \bowtie 0 \parallel e_{\mathcal{B}} \not\bowtie 0 \,]\!] \\
& \dot{\cup}\ \mathbb{D}[\![\, \pi_{\mathcal{L}}(t) \parallel \pi_{\mathcal{B}}(s) \,]\!] \circ \mathbb{F}[\![\, e_{\mathcal{L}} \not\bowtie 0 \parallel e_{\mathcal{B}} \bowtie 0 \,]\!] \\
& \dot{\cup}\ \mathbb{D}[\![\, t \,]\!] \circ \mathbb{F}[\![\, e_{\mathcal{L}} \not\bowtie 0 \parallel e_{\mathcal{B}} \not\bowtie 0 \,]\!] \\
\mathbb{D}[\![\, \textbf{if } c \textbf{ then } s \textbf{ else } t \,]\!] & \triangleq \mathbb{D}[\![\, \textbf{if } c \parallel c \textbf{ then } s \textbf{ else } t \,]\!] \\
\mathbb{D}[\![\, \textbf{while } e_{\mathcal{L}} \bowtie 0 \parallel e_{\mathcal{B}} \bowtie 0 \textbf{ do } s \,]\!] X & \triangleq \mathbb{F}[\![\, e_{\mathcal{L}} \not\bowtie 0 \parallel e_{\mathcal{B}} \not\bowtie 0 \,]\!] (\text{lfp } H) \\
\mathbb{D}[\![\, \textbf{while } c \textbf{ do } s \,]\!] & \triangleq \mathbb{D}[\![\, \textbf{while } c \parallel c \textbf{ do } s \,]\!]
\end{array}
$$

$$
\text{where } \mathbb{F}[\![\, e_{\mathcal{L}} \bowtie 0 \parallel e_{\mathcal{B}} \bowtie 0 \,]\!] X \triangleq \left\{ (\langle C_{\mathcal{L}}, \rho_{\mathcal{L}} \rangle, \langle C_{\mathcal{B}}, \rho_{\mathcal{B}} \rangle) \in X \;\middle|\; \begin{array}{l} \exists v_{\mathcal{L}} \in \mathbb{E}_{\mathcal{L}}[\![\, e_{\mathcal{L}} \,]\!]\, \langle C_{\mathcal{L}}, \rho_{\mathcal{L}} \rangle : v_{\mathcal{L}} \bowtie 0 \\ \exists v_{\mathcal{B}} \in \mathbb{E}_{\mathcal{B}}[\![\, e_{\mathcal{B}} \,]\!]\, \langle C_{\mathcal{B}}, \rho_{\mathcal{B}} \rangle : v_{\mathcal{B}} \bowtie 0 \end{array} \right\}
$$

$$
\begin{array}{rll}
\text{and} & H(I) & \triangleq X \\
& & \dot{\cup}\ \mathbb{D}[\![\, s \,]\!] \circ \mathbb{F}[\![\, e_{\mathcal{L}} \bowtie 0 \parallel e_{\mathcal{B}} \bowtie 0 \,]\!] I \\
& & \dot{\cup}\ \mathbb{D}[\![\, \pi_{\mathcal{L}}(s) \parallel \textbf{skip} \,]\!] \circ \mathbb{F}[\![\, e_{\mathcal{L}} \bowtie 0 \parallel e_{\mathcal{B}} \not\bowtie 0 \,]\!] I \\
& & \dot{\cup}\ \mathbb{D}[\![\, \textbf{skip} \parallel \pi_{\mathcal{B}}(s) \,]\!] \circ \mathbb{F}[\![\, e_{\mathcal{L}} \not\bowtie 0 \parallel e_{\mathcal{B}} \bowtie 0 \,]\!] I
\end{array}
$$

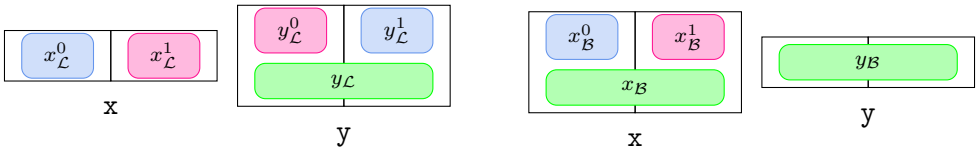
Figure 7.4: Denotational semantics of endian-diverse double C programs.

reverts to the pairing of the simple program semantics of individual simple statements $s_{\mathcal{L}}$ and $s_{\mathcal{B}}$, executed with the endiannesses of their respective platforms. Note that $\mathbb{D}[\![\, s_{\mathcal{L}} \parallel s_{\mathcal{B}} \,]\!] = \mathbb{D}[\![\, \textbf{skip} \parallel s_{\mathcal{B}} \,]\!] \circ \mathbb{D}[\![\, s_{\mathcal{L}} \parallel \textbf{skip} \,]\!]$. The semantics of other statements is the same as in Sec. 5.3.2.

7.2.4 Endian portability property of interest

We wish to prove the functional equivalence between the little-endian (left) and big-endian (right) versions of a given double program $P \in dstat$, restricted to a set of distinguished outputs, specified by **assert_sync**(l) statements.

For instance, consider the program for Example 40, shown in Fig. 7.1. We show in Fig. 7.5 the cells synthesized at the end of the program. Let $x_\alpha \triangleq \langle \mathtt{x}, 0, \textbf{u16}, \alpha \rangle$ denote 2-byte cells for `x`, encoded with the native endianness in Program $\alpha \in \{\, \mathcal{L}, \mathcal{B} \,\}$. 1-byte cells are denoted as $x^o_\alpha \triangleq \langle \mathtt{x}, o, \textbf{u8}, \alpha \rangle$ where $o \in \{\, 0, 1 \,\}$. The cells for `y` are defined in a similar way. Both program versions first call the `read_from_network` function, which reads a stream of bytes from an external source, and writes it into a buffer. The same stream is read by both program versions. A stub for `read_from_network` is shown in Fig. 7.6. After completion of the call, we have $x^0_{\mathcal{L}} = b_0 = x^0_{\mathcal{B}}$ and $x^1_{\mathcal{L}} = b_1 = x^1_{\mathcal{B}}$, where b_0



(a) Program $P_{\mathcal{L}}$ (b) Program $P_{\mathcal{B}}$

Figure 7.5: Memory cells of Example 40: $\square = b_0$, $\square = b_1$, $\square = b_0 \times 2^8 + b_1$.

```
void read_from_network(u8 buf[], u32 size) {
  for (int i=0; i<size; i++) {
    buf[i] = input_sync(0,255);
  }
}
```

Figure 7.6: Stub for the `read_from_network` function.

and b_1 are the values of the first and second bytes read from the network, respectively. Then, Program $P_{\mathcal{L}}$ swaps the bytes of `x` into those of `y`: $x^0_{\mathcal{L}} = y^1_{\mathcal{L}}$ and $x^1_{\mathcal{L}} = y^0_{\mathcal{L}}$. Program $P_{\mathcal{B}}$, in contrast, assigns `x` to `y`. `x` is thus read as a 2-byte cell, while only 1-byte cells are present. Therefore, the endian-aware cell-based memory domain introduced in Sec. 7.2.2 synthesizes $x_{\mathcal{B}}$ by adding the constraint $x_{\mathcal{B}} = 2^8 x^0_{\mathcal{B}} + x^1_{\mathcal{B}}$, following the big-endian byte-order, before performing the assignment $y_{\mathcal{B}} \leftarrow x_{\mathcal{B}}$. To sum up, we obtain the following constraints:

$$x^0_{\mathcal{L}} = x^0_{\mathcal{B}} = y^1_{\mathcal{L}} \qquad x^1_{\mathcal{L}} = x^1_{\mathcal{B}} = y^0_{\mathcal{L}} \qquad y_{\mathcal{B}} = x_{\mathcal{B}} \tag{7.1}$$

In addition to the cell constraints on `x` and `y`:

$$x_{\mathcal{L}} = x^0_{\mathcal{L}} + 2^8 x^1_{\mathcal{L}} \qquad y_{\mathcal{L}} = y^0_{\mathcal{L}} + 2^8 y^1_{\mathcal{L}} \qquad x_{\mathcal{B}} = 2^8 x^0_{\mathcal{B}} + x^1_{\mathcal{B}} \qquad y_{\mathcal{B}} = 2^8 y^0_{\mathcal{B}} + y^1_{\mathcal{B}} \tag{7.2}$$

Formally, the set of reachable double program states defined by the semantics $\mathbb{D}$ is:

$$S = \{\, \langle \langle \{\, x^0_{\mathcal{L}}, x^1_{\mathcal{L}}, y^0_{\mathcal{L}}, y^1_{\mathcal{L}}, y_{\mathcal{L}} \,\}, \rho^{(b_0,b_1)}_{\mathcal{L}} \rangle, \langle \{\, x^0_{\mathcal{B}}, x^1_{\mathcal{B}}, x_{\mathcal{B}}, y_{\mathcal{B}} \,\}, \rho^{(b_0,b_1)}_{\mathcal{B}} \rangle \rangle \mid (b_0, b_1) \in [0, 255]^2 \,\}$$

where:

$$\rho^{(b_0,b_1)}_{\mathcal{L}} : \; x^0_{\mathcal{L}} \mapsto b_0,\; x^1_{\mathcal{L}} \mapsto b_1,\; y^0_{\mathcal{L}} \mapsto b_1,\; y^1_{\mathcal{L}} \mapsto b_0,\; y_{\mathcal{L}} \mapsto 2^8 b_0 + b_1$$
$$\rho^{(b_0,b_1)}_{\mathcal{B}} : \; x^0_{\mathcal{B}} \mapsto b_0,\; x^1_{\mathcal{B}} \mapsto b_1,\; x_{\mathcal{B}} \mapsto 2^8 b_0 + b_1,\; y_{\mathcal{B}} \mapsto 2^8 b_0 + b_1$$

The portability property of interest

$$\mathbb{E}[\![\, y \,]\!]_{\mathcal{L}} \langle C_{\mathcal{L}}, \rho^{(b_0,b_1)}_{\mathcal{L}} \rangle = \mathbb{E}[\![\, y \,]\!]_{\mathcal{B}} \langle C_{\mathcal{B}}, \rho^{(b_0,b_1)}_{\mathcal{B}} \rangle = \{\, 2^8 b_0 + b_1 \,\}$$

can thus be proved for every double program state of S and every pair of input bytes b_0 and b_1.

In practice, the goal of endian portability analysis is to infer $y_{\mathcal{L}} = y_{\mathcal{B}}$ from the constraints 7.1 and 7.2. To do so, we want to leverage numerical domains to abstract the values of cells. However, such relational constraints over cells require an expressive domain, such as polyhedra or linear equalities, that can hamper the scalability of the analysis. In addition, we notice, as in Chapter 7, that we need to infer many equalities, most of which between the little-endian and big-endian versions of the same cells. This is no surprise as we expect most variables to hold equal values in the little- and big-endian memories most of the time, albeit with reversed byte-level representations. Rather than relying completely on the expressiveness of the underlying numerical domain, we optimize our memory model for this common case. To this aim, we reuse the concept of shared bi-cell introduced in Sec. 6.2.2, in order to represent cells equality symbolically.

7.3 Memory abstraction

7.3.1 Domain

As is Chapter 6, we optimize the memory model by sharing the representation of equal cells in the memory of different program versions. We only briefly sketch the formal construction, as it is the same as in Sec. 6.2, up to the extended format of cells (with explicit endianness encoding), and the renaming of P_1 and P_2 into $P_{\mathcal{L}}$ and $P_{\mathcal{B}}$, respectively.

As a first step, as in Sec. 6.2.1, we introduce the set of *single cells*

$$\widetilde{\mathcal{C}ell} \triangleq \mathcal{C}ell_{\mathcal{L}} \uplus \mathcal{C}ell_{\mathcal{B}} = (\mathcal{C}ell_{\mathcal{L}} \times \{\, \mathcal{L} \,\}) \cup (\mathcal{C}ell_{\mathcal{B}} \times \{\, \mathcal{B} \,\})$$

to account for program versions $P_{\mathcal{L}}$ and $P_{\mathcal{B}}$. $\langle V, o, \tau, \varepsilon, \mathcal{L}\rangle \in \widetilde{\mathcal{C}ell}$ denotes a cell $\langle V, o, \tau, \varepsilon\rangle$ in the memory of $P_{\mathcal{L}}$, while $\langle c, \mathcal{B}\rangle \in \widetilde{\mathcal{C}ell}$ denotes a cell $\langle V, o, \tau, \varepsilon\rangle$ in the memory of $P_{\mathcal{B}}$.

Remark 38 (Endianness of a cell versus endianness of a platform). The endian encoding ε of a single cell $\langle V, o, \tau, \varepsilon, \alpha\rangle \in \widetilde{\mathcal{C}ell}$ may be different from the native endianness α of the platform. For instance, $\langle V, 0, \mathbf{u16}, \mathcal{L}, \mathcal{L}\rangle$ denotes a 2-byte cell with native little-endian encoding in the memory of $P_{\mathcal{L}}$, while $\langle V, 0, \mathbf{u16}, \mathcal{B}, \mathcal{L}\rangle$ denotes an overlapping cell encoded in big-endian in the same memory. In a consistent memory model, we should have $\langle V, 0, \mathbf{u16}, \mathcal{L}, \mathcal{L}\rangle \mod 2^8 = \lfloor \langle V, 0, \mathbf{u16}, \mathcal{B}, \mathcal{L}\rangle / 2^8 \rfloor$ and $\lfloor \langle V, 0, \mathbf{u16}, \mathcal{L}, \mathcal{L}\rangle / 2^8 \rfloor = \langle V, 0, \mathbf{u16}, \mathcal{B}, \mathcal{L}\rangle \mod 2^8$. Cells with endianness opposite to that of the platform can be used to represent equalities modulo byte-swapping as simple equalities. In the case of Example 40, shown in Fig. 7.1, the $P_{\mathcal{L}}$ could synthesize a cell with big-endian encoding $x_{\mathcal{L}}^{\mathcal{B}} = \langle \mathtt{x}, 0, \mathbf{u16}, \mathcal{B}, \mathcal{L}\rangle$, with the constraint $x_{\mathcal{L}}^{\mathcal{B}} = x_{\mathcal{B}}$.

As a second step, as in Sec. 6.2.2, we introduce additional cells in the representation of states, in order to represent equalities between cells of $P_{\mathcal{L}}$ and cells of $P_{\mathcal{B}}$. We let $\mathcal{B}icell \triangleq \widetilde{\mathcal{C}ell} \cup (\widetilde{\mathcal{C}ell} \times \widetilde{\mathcal{C}ell})$ denote the set of bi-cells. A bi-cell is either a single cell in $\widetilde{\mathcal{C}ell}$, or a shared bi-cell in $\widetilde{\mathcal{C}ell} \times \widetilde{\mathcal{C}ell}$, *e.g.* a pair of single cells assumed to hold equal value. Bi-cell sharing allows a single representation, in the memory environment, for two cells

from different program versions and holding equal values. For instance, in the context of Example 40, shown in Fig. 7.1, $\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle$ is a shared bi-cell expressing the equality $y_{\mathcal{L}} = y_{\mathcal{B}}$, where $y_{\mathcal{L}}$ and $y_{\mathcal{B}}$ use the native endianness of their respective plateforms. $\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle$ represents the fact that $P_{\mathcal{L}}$ and $P_{\mathcal{B}}$ read the same value for variable `y`, using their respective native endiannesses. In contrast, $\langle x^{\mathcal{B}}_{\mathcal{L}}, x_{\mathcal{B}} \rangle$ is a shared bi-cell expressing the equality $x^{\mathcal{B}}_{\mathcal{L}} = x_{\mathcal{B}}$, where $x^{\mathcal{B}}_{\mathcal{L}} = \langle \mathtt{x}, 0, \mathbf{u16}, \mathcal{B}, \mathcal{L} \rangle$ is a cell of $P_{\mathcal{L}}$ with opposite byte-order, while $x_{\mathcal{B}}$ uses the native endianness of $P_{\mathcal{B}}$. $\langle x^{\mathcal{B}}_{\mathcal{L}}, x_{\mathcal{B}} \rangle$ captures the fact that variable `x` contains a big-endian number in both program versions, which is stems from the fact that `x` is read from the network (as the standard network byte-order is big-endian). In particular, $\langle x^{\mathcal{B}}_{\mathcal{L}}, x_{\mathcal{B}} \rangle$ expresses the fact that the memories for `x` are byte-wise equal in the little- and big-endian versions.

As a third step, as in Sec. 6.2.6, we unify sets of bi-cells. Our memory domain for endian-diverse double programs is thus a choice of a set of bi-cells C and a set of scalar environments on C:

$$\mathcal{D}^\flat \triangleq \bigcup_{C \subseteq \mathcal{B}icell} \{ \langle C, R \rangle \mid R \in \mathcal{P}(C \to \mathbb{V}) \}$$

The domain $\mathcal{D}^\flat$ is equipped with the partial order $\preccurlyeq^\flat$ defined Fig. 5.9 of Sec. 5.2.4 in the context of simple `C` programs.

$$\langle C, R \rangle \preccurlyeq^\flat \langle C', R' \rangle \iff C' \subseteq C \wedge \{ \rho_{|C'} \mid \rho \in R \} \subseteq R'$$

An abstract state represents a set of concrete byte-level memories. The formal construction from Sec. 6.2 ensures that the values of the bytes of these memories satisfy all the numerical constraints on bi-cells implied by the environments:

$$\gamma_{\mathcal{B}icell}\langle C, R \rangle \triangleq \left\{ (\mu_{\mathcal{L}}, \mu_{\mathcal{B}}) \in \mathcal{M}_{\mathcal{L}} \times \mathcal{M}_{\mathcal{B}} \;\middle|\; \begin{array}{l} \exists \rho \in R : \\ \forall c = \langle V, o, \tau, \varepsilon, \alpha \rangle \in \widetilde{\mathcal{C}ell} : \forall c' \in occ(c, C) : \\ \exists (b_0, \ldots, b_{sizeof(\tau)-1}) \in benc_{\tau,\varepsilon}(\rho(c')) : \\ \forall i < sizeof(\tau) : \mu_\alpha\langle V, o+i \rangle = b_i \end{array} \right\}$$

where $occ \in \widetilde{\mathcal{C}ell} \times \mathcal{P}(\mathcal{B}icell) \to \mathcal{P}(\mathcal{B}icell)$ records occurrences of a single cell among bi-cells, as in Sec. 6.2.3:

$$occ(c, C) \triangleq \{ c' \in C \mid c' = c \vee \exists c'' : c' = \langle c, c'' \rangle \vee c' = \langle c'', c \rangle \}$$

In Fig. 7.7, we depict the bi-cells obtained after analyzing the program shown in Example 40. We adapt notations from Fig. 7.5 of Sec. 7.2.4 to the bi-cell based memory domain: $x_\alpha \triangleq \langle \mathtt{x}, 0, \mathbf{u16}, \alpha, \alpha \rangle$ denotes 2-byte cells for `x`, encoded with the native endianness in Program $\alpha \in \{ \mathcal{L}, \mathcal{B} \}$, while $x^o_\alpha \triangleq \langle \mathtt{x}, o, \mathbf{u8}, \alpha, \alpha \rangle$ denotes 1-byte cells at offset $o \in \{ 0, 1 \}$. The cells for `y` are defined in a similar way. For variable `x`, since `read_from_network` writes the same value to $x^0_{\mathcal{L}}$ and $x^0_{\mathcal{B}}$, we can synthesize the shared bi-cell $\langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$ to represent the equality $x^0_{\mathcal{L}} = x^0_{\mathcal{B}}$. In a similar way, we synthesize the shared bi-cell $\langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$. Therefore, as opposed to the separate representation of the memories of Programs $P_{\mathcal{L}}$ and $P_{\mathcal{B}}$ in Fig. 7.5, the joint representation induced by bi-cell sharing

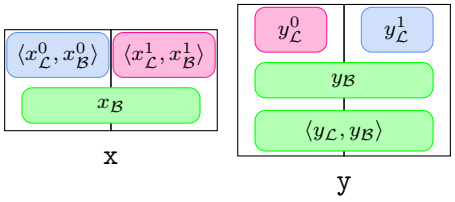


Figure 7.7: Bi-cells of Example 40.

allows reducing the burden on numerical domains, by embedding useful equalities. In the following, we describe more involved cell synthesis operations that allow us to realize $\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle$, and thus to infer that $y_{\mathcal{L}} = y_{\mathcal{B}}$.

7.3.2 Bi-cell synthesis for double programs

Bi-cell synthesis is indeed a cornerstone of our memory domain, as in Sec. 6.2.3. In order to read or write a scalar value to a given location of memory, we must create a suitable bi-cell, or retrieve an existing one from the environment. To guarantee the soundness of the analysis when adding a new bi-cell, it is necessary to ensure that values assigned to it are consistent with those of existing overlapping bi-cells. Our memory domain first attempts to synthesize shared bi-cells if an equality can be inferred from the environment, by pattern-matching. In case of failure, it safely defaults to a pair of single cells, the values of which are set according to those of existing overlapping bi-cells.

We have already used shared bi-cell synthesis implicitly on Fig. 7.7. When reading variable y at the end of Example 40, the memory domain attempts to synthesize $\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle$, as a proof of $y_{\mathcal{L}} = y_{\mathcal{B}}$. To this aim, it searches, among possible patterns, for an existing cell, equal to both $y_{\mathcal{L}}$ and $y_{\mathcal{B}}$. $x_{\mathcal{B}}$ is a candidate, assuming the equality $x_{\mathcal{B}} = y_{\mathcal{B}}$ is recorded in (an abstraction of) the environment. Therefore, the domains looks for a proof that $x_{\mathcal{B}} = y_{\mathcal{L}}$ holds too. To this aim, it looks for 1-byte bi-cells inside $y_{\mathcal{L}}$ and $x_{\mathcal{B}}$, and finds the four blue and red bi-cells $\langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$, $\langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$, $y^0_{\mathcal{L}}$, and $y^1_{\mathcal{L}}$ from Fig. 7.7. As $y_{\mathcal{L}}$ and $x_{\mathcal{B}}$ have opposite endian encodings, it queries the environment for equalities $y^0_{\mathcal{L}} = \langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$ and $y^1_{\mathcal{L}} = \langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$. The success of the synthesis relies on pattern-matching, and three equalities which may be inferred by a numerical domain implementing simple symbolic propagation.

Shared bi-cell synthesis

More generally, given $\alpha \in \mathcal{A}$, let $\mathcal{C}ell^-_\alpha$ denote the set of possible scalar dereferences in the memory of program version P_α:

$$\mathcal{C}ell^-_\alpha \triangleq \{ \langle V, o, \tau \rangle \mid \langle V, o, \tau, \varepsilon \rangle \in \mathcal{C}ell_\alpha \}$$

Consider a pair of scalar dereferences of the same type, coming from different program versions: $c \in \mathcal{C}ell^-_{\mathcal{L}}$ from $P_{\mathcal{L}}$ and $c' \in \mathcal{C}ell^-_{\mathcal{B}}$ from $P_{\mathcal{B}}$. We define the function

$$\phi^\flat(c,c')\langle C,R\rangle \triangleq \begin{cases} \langle\langle c,\mathcal{L},\mathcal{L}\rangle,\langle c',\mathcal{B},\mathcal{B}\rangle\rangle & \textbf{if } \mathit{equal}(\langle c,\mathcal{L},\mathcal{L}\rangle,\langle c',\mathcal{B},\mathcal{B}\rangle)\langle C,R\rangle \\ \langle\langle c,\mathcal{B},\mathcal{L}\rangle,\langle c',\mathcal{B},\mathcal{B}\rangle\rangle & \textbf{else if } \mathit{equal}(\langle c,\mathcal{B},\mathcal{L}\rangle,\langle c',\mathcal{B},\mathcal{B}\rangle)\langle C,R\rangle \\ \langle\langle c,\mathcal{L},\mathcal{L}\rangle,\langle c',\mathcal{L},\mathcal{B}\rangle\rangle & \textbf{else if } \mathit{equal}(\langle c,\mathcal{L},\mathcal{L}\rangle,\langle c',\mathcal{L},\mathcal{B}\rangle)\langle C,R\rangle \\ \top & \textbf{otherwise} \end{cases}$$

Figure 7.8: Shared bi-cell synthesizing function.

$$\begin{aligned} &\mathit{equal}(\langle V,o,\tau,\varepsilon,\alpha\rangle,\langle V',o',\tau,\varepsilon',\alpha'\rangle)\langle C,R\rangle \triangleq \\ &\quad \textbf{let } c=\langle V,o,\tau,\varepsilon,\alpha\rangle \textbf{ and } c'=\langle V',o',\tau,\varepsilon',\alpha'\rangle \textbf{ and } s=\mathit{sizeof}(\tau) \textbf{ in} \\ &\quad \langle c,c'\rangle\in C \vee \langle c',c\rangle\in C \ \vee \\ &\quad (\exists(x,x')\in \mathit{occ}(c,C)\times\mathit{occ}(c',C):\ \forall\rho\in R:\ \rho(x)=\rho(x')) \ \vee \\ &\quad (\forall 0\leq w<s:\ \mathit{equal}(\langle V,o+\mathit{offset}(w,s,\varepsilon),\mathbf{u8},\varepsilon,\alpha\rangle,\langle V',o'+\mathit{offset}(w,s,\varepsilon),\mathbf{u8},\varepsilon',\alpha'\rangle)\langle C,\rho\rangle) \ \vee \\ &\quad (\exists x\in \mathit{flatten}(C)\setminus\{c,c'\}:\ \mathit{equal}(c,x)\langle C,R\rangle\wedge\mathit{equal}(c',x)\langle C,R\rangle) \end{aligned}$$

Figure 7.9: Equality test between single cells.

$\phi^\flat \in \mathcal{C}ell^-_{\mathcal{L}} \times \mathcal{C}ell^-_{\mathcal{B}} \to \mathcal{D}^\flat \to \mathcal{B}icell \cup \{\top\}$ to formalize the patterns matched when attempting to synthesize a shared bi-cell for such a pair of dereferences. An implementation is proposed in Fig. 7.8. Firstly, $\phi^\flat$ returns a shared bi-cell with the endian encodings of the hosts $\langle\langle c,\mathcal{L},\mathcal{L}\rangle,\langle c',\mathcal{B},\mathcal{B}\rangle\rangle$ if $\langle c,\mathcal{L},\mathcal{L}\rangle=\langle c',\mathcal{B},\mathcal{B}\rangle$ may be inferred from the environment. For instance, the shared bi-cell $\langle y_{\mathcal{L}},y_{\mathcal{B}}\rangle$ of Fig. 7.7 is such a host bi-cell, with $c=c'=\langle \mathtt{y},0,\mathbf{u16}\rangle$. Otherwise, $\phi^\flat$ returns a shared bi-cell with big-endian encoding $\langle\langle c,\mathcal{B},\mathcal{L}\rangle,\langle c',\mathcal{B},\mathcal{B}\rangle\rangle$ if $\langle c,\mathcal{B},\mathcal{L}\rangle=\langle c',\mathcal{B},\mathcal{B}\rangle$ holds. For instance, such a shared big-endian bi-cell will be synthesized, with $c=c'=\langle \mathtt{x},0,\mathbf{u16}\rangle$, if variable `x` is read after the end of Example 40. This synthesis is consistent with the fact that `x` is indeed encoded in network byte-order in Example 40. Otherwise, $\phi^\flat$ returns a shared bi-cell with little-endian encoding $\langle\langle c,\mathcal{L},\mathcal{L}\rangle,\langle c',\mathcal{L},\mathcal{B}\rangle\rangle$ if $\langle c,\mathcal{L},\mathcal{L}\rangle=\langle c',\mathcal{L},\mathcal{B}\rangle$ holds. Finally, if all fails, $\phi^\flat$ returns an error $\top$.

$\phi^\flat$ relies on the predicate *equal* to compare two single cells of the same type, with specified endianness encodings. An implementation is shown on Fig. 7.9. *equal* returns *true* when compared cells are part of a shared bi-cell, or when equality is ensured by the environment. Otherwise, *equal* uses the function $\mathit{offset}\in\mathbb{N}\times\mathbb{N}\times\mathcal{A}\to\mathbb{N}$ to compare individual 1-byte bi-cells of the same weights at endianness-dependent offsets. $\mathit{offset}(w,s,\varepsilon)$ models the offset of the byte of weight 2^{8w} in an unsigned integer of size s and endianness ε: $\mathit{offset}(w,s,\mathcal{L})\triangleq w$, and $\mathit{offset}(w,s,\mathcal{B})\triangleq s-w-1$. Otherwise, *equal* searches for candidate single cells in the environment, equal to both c and c'. As in Sec. 6.2.3, we denote the set of single cells in the environment as

$$\mathit{flatten}(C)\triangleq\{\,c\in\widetilde{\mathcal{C}ell}\,|\,c\in C\vee\exists c'\in C:\ \langle c,c'\rangle\in C\vee\langle c',c\rangle\in C\,\}$$

equal returns *true* in case of success, *false* otherwise.

$$
\begin{array}{l}
\mathit{add\text{-}cell}^\flat(c,c')\langle C,R\rangle \triangleq \\
\quad \textbf{if } \phi^\flat(c,c')\langle C,R\rangle = \langle x_{\mathcal{L}}, x_{\mathcal{B}}\rangle \textbf{ then} \\
\quad\quad \langle C \cup \{\,\langle x_{\mathcal{L}}, x_{\mathcal{B}}\rangle\,\}, \{\,\rho[\langle x_{\mathcal{L}}, x_{\mathcal{B}}\rangle \mapsto v] \mid \rho \in R,\, v \in \mathbb{E}_{\mathcal{L}}[\![\, \phi_{\mathcal{L}}(x_{\mathcal{L}})(C) \,]\!]\, \rho\,\}\rangle \\
\quad \textbf{else} \\
\quad\quad \langle C \cup \{c_{\mathcal{L}}, c_{\mathcal{B}}\}, \{\,\rho[\forall \alpha :\, c_\alpha \mapsto v_\alpha] \mid \rho \in R,\, \forall \alpha :\, v_\alpha \in \mathbb{E}_\alpha[\![\, \phi_\alpha(c_\alpha)(C) \,]\!]\, \rho\,\}\rangle
\end{array}
$$

where $c_{\mathcal{L}} = \langle c, \mathcal{L}, \mathcal{L}\rangle$ and $c_{\mathcal{B}} = \langle c', \mathcal{B}, \mathcal{B}\rangle$

Figure 7.10: Bi-cell addition.

Single cell synthesis

If all attempts to synthesize a shared bi-cell for the scalar dereferences c and c' fail, our memory domain synthesizes the pair of single cells $\langle c, \mathcal{L}, \mathcal{L}\rangle$ and $\langle c', \mathcal{B}, \mathcal{B}\rangle$ instead. These bi-cells use the native endianness encoding of their respective host. To set their values soundly, it calls $\phi_{\mathcal{L}}\langle c, \mathcal{L}, \mathcal{L}\rangle(C)$ and $\phi_{\mathcal{B}}\langle c', \mathcal{B}, \mathcal{B}\rangle(C)$, where $\phi_\alpha\langle V, o, \tau, \varepsilon, \alpha\rangle(C)$ returns a syntactic expression denoting (an abstraction of) the value of $\langle V, o, \tau, \varepsilon, \alpha\rangle$ as a function of bi-cells existing in C. For instance, $\phi_{\mathcal{L}}\langle y, 0, \mathbf{u16}, \mathcal{B}, \mathcal{L}\rangle(C) = 2^8 y^0_{\mathcal{L}} + y^1_{\mathcal{L}}$ at the end of Example 40 (see Fig. 7.7). Functions $\phi_\alpha \in \widetilde{\mathcal{C}ell} \to \mathcal{P}(\mathcal{B}icell) \to \mathit{expr}$ are defined as simple extensions of the cell synthesizing function ϕ for endian-aware low-level C programs, which we introduced in Sec. 7.2.2. The functions $\phi_{\mathcal{L}}$ and $\phi_{\mathcal{B}}$ are defined exactly like the functions ϕ_1 and ϕ_2 in Sec. 6.2.3: we project bi-cells of the appropriate side onto cells, apply ϕ, and lift the resulting cell expression back to a bi-cell expression. We do not detail these definitions here, and refer the reader to Sec. 6.2.3 for a precise description.

Bi-cell addition

Bi-cell addition, $\mathit{add\text{-}cell}^\flat \in \mathcal{C}ell^-_{\mathcal{L}} \times \mathcal{C}ell^-_{\mathcal{B}} \to \mathcal{D}^\flat \to \mathcal{D}^\flat$, then simply adds the synthesized bi-cell(s) to the environment, and initializes their value(s), as shown on Fig. 7.10.

Remark 39 (Initializing a shared bi-cell from either single cell). The choice of using $\mathbb{E}_{\mathcal{L}}[\![\, \phi_{\mathcal{L}}(x_{\mathcal{L}})(C) \,]\!]$ (rather than $\mathbb{E}_{\mathcal{B}}[\![\, \phi_{\mathcal{B}}(x_{\mathcal{B}})(C) \,]\!]$) to initialize $\langle x_{\mathcal{L}}, x_{\mathcal{B}}\rangle$ is arbitrary, as they are equal.

7.3.3 Abstract join

As in Sec. 6.2.6, the abstract join must merge environment sets defined on heterogeneous bi-cell sets. We therefore use a unification function $\mathit{unify}^\flat \in (\mathcal{D}^\flat)^2 \to (\mathcal{D}^\flat)^2$. $\mathit{unify}^\flat$ is defined exactly like the unification $\widehat{\mathit{unify}}^\flat$ from Sec. 6.2.6. The join is defined as

$$\langle C_1, R_1\rangle \sqcup^\flat \langle C_2, R_2\rangle \triangleq \langle C'_1 \cup C'_2, R'_1 \cup R'_2\rangle$$

where $(\langle C'_1, R'_1\rangle, \langle C'_2, R'_2\rangle) = \mathit{unify}^\flat(\langle C_1, R_1\rangle, \langle C_2, R_2\rangle)$.

7.3.4 Semantics of simple and double statements

In Sec. 6.2.4 and 6.2.5 of Chapter 7, we introduced the transfer functions for simple and double statements in the context of program versions P_1 and P_2 running on plateforms with the same endianness. The adaptation to platforms with different endianness is straightforward. Besides the replacement of the identifiers $k \in \{ 1, 2 \}$ of program versions by the native endiannesses $\alpha \in \{ \mathcal{L}, \mathcal{B} \}$ of their execution plateforms and the replacement of states $\langle C, \rho \rangle \in X \in \mathcal{P}(\hat{\mathcal{D}})$ by unified states $\langle C, R \rangle \in \mathcal{D}^\flat$, the only significant change is bi-cell synthesis, as Chapters 6 and 7 rely on different bi-cell synthesizing functions. We already presented bi-cell synthesis for double programs in Sec. 7.3.2. In this section, we first describe bi-cell synthesis for for simple programs. Then, we provide a brief overview of the transfer functions of statements, and refer the reader to Appendix B.1 for a complete description.

Bi-cell synthesis for simple programs.

The transfer functions of assignments and tests for the simple program P_α rely on bi-cell synthesis. As for double programs, the memory model attempts to synthesize shared bi-cells whenever possible for dereferences in the memory of P_α, safely defaulting to single cells on side α.

Shared bi-cell synthesis for simple program P_α. More precisely, the function $\phi^\flat_\alpha \in \mathcal{C}ell^-_\alpha \to \mathcal{D}^\flat \to \mathcal{B}icell \cup \{ \top \}$ formalizes the patterns matched when attempting to synthesize a shared bi-cell for a scalar dereference $c_\alpha \in \mathcal{C}ell^-_\alpha$ in the memory of P_α:

$$\phi^\flat_\alpha(c_\alpha) \triangleq \begin{cases} \phi^\flat(c_\mathcal{L}, c_\mathcal{B}) & \textbf{if } \exists \langle c_\mathcal{L}, c_\mathcal{B} \rangle \in \mathfrak{B} \\ \top & \textbf{otherwise} \end{cases}$$

As in Sec. 6.2.4, $\phi^\flat_\alpha$ assumes a relation $\mathfrak{B} \in \mathcal{P}(\mathcal{C}ell^-_\mathcal{L} \times \mathcal{C}ell^-_\mathcal{B})$ given between the possible scalar dereferences in the memory of $P_\mathcal{L}$ and the possible scalar dereferences in the memory of $P_\mathcal{B}$. $\mathfrak{B}$ is a heuristic used as a hint for the memory domain, that we already introduced in Sec. 6.2.4. Its elements are pairs of scalar dereferences that are likely to be equal most of the time during program execution. In our implementation, $\mathfrak{B}$ defines a partial bijection from $\mathcal{C}ell^-_\mathcal{L}$ to $\mathcal{C}ell^-_\mathcal{B}$. It is computed by the analysis front-end, and contains pairs of scalar dereferences likely to be equal most of the time during program execution, namely pairs of scalar dereferences from different program versions that represent the "same" scalar variables or fields in $P_\mathcal{L}$ and $P_\mathcal{B}$. We refer the reader to Sec. 6.2.4 for further details on our implementation of $\mathfrak{B}$.

Single cell synthesis for simple program P_α. If all attempts to synthesize a shared bi-cell for the scalar dereference c fail, our memory domain synthesizes the single single cell $\langle c, \alpha, \alpha \rangle$ instead, using $\phi_\alpha \langle c, \alpha, \alpha \rangle(C)$ to set its value soundly.

$$
\begin{array}{l}
\mathit{add\text{-}cell}^\flat_\alpha(c)\langle C, R\rangle \triangleq \\
\quad \textbf{if } \phi^\flat_\alpha(c)\langle C, R\rangle = \langle x_\mathcal{L}, x_\mathcal{B}\rangle \textbf{ then} \\
\quad\quad \langle C \cup \{\, \langle x_\mathcal{L}, x_\mathcal{B}\rangle \,\}, \{\, \rho[\langle x_\mathcal{L}, x_\mathcal{B}\rangle \mapsto v] \mid \rho \in R,\, v \in \mathbb{E}_\mathcal{L}[\![\, \phi_\mathcal{L}(x_\mathcal{L})(C) \,]\!]\, \rho \,\}\rangle \\
\quad \textbf{else} \\
\quad\quad \langle C \cup \{c_\alpha\}, \{\, \rho[c_\alpha \mapsto v] \mid \rho \in R,\, v \in \mathbb{E}_\alpha[\![\, \phi_\alpha(c_\alpha)(C) \,]\!]\, \rho \,\}\rangle
\end{array}
$$

where $c_\alpha = \langle c, \alpha, \alpha\rangle$

Figure 7.11: Bi-cell addition for simple program P_α.

Bi-cell addition for simple program P_α. Bi-cell addition, $\mathit{add\text{-}cell}^\flat_\alpha \in \mathcal{C}ell^-_\alpha \to \mathcal{D}^\flat \to \mathcal{D}^\flat$, then simply adds the synthesized bi-cell to the environment, and initializes its value, as shown on Fig. 7.11.

Remark 40 (Initializing a shared bi-cell from either single cell). As with Remark 39, the choice of using $\mathbb{E}_\mathcal{L}[\![\, \phi_\mathcal{L}(x_\mathcal{L})(C) \,]\!]$ (rather than $\mathbb{E}_\mathcal{B}[\![\, \phi_\mathcal{B}(x_\mathcal{B})(C) \,]\!]$) to initialize $\langle x_\mathcal{L}, x_\mathcal{B}\rangle$ is arbitrary, as they are equal.

Transfer functions for simple and double statements

Semantics of simple statements. Let $\alpha \in \mathcal{A}$. Simple memory reads and writes in program version P_α enjoy the semantics $\mathbb{E}^\flat_\alpha[\![\, *_t\, e \,]\!] \in \mathcal{D}^\flat \to \mathcal{D}^\flat \times \mathcal{P}(\mathbb{V})$ and $\mathbb{S}^\flat_\alpha[\![\, *_t\, e_1 \leftarrow e_2 \,]\!] \in \mathcal{D}^\flat \to \mathcal{D}^\flat$ in this domain. The transfer functions are defined as in Sec. 6.2.4. To read from memory, the domain enriches the abstract state with the bi-cells corresponding to valid dereferences. Expression evaluation is then performed classically with environment fonctions. To write to memory, the domain materializes the bi-cells for the target location, taking care to split any shared bi-cells, in a copy-on-write strategy. The transfer function for assignments then updates environments for these bi-cells, removing any overlapping bi-cells for soundness.

Semantics of double statements. The semantics $\mathbb{D}^\flat[\![\, \mathit{dstat} \,]\!] \in \mathcal{D}^\flat \to \mathcal{D}^\flat$ for double statements is also defined as in Sec. 6.2.5. $\mathbb{D}^\flat[\![\, l \leftarrow \textbf{input\_sync}(a, b) \,]\!]$ adds a shared bi-cell for l if l is a deterministic left-value expression $*_\tau e$ containing a single dereference. $\mathbb{D}^\flat[\![\, \textbf{assert\_sync}(l) \,]\!]$ tests whether l is a left-value expression $*_\tau e$ that evaluates to a single shared bi-cell. The semantics for tests also uses shared bi-cells to detect conditions that can be symbolically guaranteed to be stable.

In an assignment $\mathbb{D}^\flat[\![\, *_t\, e_1 \leftarrow e_2 \,]\!]\langle C, R\rangle$, although both programs execute the same syntactic assignment, their semantics are different, as are their endiannesses. For instance, recall Example 3 from Sec. 1.1.4. The value assigned to field `s.x` by the statement `p[4]=1` depends on the endianness of the platform. In addition, available bi-cells may be different. By default, double assignments are straightforward extensions of simple assignments: $\mathbb{D}^\flat[\![\, *_t\, e_1 \leftarrow e_2 \,]\!] = \mathbb{S}^\flat_2[\![\, *_t\, e_1 \leftarrow e_2 \,]\!] \circ \mathbb{S}^\flat_1[\![\, *_t\, e_1 \leftarrow e_2 \,]\!]$. As in Sec. 6.2.5, we introduce two precision optimizations, taking advantage of implicit equalities represented by shared bi-cells.

1. If $*_t\, e_1$ and e_2 are deterministic expressions, and if they evaluate to bi-cells that are all shared, then $P_{\mathcal{L}}$ and $P_{\mathcal{B}}$ write the same value to the same destination. The case occurs typically for pervasive statements such as `x=x+1;` if the environment already contains shared bi-cells for `x`. We thus update shared destination bi-cells, and remove any overlapping bi-cells for soundness. This is more efficient than composing simple assignments, as a single (shared bi-cell) variable is updated by a single transfer function for both program versions at the same time. This scheme is also precise, as it preserves shared bi-cells representing equalities. For instance, consider the source code of Example 40 on Fig. 7.1, extended with a final assignment `y++;`. The shared bi-cell $\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle$ is synthesized before this assignment. It is therefore preserved by the transfer function.
2. If, in addition, $e_2 = *_t\, e_2'$, and both $*_t\, e_1$ and $*_t\, e_2'$ evaluate to single bi-cells, then we are dealing with a copy assignment, as in `y=x;`. In that case we copy any bi-cell for the bytes of `x` to a corresponding bi-cell for the bytes of `y`. For instance, consider the source code of Example 40 on Fig. 7.1, extended with a final copy assignment `u16 z=y;`. Before this statement, the synthesized bi-cells for the bytes of variable `y` are $y_{\mathcal{L}}^0$, $y_{\mathcal{L}}^1$, $y_{\mathcal{B}}$ and $\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle$. The transfer function for copy assignments synthesizes the bi-cells $z_{\mathcal{L}}^0$, $z_{\mathcal{L}}^1$, $z_{\mathcal{B}}$ and $\langle z_{\mathcal{L}}, z_{\mathcal{B}} \rangle$ for the bytes of variable `z`. It additionally records equalities in (an abstraction of) the environment:

$$y_{\mathcal{L}}^0 = z_{\mathcal{L}}^0 \qquad y_{\mathcal{L}}^1 = z_{\mathcal{L}}^1 \qquad y_{\mathcal{B}} = z_{\mathcal{B}} \qquad \langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle = \langle z_{\mathcal{L}}, z_{\mathcal{B}} \rangle$$

7.4 Numerical abstraction

We now rely on numerical abstractions to abstract further $\mathbb{D}^\flat$ into a computable abstract semantics $\mathbb{D}^\sharp$, resulting in an effective static analysis.

Connecting to numerical domains. As in Sec. 6.2.7, our memory domain translates memory reads and writes into purely numerical operations on synthetic bi-cells, that are oblivious to the double semantics of double programs: each bi-cell is viewed as an independent numeric variable, and each numeric operation is carried out on a single bi-cell store, as if emanated from a single program. Bi-cells may thus be fed, as variables, to a numerical abstract domain for environment abstraction. Any standard numerical domain, such as polyhedra [51], may be used. Yet, as we aim at scaling to large programs, we restrict ourselves to combinations of efficient non-relational domains, intervals and congruences [82], together with a dedicated symbolic predicate domain.

Introducing a dedicated symbolic predicate domain. Recall Example 40 from Sect. 7.1. Various implementations are possible for the byte-swaps enforcing endian portability of software. Though Example 40 shows an implementation relying on type-punning, implementations relying on bitwise arithmetics are also commonplace. In addition, system-level software, such as Linux device drivers [157], often rely on combinations of type-punning and bitwise arithmetics.

```
1   u16 x; u8 *p = (u8 *)&x;
2   u8 y = input_sync(0,255);
3 # if __BYTE_ORDER == __LITTLE_ENDIAN
4     x = y | 0xff00;
5 # else
6     x = (y << 8) | 0xff;
7 # endif
8   assert_sync(p[0]); assert_sync(p[1]);
```

Figure 7.12: Byte-wise equal memories in different endiannesses (Example 41)

Example 41 (Byte-wise equal memories in different endiannesses)**.** Fig. 7.12 shows a simplified instance of such programming idioms: as `y` has type `unsigned char`, `y|0xff00` and `(y<<8)|0xff` represent the same 16-bit word in different endiannesses. For a successful analysis of this example, the numerical domain must interpret bitwise arithmetic expressions precisely, and infer relations such as: the low-order (respectively high-order) byte of the little-endian (respectively big-endian) version of integer `x` is equal to `y`. Then, the interpretation of dereferences of `p` by the memory domain introduces similar relations between cells, thanks to the bi-cell synthesizing function. In this example, it infers that the little-endian version of the low-address (respectively high-address) byte cell in `x` is equal to the low-order (respectively high-order) byte of `x` – and the converse for big-endian. More precisely, let $x_\alpha \triangleq \langle \mathtt{x}, 0, \mathbf{u16}, \alpha, \alpha \rangle$ denote 2-byte cells for `x`, encoded with the native endianness of P_α, and let $x^o_\alpha \triangleq \langle \mathtt{x}, o, \mathbf{u8}, \alpha, \alpha \rangle$ denote 1-byte cells at offset $o \in \{ 0, 1 \}$. Let $y_\alpha \triangleq \langle \mathtt{y}, 0, \mathbf{u8}, \alpha, \alpha \rangle$ denote 1-byte cells for `y` in the memory of P_α. The analysis of the snippet on Fig. 7.12 synthesizes the bi-cells $\langle y_\mathcal{L}, y_\mathcal{B} \rangle$, $x_\mathcal{L}$, $x_\mathcal{B}$, $x^0_\mathcal{L}$, $x^1_\mathcal{L}$, $x^0_\mathcal{B}$ and $x^1_\mathcal{B}$, with the constraints:

$$x_\mathcal{L} = \langle y_\mathcal{L}, y_\mathcal{B} \rangle + 65280 \qquad\qquad x_\mathcal{B} = 256 \times \langle y_\mathcal{L}, y_\mathcal{B} \rangle + 255 \tag{7.3}$$

in addition to the cell constraints on `x`:

$$x_\mathcal{L} = x^0_\mathcal{L} + 256 \times x^1_\mathcal{L} \qquad\qquad x_\mathcal{B} = 256 \times x^0_\mathcal{B} + x^1_\mathcal{B}$$

The goal of the analysis is to infer the invariants $x^0_\mathcal{L} = x^0_\mathcal{B}$ and $x^1_\mathcal{L} = x^1_\mathcal{B}$. To infer these invariants from such relational constraints over bi-cells with an expressive numerical domain such as linear equalities might hamper the scalability of the analysis. In this section, we introduce a dedicated symbolic domain with near-linear cost that expresses the right invariants for a successful analysis.

7.4.1 The bit-slice symbolic predicate domain

We use a domain based on pattern-matching of expressions to detect arithmetic manipulations of byte values commonly implemented as bitwise arithmetics. It is not sufficient to match each expression independently, as computations are generally spread across

$$\mathcal{B}its ::= \top \mid \mathcal{S}lice$$
$$\mathcal{S}lice ::= n \mid c \mid \overrightarrow{c[i,j)}^{k} \mid (\mathcal{S}lice \,|\, \mathcal{S}lice) \qquad (n \in \mathbb{Z},\ c \in C,\ i,j,k \in \mathbb{N})$$

Figure 7.13: $\mathcal{B}its$, a language of syntactic expressions

sequences of statements. We need, in addition, to maintain some state that retains and propagates information between statements. We maintain this state in a predicate domain $\mathcal{P}red^\sharp \subseteq C \to \mathcal{B}its$, which maps each bi-cell $c \in C \subseteq \mathcal{B}icell$ to a syntactic expression e in a language $\mathcal{B}its$, as a symbolic representation of predicate $c = e$. The precise definition of $\mathcal{P}red^\sharp$ will be stated by Equation 7.4. Fig. 7.13 shows the syntax of $\mathcal{B}its$. $\top$ denotes the absence of information. Otherwise, a syntactic predicate expression may be either a bit-slice, or a bitwise OR of bit-slices. A bit-slice may be an integer constant n, a bi-cell c, or a slice expression $\overrightarrow{c[i,j)}^{k}$ denoting the value obtained by shifting the bits of the bi-cell c between i and $j-1$ to position k: $\overrightarrow{c[i,j)}^{k} \triangleq \lfloor (c \mod 2^j)/2^i \rfloor \times 2^k$. We assume $0 \leq i < j \leq n$, $k \geq 0$ and $k + j - i \leq n$, where $n = 4 \times \mathit{sizeof}(\tau)$ and c has unsigned type τ. Each term of a bitwise OR of bit-slices represents an interval of bits, *e.g.* $[k, k + j - i)$ for a term $\overrightarrow{c[i,j)}^{k}$. We assume that bit-intervals do not overlap: each bit from the result comes from a single cell or constant. For instance, the constraints of Equation 7.3 of Example 41 can be expressed as $\mathcal{B}its$ expressions:

$$x_{\mathcal{L}} = \langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle \mid 65280 \qquad\qquad x_{\mathcal{B}} = 255 \mid \overrightarrow{\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle [0,8)}^{8}$$

They can be rewritten as normal forms that make explicit the interval of bits represented by each term of a bitwise OR of bit-slices:

$$x_{\mathcal{L}} = \overrightarrow{\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle [0,8)}^{0} \mid \overrightarrow{255[0,8)}^{8} \qquad\qquad x_{\mathcal{B}} = \overrightarrow{255[0,8)}^{0} \mid \overrightarrow{\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle [0,8)}^{8}$$

Remark 41 (Bit-slice predicates for signed bi-cells are $\top$). Bi-cells representing scalar dereferences with signed types are mapped to $\top$, which is sound but coarse. This is sufficient for most practical examples, as bitwise arithmetic is mostly used with unsigned integers. This is considered best practice for portability, as the effect of right-shifting negative integers is implementation-defined in the C standard [91]. Most practical implementations use arithmetic shifts with sign extensions. We did not implement sign extension to keep our bit-slice domain simple.

Remark 42 (Bit-slices as bitwise expressions). $\overrightarrow{c[i,j)}^{k}$ can be defined as C bitwise expressions:

$$\overrightarrow{c[i,j)}^{k} = \Big(\big(c \,\%\, 2^j\big) \gg i\Big) \ll k = \Big(\big(c \,\&\, (2^j - 1)\big) \gg i\Big) \ll k$$

The ordering of the domain is flat, based on syntactic predicate equality:

$$X^\sharp \sqsubseteq^\sharp Y^\sharp \iff^{\triangle} \forall c \in C :\ X^\sharp(c) = Y^\sharp(c) \vee Y^\sharp(c) = \top$$

Concretization

An abstract element $X^\sharp \in \mathcal{P}red^\sharp$ denotes the set of environments that satisfy all the predicates in $X^\sharp$, where predicates are evaluated as expressions:

$$\gamma_{\mathcal{P}red}(X^\sharp) \triangleq \{\, \rho \in C \to \mathbb{V} \mid \forall c \in C :\ X^\sharp(c) = \top \vee \rho(c) \in \mathbb{E}[\![\, X^\sharp(c) \,]\!]\rho \,\}$$

Abstract operators

The bit-slice predicate domain can be seen as a specialized instance of the more general symbolic constant domain [129], which maps cells to arbitrary expressions from the source language (instead the selected expressions of interest in $\mathcal{B}its$). We thus give a similar description in this section, following the structure of [129, Sec. 5.1].

Enriched Expressions. The language of C expressions is enriched with the $\mathcal{B}its$ language, where bi-cells play the role of variables, *e.g.* $\mathcal{V} \triangleq C$. Let $\mathcal{C}_{\mathcal{B}its}$ denote the resulting set of syntactic expressions. We use two functions on expression trees:

- $occ \in \mathcal{C}_{\mathcal{B}its} \to \mathcal{P}(\mathcal{V})$ returns the set of variables occurring in an expression;
- $subst \in \mathcal{C}_{\mathcal{B}its} \times \mathcal{V} \times \mathcal{C}_{\mathcal{B}its} \to \mathcal{C}_{\mathcal{B}its}$ substitutes, in its first argument, every occurrence of a given variable with its last argument.

The definition of *occ* and *subst* is standard for non-$\top$ expressions. As in [129, Sec. 5.1], we extend them to $\mathcal{C}_{\mathcal{B}its}$ with:

- $occ(\top) \triangleq \emptyset$;
- $subst(e, V, \top) \triangleq \begin{cases} \top & \textbf{if } V \in occ(e) \\ e & \textbf{otherwise} \end{cases}$.

Abstract Symbolic Environments. We impose a restriction on the domain of symbolic bit-slice predicates: a valid predicate map $P \in \mathcal{P}red^\sharp$ should feature no cyclic dependencies. More precisely,

$$\mathcal{P}red^\sharp \triangleq \left\{ P \in \mathcal{V} \to \mathcal{B}its \;\middle|\; \begin{array}{l} \forall n \in \mathbb{N} :\ \forall\, (V_1, \ldots, V_n) \in \mathcal{V}^n : \\ V_n \in occ(P(V_1)) \wedge \forall i < n :\ V_i \in occ(P(V_{i+1})) \\ \quad \Longrightarrow\ n = 0 \end{array} \right\} \tag{7.4}$$

In particular, $V \notin occ(P(V))$.

This restriction allows defining an additional function $expand \in \mathcal{C}_{\mathcal{B}its} \times \mathcal{P}red^\sharp \to \mathcal{C}_{\mathcal{B}its}$ that substitutes every variables by their (non-$\top$) bindings:

$$expand(e, P) \triangleq subst(\ldots, (subst(subst(e, x_1, P(x_1))), x_2, P(x_2)), \ldots, x_n, P(x_n))$$

where $\{\, x_1, \ldots, x_n \,\} \triangleq \{\, x \in occ(e) \mid P(x) \neq \top \,\}$ is such that $\forall i, j :\ x_j \in occ(P(x_i)) \implies i < j$. It also allows defining the function $expand^\star \in \mathcal{C}_{\mathcal{B}its} \times \mathcal{P}red^\sharp \to \mathcal{C}_{\mathcal{B}its}$, such that $expand^\star(e, P)$ iterates $x \mapsto expand(x, P)$ from e until a fixpoint is reached.

$\underline{\textit{to-bits} \in \mathcal{C}_{\mathcal{B}its} \to \mathcal{B}its}$

$$
\begin{aligned}
\text{Notation: } n &\triangleq 8 \times \textit{sizeof}(\textit{typeof}(e)) \\
\textit{to-bits}(e) &\triangleq e && \textbf{if } e \in \mathbb{N} \cup \mathcal{V} \\
\textit{to-bits}(\overrightarrow{e[i,j)}^{k}) &\triangleq \overrightarrow{\textit{to-bits}(e)[i,j)}^{k} \\
\textit{to-bits}(e \mid e') &\triangleq \textit{to-bits}(e) \mid \textit{to-bits}(e') \\
\textit{to-bits}(\textit{byte}(e,w)) &\triangleq \overrightarrow{\textit{to-bits}(e)[8w,8w+7)}^{0} && \textit{byte}(e,w) = \lfloor e/2^{8w} \rfloor \mod 2^8 \\
\textit{to-bits}(e << q) &\triangleq \overrightarrow{\textit{to-bits}(e)[0,n)}^{q} \\
\textit{to-bits}(e >> q) &\triangleq \overrightarrow{\textit{to-bits}(e)[0,n)}^{-q} \\
\textit{to-bits}((e >> q)\&m) &\triangleq \textit{to-bits}((e \,\&\, (m << q)) >> q) \\
\textit{to-bits}((e << q)\,\&\, m) &\triangleq \textit{to-bits}((e \,\&\, (m >> q)) << q) \\
\textit{to-bits}(e \,\&\, \texttt{0xff}\underbrace{00\ldots00}_{2m}) &\triangleq \overrightarrow{\textit{to-bits}(e)[8m,8(m+1))}^{8m} \\
\textit{to-bits}(e) &\triangleq \top && \text{in all other cases.}
\end{aligned}
$$

(a) Translating expressions to bit-slices

$\underline{[\![\, \textit{stat} \,]\!]^\sharp \in \mathcal{N}um^\sharp \times \mathcal{P}red^\sharp \to \mathcal{P}red^\sharp}$

$$
[\![\, V \leftarrow e \,]\!]^\sharp\langle I, P\rangle \triangleq x \mapsto \begin{cases} nf\langle I, \textit{to-bits}(\textit{subst}(e, V, P(V)))\rangle & \textbf{if } x = V \\ nf\langle I, \textit{to-bits}(\textit{subst}(P(x), V, P(V)))\rangle & \textbf{otherwise} \end{cases}
$$

$$
[\![\, e \bowtie e' \,]\!]^\sharp\langle I, P\rangle \triangleq P
$$

$$
P \sqcup^\sharp P' \triangleq x \mapsto \begin{cases} P(x) \textbf{ if } P(x) = P'(x) \\ \top \textbf{ otherwise} \end{cases}
$$

$$
P \sqcap^\sharp P' \triangleq P
$$

(b) Transfer functions and lattice operators

Figure 7.14: Expression translation, transfer functions and lattice operators.

Abstract lattice operators and transfer functions. The abstract operators are shown on Fig. 7.14(b). Like that of the related symbolic constant domain [129], they are based on symbolic propagation, and implement simple algebraic simplifications. They exhibit similar, near-linear time cost in our experiments.

The transfer function of assignments $V \leftarrow e$ first substitutes V with $P(V)$ in P and e before updating the mapping of V in P. This is necessary to remove all prior information on V, that is no longer valid after the assignment, and to prevent the apparition of dependency cycles. This is also more precise than replacing with $\top$ the mapping of every variable $V \neq W$ such that $V \in \textit{occ}(P(W))$. Then, it translates the transformed e into the $\mathcal{B}its$ language, and maps V to the result of the translation. This translation relies on the function $\textit{to-bits} \in \mathcal{C}_{\mathcal{B}its} \to \mathcal{B}its$, which converts a general expression to a $\mathcal{B}its$

expression, and on function $nf \in \mathcal{N}um^\sharp \times \mathcal{C}_{\mathcal{B}its} \to \mathcal{B}its$, which rewrites a $\mathcal{B}its$ expression to a normal form: either a bit-slice of a variable or a constant, or bitwise OR of disjoint bit-slices, or 0, or $\top$. The definition of *to-bits* is shown on Fig. 7.14(a). *to-bits* translates commonplace bitwise arithmetic expressions. For instance, `x << 16` is translated to $\overrightarrow{x[0,16)}^{16}$ if `sizeof(x)=4`, and `x & 0xff00` is translated to $\overrightarrow{x[8,16)}^{8}$. *to-bits* additionally recognizes some arithmetic expressions from the memory domain. For instance, it translates to $\overrightarrow{\textit{to-bits}(c)[8w,8w+7)}^{0}$ bi-cell expressions $byte(c,w)$ introduced by the bi-cell synthesizing functions $\phi_{\mathcal{L}}$ and $\phi_{\mathcal{B}}$: see Figs. 7.3 and 7.10. *to-bits* safely defaults to $\top$ when failing to recognize a pattern. *nf* rewrites the output of *to-bits* to a normal form. Normal forms ensure that each term of a bitwise OR of bit-slices represents a disjoint interval of bits, sorted by increasing lower bounds. To this aim, *nf* normalizes the representation of constants and variables to bit-slices of themselves. For instance, it relies on the numerical abstraction to derive tight slices for variables x: $nf\langle I, x\rangle \triangleq \overrightarrow{x[0,j)}^{0}$, where $j = \min\{\, 0 \le q < n \mid \mathbb{C}_{\mathcal{N}um}[\![\, x \ge 2^q \,]\!]^\sharp I = \bot \,\}$ and $n = 8 \times sizeof(typeof(x))$. We refer the reader to Fig. B.1 of App. B.2 for the complete definition of *nf*.

As we are primarily interested in propagating assignments, tests are abstracted as the identity, which is sound, though coarse.

Our join only keeps syntactically equal expressions. This corresponds to the least upper bound with respect to $\sqsubseteq^\sharp$. Our meet keeps only the information of the left argument. All these operators respect the non-cyclicity condition.

Remark 43. One could be tempted to refine the meet by mixing information from the left and right arguments in order to minimize the number of variables mapping to $\top$. Unfortunately, careless mixing may break the non-cyclicity condition. As in [129, Sec. 5.1], we settled, as a simpler but safe solution, to keep the left argument.

Finally, we do not need any widening, as the domain is flat.

7.4.2 Integration with the numerical and memory abstractions

As seen in Sec. 7.4.1, our bit-slice symbolic predicate domain cooperates with the numerical abstraction. The cooperation is one-way, in that the symbolic domain relies on variable ranges from the numerical domain to infer tighter slices for variables. The bit-slice domain is additionally queried by our memory domain for equalities, or equalities modulo byte-swapping. We have already introduced queries for equality in Sec. 7.3.2: the $\phi^\flat$ function of Fig. 7.8 relies on the predicate *equal* of Fig. 7.9 to synthesize shared bi-cells. That version of *equal* enables $\phi^\flat$ to synthesize shared bi-cells representing equalities modulo byte-swapping, when byte-swaps are implemented by type-puning techniques. For instance, `x` is byte-swapped into `y` by type-puning in Example 40, and the shared bi-cell $\langle y_{\mathcal{L}}, y_{\mathcal{B}}\rangle$ of Fig. 7.7 is synthesized. Unfortunately, such synthesis fails if some byte operations are performed through bitwise arithmetic, rather than type-puning.

Example 42 (Constructing a big-endian number)**.** More precisely, Fig. 7.16 shows a snippet of code that converts a 2-byte variable `x`, in host byte-order, into a variable `y`, in big-endian byte order. The little-endian version of the program relies on bitwise

$$
\begin{array}{l}
\mathit{equal}(\langle V, o, \tau, \varepsilon, \alpha\rangle, \langle V', o', \tau, \varepsilon', \alpha'\rangle)\langle C, R\rangle \triangleq \\
\quad \textbf{let } c = \langle V, o, \tau, \varepsilon, \alpha\rangle \textbf{ and } c' = \langle V', o', \tau, \varepsilon', \alpha'\rangle \textbf{ and } s = \mathit{sizeof}(\tau) \textbf{ in} \\
\quad \langle c, c'\rangle \in C \vee \langle c', c\rangle \in C \ \vee \\
\quad (\exists (x, x') \in \mathit{occ}(c, C) \times \mathit{occ}(c', C) : \forall \rho \in R : \ \rho(x) = \rho(x')) \ \vee \\
\quad (\forall 0 \leq w < s : \ \mathit{equal}(\langle V, o + \mathit{offset}(w, s, \varepsilon), \mathbf{u8}, \varepsilon, \alpha\rangle, \langle V', o' + \mathit{offset}(w, s, \varepsilon), \mathbf{u8}, \varepsilon', \alpha'\rangle)\langle C, \rho\rangle) \ \vee \\
\quad (\exists x \in \mathit{flatten}(C) \setminus \{\, c, c' \,\} : \ \mathit{equal}(c, x)\langle C, R\rangle \wedge \mathit{equal}(c', x)\langle C, R\rangle) \ \vee \\
\quad \textbf{let } \bar{\varepsilon} \neq \varepsilon \textbf{ and } \bar{c} = \langle V, o, \tau, \bar{\varepsilon}, \alpha\rangle \textbf{ in } \ \mathit{is\text{-}bswap}(\bar{c}, c')\langle C, R\rangle
\end{array}
$$

where

$$
\mathit{is\text{-}bswap}(c, c')\langle C, R\rangle \triangleq \exists (x, x') \in \mathit{occ}(c, C) \times \mathit{occ}(c', C) : \ \mathit{bswap\text{-}query}(x, x', R)
$$

$$
\begin{array}{l}
\mathit{bswap\text{-}query}(c, c', R) \triangleq \\
\quad \textbf{let } c = \langle V, o, \tau, \varepsilon, \alpha\rangle \textbf{ and } c' = \langle V', o', \tau, \varepsilon', \alpha'\rangle \textbf{ and } n = \mathit{sizeof}(\tau) \textbf{ in} \\
\quad \forall \rho \in R : \ \rho(c) = \mathit{bswap}(\rho(c'), n)
\end{array}
$$

Figure 7.15: Extension of the *equal* predicate of the memory abstraction to equalities modulo bitwise arithmetic byte-swapping.

```
1   u16 x = input_sync(0,65535);
2   u16 y;
3 # if __BYTE_ORDER == __LITTLE_ENDIAN
4     y = ((x >> 8) & 0xff) | ((x & 0xff) << 8);
5 # else
6     y = x;
7 # endif
8   y;
```

Figure 7.16: Constructing a big-endian number (Example 42)

arithmetic to byte-swap `x` into `y`, while the big-endian version performs an assignment. Then, both program versions read the value of `y`. Using the *equal* predicate from Fig. 7.9, the $\phi^\flat$ function of our memory domain fails to synthesize any shared bi-cell for the dereference $c = \langle \mathtt{y}, 0, \mathbf{u16}\rangle$. Nonetheless, the shared bi-cells $c^{\mathcal{B}} = \langle\langle c, \mathcal{B}, \mathcal{L}\rangle, \langle c, \mathcal{B}, \mathcal{B}\rangle\rangle$ and $c^{\mathcal{L}} = \langle\langle c, \mathcal{L}, \mathcal{L}\rangle, \langle c, \mathcal{L}, \mathcal{B}\rangle\rangle$ would both be sound. For instance, the memory domain should infer the invariant $\langle c, \mathcal{L}, \mathcal{L}\rangle = \mathit{bswap}(\langle c, \mathcal{B}, \mathcal{B}\rangle, \mathit{sizeof}(\mathbf{u16}))$ to synthesize $c^{\mathcal{B}}$, where the function $\mathit{bswap}(x, n)$ byte-swaps the unsigned machine integer x of byte-size n:

$$
\mathit{bswap}(x, n) = \overrightarrow{x[8(n-1), 8n)}^{0} \mid \ldots \mid \overrightarrow{x[8(n-k-1), 8(n-k))}^{8k} \mid \ldots \mid \overrightarrow{x[0, 8)}^{8(n-1)}
$$

To this aim, we extend the *equal* predicate of Fig. 7.9 to query for such equalities modulo bitwise byte-swapping. The new version of *equal* is shown on Fig. 7.15. It appends a case at the end of the disjunction that tests single cells c and c' for equality.

This case tests whether $\bar{c}$ and c' are arithmetic byte-swaps of each other, where $\bar{c}$ is a single cell representing the same bytes as c, albeit with opposite endianness encoding.

This test relies on the *bswap-query* predicate. In the numerical abstraction, this predicate is supported by a query from the memory domain to the bit-slice domain. Given an abstract state $\langle I, P\rangle$, the bit-slice domain first expands recursively the mappings of $\bar{c}$ and c', and reduces the resulting $\mathcal{B}its$ expressions to normal forms:

$$\begin{cases} \bar{s} = nf\langle I, expand^\star(\bar{c}, P)\rangle \\ s' = nf\langle I, expand^\star(c', P)\rangle \end{cases}$$

Then it tests whether $\bar{s}$ and s' have the form

$$\begin{cases} \bar{s} = \overrightarrow{x_{n-1}[8(n-1), 8n)}^{0} \mid \ldots \mid \overrightarrow{x_{n-k-1}[8(n-k-1), 8(n-k))}^{8k} \mid \ldots \mid \overrightarrow{x_0[0, 8)}^{8(n-1)} \\ s' = \overrightarrow{x_0[0, 8)}^{0} \mid \ldots \mid \overrightarrow{x_k[8k, 8(k+1))}^{8k} \mid \ldots \mid \overrightarrow{x_{n-1}[8(n-1), 8n)}^{8(n-1)} \end{cases}$$

where $(x_0, \ldots, x_{n-1})$ are single cells or constants. In this case, the answer from the bit-slice domain to the *bswap-query* query is *true*. Otherwise it is *false*.

In the case of Example 42, this version of *equal* enables the synthesis of the shared big-endian bi-cell $c^{\mathcal{B}}$ by the memory domain, and the bit-slice predicate domain records the equality: $c^{\mathcal{B}} = \langle\langle \mathtt{x}, 0, \mathbf{u16}, \mathcal{L}, \mathcal{L}\rangle, \langle \mathtt{x}, 0, \mathbf{u16}, \mathcal{B}, \mathcal{B}\rangle\rangle$

Remark 44 (Queries for variable equality). The bit-slice domain handles equality queries in a similar way. The third case of the disjunction of the *equal* predicate, $\forall \rho \in R : \rho(x) = \rho(x')$, is indeed supported in the numerical abstraction by another dedicated query from the memory domain to the bit-slice domain, to test the equality of x and x'. Given an abstract state $\langle I, P\rangle$, the bit-slice domain interprets this test as

$$nf\langle I, expand^\star(V, P)\rangle = nf\langle I, expand^\star(V', P)\rangle \neq \top$$

i.e. x and x' are equal if they can be expanded to identical non-$\top$ normal $\mathcal{B}its$ expressions.

7.4.3 Analysis of Example 41

After this integration of our bit-slice predicate domain with our bi-cell based memory abstraction, we are now ready to analyze Example 41, using any underlying non-relational numerical domain, such as intervals.

Four cells are synthesized by the memory domain before line 8:

$$C_8 = \{\, x_{\mathcal{L}}, x_{\mathcal{B}}, \langle y_{\mathcal{L}}, y_{\mathcal{B}}\rangle, \langle p_{\mathcal{L}}, p_{\mathcal{B}}\rangle \,\}$$

$x_{\mathcal{L}} = \langle \mathtt{x}, 0, \mathbf{u16}, \mathcal{L}, \mathcal{L}\rangle$ is a little-endian single cell for the bytes of variable `x`, and $x_{\mathcal{B}} = \langle \mathtt{x}, 0, \mathbf{u16}, \mathcal{B}, \mathcal{B}\rangle$ is a big-endian one. $\langle y_{\mathcal{L}}, y_{\mathcal{B}}\rangle = \langle\langle \mathtt{y}, 0, \mathbf{u8}, \mathcal{L}, \mathcal{L}\rangle, \langle \mathtt{y}, 0, \mathbf{u8}, \mathcal{B}, \mathcal{B}\rangle\rangle$ and $\langle p_{\mathcal{L}}, p_{\mathcal{B}}\rangle = \langle\langle \mathtt{p}, 0, \mathbf{ptr}, \mathcal{L}, \mathcal{L}\rangle, \langle \mathtt{p}, 0, \mathbf{ptr}, \mathcal{B}, \mathcal{B}\rangle\rangle$ are shared bi-cells for variable `y` and pointer `p`.

– $\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle$ is created at line 2, and represents the fact that variable `y` has the same value in the little- and big-endian versions.
– The transfer function for assignment of the symbolic predicate domain infers invariants $x_{\mathcal{L}} = \langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle \,|\, 65280$ from line 4, and $x_{\mathcal{B}} = 255 \,|\, \overrightarrow{\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle [0,8)}^{8}$ from line 6.
– Then, the dereferences of pointer `p` at line 8 are interpreted by the memory domain. Four more cells $\{\, x^o_\alpha \,|\, (\alpha, o) \in \{\, \mathcal{L}, \mathcal{B} \,\} \times \{\, 0, 1 \,\} \,\}$ are added to the abstract environment, to denote the bytes of variable `x` in the little- and big-endian programs. More precisely, $x^o_{\mathcal{L}} = \langle x, o, \mathbf{u8}, \mathcal{L}, \mathcal{L} \rangle$, and $x^o_{\mathcal{B}} = \langle x, o, \mathbf{u8}, \mathcal{B}, \mathcal{B} \rangle$, at offsets $o \in \{\, 0, 1 \,\}$. Following the definition of *add-cell*$^\flat$ on Fig. 7.10, these new bi-cells are initialized with values computed from the outputs of the bi-cell synthesizing functions $\phi_{\mathcal{L}}$ and $\phi_{\mathcal{B}}$. In practice, these initializations are $x^o_{\mathcal{L}} = \mathit{byte}(x_{\mathcal{L}}, o)$, and $x^o_{\mathcal{B}} = \mathit{byte}(x_{\mathcal{L}}, 1 - o)$, for offsets $o \in \{\, 0, 1 \,\}$, with $\mathit{byte}(n, k) = \lfloor n / 2^{8k} \rfloor \mod 2^8$. These assignments are then interpreted by the numerical domain, and by the symbolic predicate domain in particular, as $x^o_{\mathcal{L}} = \mathit{byte}'(x_{\mathcal{L}}, o)$, and $x^o_{\mathcal{B}} = \mathit{byte}'(x_{\mathcal{B}}, 1 - o)$, with $\mathit{byte}'(n, k) = \overrightarrow{n[8k, 8k+8)}^{0}$.
– Finally, the memory domain attempts to synthesize the shared bi-cells $\langle x^0_{\mathcal{L}}, x^0_{\mathcal{B}} \rangle$ and $\langle x^1_{\mathcal{L}}, x^1_{\mathcal{B}} \rangle$ as a proof of the assertions line 8. To this aim, it queries the bit-slice domain for the equalities $x^0_{\mathcal{L}} = x^0_{\mathcal{B}}$ and $x^1_{\mathcal{L}} = x^1_{\mathcal{B}}$. As shown in Remark 44, the bit-slice domain replaces bi-cells with the symbolic expressions bound to them, if any. The tests are thus $\overrightarrow{(\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle \,|\, 65280)\,[0,8)}^{0} = \overrightarrow{\left(255 \,|\, \overrightarrow{\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle [0,8)}^{8}\right)[8,16)}^{0}$ and $\overrightarrow{(\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle \,|\, 65280)\,[8,16)}^{0} = \overrightarrow{\left(255 \,|\, \overrightarrow{\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle [0,8)}^{8}\right)[0,8)}^{0}$. Both tests evaluate to true, using symbolic simplifications (and integer arithmetic computations) supported by the normalization of bit-slice expressions (see App. B.2). Indeed the first test is normalized to $\overrightarrow{\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle [0,8)}^{0} = \overrightarrow{\langle y_{\mathcal{L}}, y_{\mathcal{B}} \rangle [0,8)}^{0}$, while the second test is normalized to $\overrightarrow{255[0,8)}^{0} = \overrightarrow{255[0,8)}^{0}$.

Hence, the assertions line 8 are proved correct: at the end of the program, the memories for variable `x` are byte-wise equal in the little and big-endian versions.

7.5 Implementation

We implemented a static analysis based on the $\mathbb{D}^\sharp$ abstract semantics on top of Mopsa. The main differences, with respect to the $\hat{\mathbb{D}}^\sharp$ semantics presented in Sec. 6.3, are:
– an extended representation of the platforms of the two program versions, to record their native endiannesses;
– the extended representation of bi-cells, which features their endianness encoding;
– the extended bi-cell synthesizing function $\phi^\flat$;
– the integration of our symbolic domain of bit-slice predicates.

This additional domain is reflected in a modular way in the configuration of the analysis shown on Fig. 6.14. The only change, with respect to Fig. 6.14 from Sec. 6.3, is the abstraction of scalar values. The domain C.machineNum is replaced by the prod-

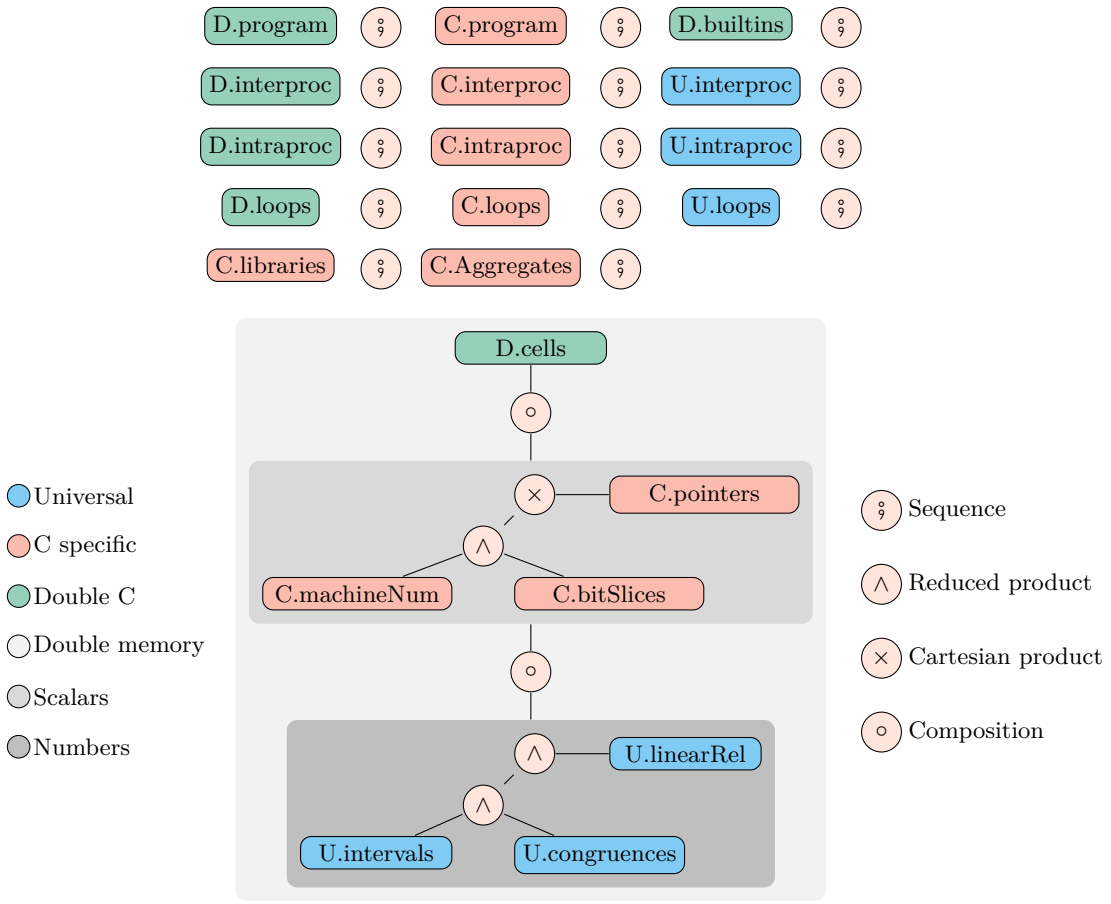


Figure 7.17: Analysis of double endian-diverse C programs with bi-cells and bit-slice predicates

uct C.machineNum ∧ C.bitSlices. C.machineNum translates machine integer arithmetic into mathematical arithmetic on unbounded integers, while C.bitSlices implements our symbolic domain of bit-slice predicates directly at the level of C (not mathematical) expressions. Both domains abstract the machine integer values of scalar bi-cells synthesized by the memory domain. As opposed to purely numerical domains such as U.intervals, both C.machineNum and C.bitSlices rely on the ranges and bit-sizes of machine integer expressions. The C.bitSlices domain accounts for 1000 lines of OCaml in our implementation.

7.6 Evaluation

We have experimented our prototype on small idiomatic examples, open source software, and large industrial software. The analyses were run on a 3.4 GHz Intel® Xeon® CPU,

using a single core as the analysis is not multi-threaded.

7.6.1 Idiomatic examples

We first check the precision and robustness of our analysis against a collection of small double C programs (between 20 and 100 LOC), inspired by various implementations of byte-swaps in Linux drivers, POSIX `htonl` functions, and industrial software.

A set of 9 programs illustrate network data processing. These programs are similar to Example 40 of Sect. 7.1. They receive an integer from the network, increment it, and send over the result. Necessary byte-swaps are implemented for little-endian versions of these programs. Each example program implements a different byte-swapping technique on a 2, 4, or 8-byte integer: type-punning with pointer casts (like in Example 40), unions, or bitwise arithmetics. Refer to Examples 44 and 45 of App. C.2.1 and Example 46 of App. C.2.2 for the source codes, and to Fig. C.1(a) of App. C.2 for analysis times. We also analyze Example 41 from Fig. 7.12 to demonstrate the efficiency of our symbolic predicate domain.

Our prototype also handles floating-point data, which was omitted in the formalization for the sake of conciseness. We developed small floating-point examples representative of industrial use-cases of Sect. 7.6.3. They include byte-swappings of simple or double precision floating-point numbers sent to or received from the network, on architectures where integers and floats are guaranteed to have the same byte-order. Type-punning is used to reinterpret floats as integers of the same size, which are byte-swapped using bitwise arithmetics. Also, a combination of type-punning and byte-swapping is used to extract exponents from double precision floats. The source codes of these Examples 48 and 49 are available in App. C.2.3.

All analyses run in less than 200 ms and report no false alarm, thus proving the functional equivalence of the little-endian and big-endian versions of each program with respect to the values of outputs. Analysis times are shown on Fig. C.1(b) of App. C.2. These results can be reproduced by analyzing the benchmarks with reference numbers 4, 5, 6, 8 and 9 in the companion artifact [68] to our related paper [67].

7.6.2 Open source benchmarks

We then check the soundness, precision, and modularity of our analysis on three benchmarks based on open source software available on GitHub, with multiple commits for bug-fixes related to endianness portability. We analyze slices between 100 and 250 LOC, using primitives **input_sync** and **assert_sync** for modular specifications of program parts. Refer to Examples 50, 51, and 52 in App. C.2.4 for relevant source codes excerpts.

Our first benchmark is an implementation of a tunneling driver [157] based on the Geneve [83] encapsulation network protocol, which uses big-endian integers as tunnel identifiers. The driver was introduced in 2014[1] in the Linux kernel, and patched several

[1] `https://github.com/torvalds/linux/commit/0b5e8b8eeae40bae6ad7c7e91c97c3c0d0e57882`

times[2] for endianness-related issues detected by SPARSE [33]. Then, a performance optimization introduced in 2016[3] a new endianness portability bug, which SPARSE failed to detect. It was fixed a year later[4]. Our analysis soundly reports this bug, as well as previous issues detected by SPARSE. It reports no alarm on the fixed code.

Our second benchmark is a core library of the mlx5 Linux driver [125] for ethernet and RDMA net devices [120]. We analyze a slice related to a patch[5], committed to fix an endianness bug introduced 3 years earlier[6], and undetected by SPARSE despite the use of relevant annotations. The fix turned out to be incomplete, and was updated 6 months later[7]. Our analysis soundly reports bugs on the two first versions, and no alarm on the third.

Our third benchmark is extracted from a version of Squashfs [171], a compressed read-only filesystem for Linux, included in the LineageOS [170] alternative Android distribution. We analyze a slice related to a patch[8], committed to fix an endianness bug undetected by SPARSE due to a lack of type annotations. Our analysis soundly reports the bug, and no alarm on the fixed version.

All the analyses run within 1 second. Analysis times are shown on Fig. C.1(b) of App. C.2. These results can be reproduced by analyzing the benchmarks with reference numbers 10, 11, and 12 in the companion artifact [68] to our related paper [67].

7.6.3 Industrial case study

We analyzed two components of a prototype avionics application, developed at Airbus for a civil aircraft. This application is written in C, and primarily targets an embedded big-endian processor. Nonetheless, it must be portable to little-endian commodity hardware, as its source code is reused as part of a simulator used for functional verification of SCADE [24] models. The supplement to the applicable aeronautical standard [4] related to model-based development [5] mandates, in this case, that "*an analysis should provide compelling evidence that the simulation approach provides equivalent defect detection and removal as testing of the Executable Object Code*". Airbus, known to rely on formal methods for other verification objectives [61, 166, 64, 133, 28], is currently considering the use of static analysis to verify this portability property.

Endianness is the main difference between the ABIs of the embedded computer and the simulator. We thus experimented our prototype analyzer on the modules of the application integrated to the simulator, to which we refer as A and S. For both modules, the analysis aims at proving the functional equivalence of the little-endian and the big-endian versions with respect to the values of outputs. Modules A and S are data-

[2] https://github.com/torvalds/linux/commit/42350dcaaf1d8d95d58e8b43aee006d62c84bc2e
https://github.com/torvalds/linux/commit/0a5d1c55faa5414858857875496f6f6a9926fa51

[3] https://github.com/torvalds/linux/commit/2e0b26e1035253bda7587f705f346385352e942d

[4] https://github.com/torvalds/linux/commit/772e97b57a4aa00170ad505a40ffad31d987ce1d

[5] https://github.com/torvalds/linux/commit/404402abd5f90aa90a134eb9604b1750c1941529

[6] https://github.com/torvalds/linux/commit/2b64beba025109f64e688ae675985bbf72196b8c

[7] https://github.com/torvalds/linux/commit/82198d8bcdeff01d19215d712aa55031e21bccbc

[8] https://github.com/LineageOS/android_kernel_sony_msm8960t/commit/5f61dc71accfea6c946
7499d0e3eb5462dab8d63

intensive reactive software, processing thousands of global variables, with very flat call graphs. Module A is in charge of acquiring and emitting data through aircraft buses. It is composed of about 1 million LOC, most of which generated automatically from a description of the avionics network. It handles integers, Booleans, single and double precision floats, and outputs about 3,500 individual scalar data. The code features bounded loops, memcpys, pointer arithmetics, and type-punning with unions and pointer casts. It also uses bitwise arithmetics, among which several thousand byte-swaps related to endianness portability. Module S is in charge of the main applicative functions. It is composed of about 300,000 LOC, most of which generated automatically from SCADE models. It handles mostly Booleans and double precision floats, and outputs about 200 individual scalar data. It features bounded loops and bitwise arithmetics, but no type-punning. The target application is required to meet its specifications for long missions. We therefore use unbounded loops to emulate the main reactive loop of the application in the analysis drivers of modules A and S.

Both analyses run in 5 abstract iterations. The analysis of A runs in 20.4 hours and uses 5.5 GB RAM. The analysis of S runs in 9.7 hours and uses 2.7 GB RAM. We worked with the development and simulation teams to analyze early prototypes, and incorporate findings into the development cycle. Therefore analyses report zero alarm on current versions of modules A and S.

7.7 Conclusion

In this chapter, we presented a sound static analysis of endian portability for low-level C programs. Our method is based on abstract interpretation, and parametric in the choice of a numerical abstract domain. To this aim, we first parameterized the cell-based semantics of low-level C programs introduced in Chapter 5 with an explicit endianness parameter. Then, we reused the bi-cell based memory model introduced in Chapter 6 to lift this semantics to double programs. We tailored this memory model to support the synthesis of shared bi-cells representing either equalities, or equalities modulo byte-swapping. Finally, we introduced a novel symbolic predicate domain with near-linear cost, that is able to infer relations between individual bytes of the variables in the two programs, such as those established by bitwise arithmetic operations. We implemented a prototype static analyzer, able to scale to large real-world industrial software, with zero false alarms.

For future work, we are starting a project to industrialize this prototype, towards an industrial deployment of endian portability analysis as a means to address avionics certification objectives related to simulation fidelity. We are also considering extending our analysis to further ABI-related properties, such as portability between different sizes of machine integers. We anticipate that our bi-cell sharing approach will benefit to such analyses.

Chapter 8

Conclusion

In this thesis, we have contributed to the design of methods for the analysis of software patches, and the analysis of endian and structure layout portability, in the context of low-level C programs manipulating memory at the byte-level. We have successfully experimented our analyses on real-world open source and industrial software. In particular, our prototype static analyzer for endian portability analyzes successfully large avionics programs up to one million lines of C in less than a day of computation.

Contributions

Our main contributions are a semantics for double programs, a memory abstraction for double C programs, numerical domains for patch and endian portability analysis, and a prototype static analyzer for patch and portability analysis of C programs.

Double program semantics. Our first contribution is a novel concrete collecting semantics, expressing the behaviors of two versions of a program at the same time. It is defined by induction on the syntax of a joint syntactic representation of two program versions, coined *double program*. Our semantics allows for modular, joint analyses of double programs that maintain input-output relations on the variables. It allows proving the functional equivalence of two program versions running in the same environment and reading from the same input stream, which is the goal of regression verification. Patch analysis is subject to a trade-off between the computational resources invested in the construction of a suitable double program and the expressive power of the abstract domain used to infer the necessary invariants. We propose a heuristic algorithm for constructing a double program from a pair of program versions that allows, in most practical cases, successful patch analyses relying on linear invariants only.

Bi-cell memory domain. Our second contribution is a memory domain for double low-level C programs. This domain allows representing symbolically some relations between the bytes of the two memories. This improves the scalability of patch analysis, as

it enables successful analyses of some real-world patches of C programs using only non-relational numerical abstractions. This domain additionally enables scalable portability analyses. Indeed, it can be tailored to support pairs of programs running in environments with different representations of the computer memory, due to different ABIs. Our implementation supports differences in the memory layout (offsets of scalar fields in C structs) and in the order of bytes in the representation of scalars (endianness). It allows successful analyses of large, real-world avionics software.

Numerical domains. Our third contribution is a pair of numerical domains with near-linear cost to support patch and portability analysis. The Delta domain relies on an underlying numerical abstraction and symbolic simplifications to bound differences between the values of the variables in the two program versions. The bit-slice domain infers relations between individual bytes of the variables in the two programs, such as those established by bitwise arithmetic operations. It is able to recognize patterns commonly used to make C programs endian-portable.

Implementation and experimentation. Our last contribution is a prototype static analyzer on top of the Mopsa platform. Our prototype supports both patch and portability analysis. We experimented it on real-world open source and industrial software. It is able to analyze successfully real patches from the repositories of the GNU core utilities and the Linux kernel. It is also able to prove the endian portability of small slices of open source software, such as Linux drivers. Moreover, it allows analyzing successfully two modules of an avionics application designed to be endian portable. The first module features about 300,000 lines of C, and is analyzed in 9.7 hours. The second module features about a million lines of C, and is analyzed in 20.4 hours.

Future work

This thesis leaves multiple directions for future investigation.

Industrialization. A first objective is to industrialize our prototype implementation on top of Mopsa, so that it may be used as part of industrial processes. Candidate applications of our portability analysis include the support of avionics certification activities related to simulation fidelity. Candidate applications of our patch analysis include automating component change impact analyses in product-line architectures.

This objective involves improvements, such as addressing the portability across platforms where the plain `char` type has different signednesses, improving the performance of our algorithm for double program synthesis, supporting C bit-fields, most non-local control flow operators, and C extensions specifying a fixed, platform-independent storage order for some variables.

Portability analysis. Our portability analysis could be extended to additional properties, especially ABI-related properties.

For instance, an important property is portability between platforms where machine integers have different sizes. The case occurs typically when porting C programs from a 32-bit architecture to a 64-bit architecture. In addition, different operating systems may rely on different 64-bit data models. For instance, Linux uses the LP64 model, where `sizeof(long)=8`, while Windows uses the LLP64 model, where `sizeof(long)=4`, which may result in different behaviors. The definition of the portability property of interest is not obvious, as increasing the range of integers may remove arithmetic wrap-arounds, some of which may be intended.

Moreover, porting existing C software developed for x86 or PowerPC to a new processor such as ARM has additional pitfalls, such as the sign of the plain `char` type for the former, and endianness for the latter. In addition, unaligned data accesses well-tolerated by x86 may trigger unexpected behaviors on some ARM processors, such as exceptions, swapped memory bytes, or poor performance. Our analysis could be extended to avoid such pitfalls.

In addition, our patch analysis and our structure layout portability analysis could be tailored to ensure the portability of applications against changes in operating systems data types. Practical problems include the representation of file offsets, as well as date and time to prepare Unix-based systems for the *Year 2038 problem* (*a.k.a. Epochalypse*).

Finally, a different kind of portability property should be considered when software is reused in an environment where input values have different ranges. For instance, the failure of Ariane 5's maiden flight was caused by the reuse of part of the software of the Ariane 4 launcher in an environment where the ranges of some inputs from sensors were significantly different.

Semantic differencing. Our work on patch analysis has focused on proving equivalence so far, while other authors characterize the semantic differences between two non equivalent versions of a program. Our method can be extended to address this problem. In particular, our method could be used to infer a semantic distance between program versions. Instead of proving that they compute the same values for all outputs, we could bound the differences between program versions for selected outputs. The analysis could additionally evaluate the cost of a patch, *e.g.* the worst-case increases of the sizes of arrays, of the numbers of loop iterations, etc. Finally, another interesting extension could be to define a relational "improvement" property between two program versions, that the analysis could infer. For instance, improving a program could mean removing only unwanted behaviors such as run-time errors.

Hyperproperties and information flow. When two program versions share exactly the same code, our double program structure defines a form of self-composition [21, 143]. Our double program semantics is then able to express 2-safety properties [169], which are hyperproperties [42, Sec. 4]. In practice, we conducted preliminary experiments on small pieces of code from the papers of other authors [70, 169, 20, 18], demonstrating that our analysis could prove automatically information flow properties such as secrecy and noninterference. Yet, the theoretical connections between our semantics and information

flow problems remain to be investigated, as well as practical experimentation on larger, more complex programs and more varied 2-safety properties, that may raise the need for new abstract domains.

Double trace semantics. Our concrete collecting semantics for double programs is based on pairs of program states. This semantics could be redefined as an abstraction of a semantics based on pairs of traces, the same way classic state semantics can be seen as an abstraction of trace semantics [53]. Such a lower-level semantics could help study the cases of equivalent program versions with traces of different lengths, and partitioning strategies for double program analysis. It may also allow developing static analyses inferring more refined program equivalence properties, *e.g.* taking termination into account. It may finally enable analyses for (hyper-)liveness properties relating the behaviors of two program versions.

Bibliography

[1] The Be Book, https://www.haiku-os.org/legacy-docs/bebook/index.html

[2] The Haiku Operating System, https://www.haiku-os.org/

[3] DO-178B: Software considerations in airborne systems and equipment certification (1992)

[4] DO-178C: Software considerations in airborne systems and equipment certification (2011)

[5] DO-331: Model-based development and verification supplement to DO-178C and DO-278A (2011)

[6] DO-333 formal methods supplement to do-178c and do-278a. Tech. rep. (Dec 2011)

[7] National Highway Traffic Safety Administration – Toyota Unintended Acceleration Investigation. Technical assessment report, NASA Engineering and Safety Center (2011)

[8] Alexandrescu, Andrei: Three optimization tips for c++. A presentation at Facebook NYC (2012), `http://www.facebook.com/notes/facebook-engineering/three-optimization-tips-for-c/10151361643253920`

[9] Alglave, J., Cousot, P.: Ogre and pythia: An invariance proof method for weak consistency models. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. p. 3–18. POPL '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009883, `https://doi.org/10.1145/3009837.3009883`

[10] Amato, G., Rubino, M., Scozzari, F.: Inferring linear invariants with parallelotopes. Sci. Comput. Program. **148**, 161–188

[11] AT & T, The Santa Cruz Operation Inc.: System V application binary interface (1997)

[12] Baeten, J.C.M., Bergstra, J.A., Klop, J.W.: Term rewriting systems with priorities. In: Lescanne, P. (ed.) Rewriting Techniques and Applications. pp. 83–94. Springer Berlin Heidelberg, Berlin, Heidelberg (1987)

[13] Baeten, J., Bergstra, J., Klop, J., Weijland, W.: Term-rewriting systems with rule priorities. Theoretical Computer Science **67**(2), 283–301 (1989). https://doi.org/https://doi.org/10.1016/0304-3975(89)90006-6, https://www.sciencedirect.com/science/article/pii/0304397589900066

[14] Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1–2), 3–21 (jun 2008). https://doi.org/10.1016/j.scico.2007.08.001, https://doi.org/10.1016/j.scico.2007.08.001

[15] Bagnara, R., Rodríguez-Carbonell, E., Zaffanella, E.: Generation of basic semi-algebraic invariants using convex polyhedra. In: Hankin, C., Siveroni, I. (eds.) Static Analysis. pp. 19–34. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

[16] Banerjee, A., Naumann, D.A., Nikouei, M.: Relational Logic with Framing and Hypotheses. In: Lal, A., Akshay, S., Saurabh, S., Sen, S. (eds.) 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 65, pp. 11:1–11:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016). https://doi.org/10.4230/LIPIcs.FSTTCS.2016.11, http://drops.dagstuhl.de/opus/volltexte/2016/6846

[17] Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects. pp. 364–387. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)

[18] Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011: Formal Methods. pp. 200–214. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[19] Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: Asymmetric product programs for relational program verification. In: Artemov, S., Nerode, A. (eds.) Logical Foundations of Computer Science. pp. 29–43. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[20] Barthe, G., Crespo, J.M., Kunz, C.: Product programs and relational program logics. Journal of Logical and Algebraic Methods in Programming **85**(5, Part 2), 847–859 (2016). https://doi.org/https://doi.org/10.1016/j.jlamp.2016.05.004, https://www.sciencedirect.com/science/article/pii/S235222081630044X, articles dedicated to Prof. J. N. Oliveira on the occasion of his 60th birthday

[21] Barthe, G., D'argenio, P.R., Rezk, T.: Secure information flow by self-composition. Mathematical. Structures in Comp. Sci. **21**(6), 1207–1252 (Dec 2011)

[22] Bedin França, R., Favre-Felix, D., Leroy, X., Pantel, M., Souyris, J.: Towards formally verified optimizing compilation in flight control software. In: Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011). OASIcs, vol. 18, pp. 59–68 (2011)

[23] Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.F.: Revising hull and box consistency. In: Proceedings of the 1999 International Conference on Logic Programming. p. 230–244. Massachusetts Institute of Technology, USA (1999)

[24] Berry, G.: Scade: Synchronous design and validation of embedded control software. In: Ramesh, S., Sampath, P. (eds.) Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems. pp. 19–33. Springer Netherlands, Dordrecht (2007)

[25] Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis and verification of aerospace software by abstract interpretation. In: AIAA Infotech@Aerospace. pp. 1–38. No. 2010-3385, AIAA (Apr 2010)

[26] Besson, F., Blazy, S., Wilke, P.: A precise and abstract memory model for c using symbolic values. In: Garrigue, J. (ed.) Programming Languages and Systems. pp. 449–468. Springer International Publishing, Cham (2014)

[27] Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. Journal of Automated Reasoning **43**(3), 263–288 (2009), `http://xavierleroy.org/publi/Clight.pdf`

[28] Brahmi, A., Delmas, D., Essoussi, M.H., Randimbivololona, F., Atki, A., Marie, T.: Formalise to automate: deployment of a safe and cost-efficient process for avionics software. In: 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018). Toulouse, France (Jan 2018), `https://hal.archives-ouvertes.fr/hal-01708332`

[29] Brahmi, A., Essoussi, M.H., Lacabanne, P., Moya Lamiel, V., Souyris, J., Carolus, M.J., Delmas, D., Randimbivololona, F.: Industrial use of a safe and efficient formal method based software engineering process in avionics. In: 10th European Congress on Embedded Real Time Software and Systems (ERTS 2020). Toulouse, France (Jan 2020)

[30] Brat, G., Navas, J.A., Shi, N., Venet, A.: Ikos: A framework for static analysis based on abstract interpretation. In: Giannakopoulou, D., Salaün, G. (eds.) Software Engineering and Formal Methods. pp. 271–277. Springer International Publishing, Cham (2014)

[31] Brevnov, E., Domeika, M., Loenko, M., Ozhdikhin, P., Tang, X., Willkinson, H.: Bec: Bi-endian compiler technology for porting byte order sensitive applications **16** (2012)

[32] Briere, D., Traverse, P.: Airbus a320/a330/a340 electrical flight controls - a family of fault-tolerant systems. In: FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing. pp. 616–623 (1993). https://doi.org/10.1109/FTCS.1993.627364

[33] Brown, N.: Sparse: a look under the hood (2016), `https://lwn.net/Articles/689907/`

[34] Buxton, J.N., Randell, B. (eds.): Software Engineering Techniques. Report on a conference sponsored by the NATO Science Committee. NATO, Science Committee, Brussels 39, Belgium (April 1970), rome, Italy, 27^{th} to 31^{th} October 1969; `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF` – geprüft: 28. März 2004

[35] Chernikova, N.V.: Algorithm for discovering the set of all the solutions of a linear programming problem. Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki **8**(6), 1387—-1395, (In Russian), English version: USSR Computational Mathematics and Mathematical Physics, 1968, 8:6, 282–293

[36] Chevalier, M.: Proving the Security of Software-Intensive Embedded Systems by Abstract Interpretation. Ph.D. thesis, Université PSL (Nov 2020)

[37] Chevalier, M., Feret, J.: Sharing ghost variables in a collection of abstract domains. In: Beyer, D., Zufferey, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 158–179. Lecture Notes in Computer Science, Springer International Publishing (2020)

[38] Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5**(2), 56–68 (1940). https://doi.org/10.2307/2266170

[39] Churchill, B., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1027–1040. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314596, `https://doi.org/10.1145/3314221.3314596`

[40] Clarisó, R., Cortadella, J.: The octahedron abstract domain. Sci. Comput. Program. **64**(1), 115–139

[41] Clarke, E.M., Grumberg, O., Kroening, D., Peled, D.A., Veith, H.: Model checking, 2nd Edition. MIT Press (2018), `https://mitpress.mit.edu/books/model-checking-second-edition`

[42] Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (sep 2010)

[43] Cohen, D.: On holy wars and a plea for peace. Computer **14**(10), 48–54 (1981). https://doi.org/10.1109/C-M.1981.220208

[44] Comar, C., Kanig, J., Moy, Y.: ntegrating Formal Program Verification with Testing. In: Embedded Real Time Software and Systems (ERTS2012). Toulouse, France (Feb 2012), https://hal.archives-ouvertes.fr/hal-02263435

[45] Condit, J., Hackett, B., Lahiri, S.K., Qadeer, S.: Unifying type checking and property checking for low-level code. SIGPLAN Not. **44**(1), 302–314 (jan 2009). https://doi.org/10.1145/1594834.1480921, https://doi.org/10.1145/1594834.1480921

[46] Cousot, P.: Principles of Abstract Interpretation. MIT Press (2021), https://mitpress.mit.edu/books/principles-abstract-interpretation

[47] Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proc. of the 2d Int. Symp. on Programming. pp. 106–130. Dunod, Paris, France (1976)

[48] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL'77. pp. 238–252. ACM (Jan 1977)

[49] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conf. Rec. of the 6th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'79). pp. 269–282. ACM Press, New York, NY (1979)

[50] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the Astrée static analyzer. In: Proc. of ASIAN'06. LNCS, vol. 4435, pp. 272–300. Springer (Dec 2006)

[51] Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL'78. pp. 84–97. ACM (1978)

[52] Cousot, P.: Types as abstract interpretations. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 316–331. POPL '97, Association for Computing Machinery, New York, NY, USA (1997). https://doi.org/10.1145/263699.263744, https://doi.org/10.1145/263699.263744

[53] Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theor. Comput. Sci. **277**(1–2), 47–103 (apr 2002). https://doi.org/10.1016/S0304-3975(00)00313-3, https://doi.org/10.1016/S0304-3975(00)00313-3

[54] Cousot, P., Cousot, R.: Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) Programming Language Implementation and Logic Programming. pp. 269–295. Springer Berlin Heidelberg, Berlin, Heidelberg (1992)

[55] Cousot, P., Cousot, R.: Temporal abstract interpretation. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 12–25. POPL '00, Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/325694.325699, https://doi.org/10.1145/325694.325699

[56] Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. SIGPLAN Not. **37**(1), 178–190 (jan 2002). https://doi.org/10.1145/565816.503290, https://doi.org/10.1145/565816.503290

[57] Cuoq, P., Delmas, D., Duprat, S., Lamiel, V.M.: Fan-C, a Frama-C plug-in for data flow verification. In: ERTS'12. SIA (2012)

[58] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c - A software analysis perspective. In: SEFM. Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer

[59] Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: Chang, B.Y.E. (ed.) Programming Languages and Systems. pp. 127–147. Springer International Publishing, Cham (2017)

[60] Darvas, A., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Proceedings of the Second International Conference on Security in Pervasive Computing. p. 193–209. SPC'05, Springer-Verlag, Berlin, Heidelberg (2005), https://doi.org/10.1007/978-3-540-32004-3_20

[61] Delmas, D., Souyris, J.: Astrée: from research to industry. In: SAS'07, LNCS, vol. 4634, pp. 437–451. Springer (Aug 2007)

[62] Delmas, D., Duprat, S., Lamiel, V.M., Signoles, J.: Taster, a frama-c plug-in to enforce coding standards. In: ERTSS 2010: Proceedings of Embedded Real Time Software and Systems. SIA (2010)

[63] Delmas, D., Duprat, S., Monate, B., Baudin, P.: Proving temporal properties at code level for basic operators of control/command programs. In: ERTS 2006: Proceedings of Embedded Real Time Software. SIA (2008)

[64] Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of fluctuat on safety-critical avionics software. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS. Lecture Notes in Computer Science, vol. 5825, pp. 53–69. Springer (2009)

[65] Delmas, D., Miné, A.: Analysis of Program Differences with Numerical Abstract Interpretation. In: PERR 2019. Prague, Czech Republic (Apr 2019)

[66] Delmas, D., Miné, A.: Analysis of Software Patches Using Numerical Abstract Interpretation. In: Chang, B.Y.E. (ed.) Proc. of the 26th International Static Analysis Symposium (SAS'19). Lecture Notes in Computer Science, vol. 11822, pp. 225–246. Bor-Yuh Evan Chang, Springer, Porto, Portugal (Oct 2019)

[67] Delmas, D., Ouadjaout, A., Miné, A.: Static Analysis of Endian Portability by Abstract Interpretation. In: 28th Static Analysis Symposium (SAS 2021). Lecture Notes in Computer Science, vol. 12913, pp. 102–123. Springer International Publishing, Chicago, Illinois, United States (Oct 2021). https://doi.org/10.1007/978-3-030-88806-0_5, https://hal.sorbonne-universite.fr/hal-03450165

[68] Delmas, D., Ouadjaout, A., Miné, A.: Artifact for Static Analysis of Endian Portability by Abstract Interpretation (2021). https://doi.org/10.5281/zenodo.5206794

[69] Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at facebook. Commun. ACM (2019)

[70] Dufay, G., Felty, A., Matwin, S.: Privacy-sensitive information flow with jml. In: Nieuwenhuis, R. (ed.) Automated Deduction – CADE-20. pp. 116–130. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

[71] Duprat, S., Favre-Félix, D., Souyris, J.: Formal verification workbench for airbus avionics software. In: ERTS'08. SIA (2008)

[72] Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: 29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014). pp. 349–360. ASE '14, ACM (Sep 2014). https://doi.org/10.1145/2642937.2642987

[73] Feret, J.: Static analysis of digital filters. In: ESOP'04. LNCS, vol. 2986, pp. 33–48. Springer (2004)

[74] Feret, J.: The arithmetic-geometric progression abstract domain. In: Proc. of the 6th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'05). LNCS, vol. 3385, pp. 42–58. Springer (Jan 2005)

[75] Floyd, R.W.: Assigning meanings to programs. In: Proc. of the American Mathematical Society Symposia on Applied Mathematics. vol. 19, pp. 19–32. Providence, USA (1967)

[76] Floyd, R.W.: Assigning meanings to programs. Proceedings of Symposium on Applied Mathematics **19**, 19–32 (1967)

[77] Fromherz, A., Ouadjaout, A., Miné, A.: Static Value Analysis of Python Programs by Abstract Interpretation. In: NFM 2018 - 10th International Symposium NASA Formal Methods. Lecture Notes in Computer Science, vol. 10811, pp. 185–202. Springer, Newport News, VA, United States (Apr 2018). https://doi.org/10.1007/978-3-319-77935-5_14, https://hal.sorbonne-universite.fr/hal-01782390

[78] Girka, T., Mentré, D., Régis-Gianas, Y.: A mechanically checked generation of correlating programs directed by structured syntactic differences. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9364, pp. 64–79. Springer (2015). https://doi.org/10.1007/978-3-319-24953-7_6, https://doi.org/10.1007/978-3-319-24953-7_6

[79] Godlin, B., Strichman, O.: Regression verification. In: Proceedings of the 46th Annual Design Automation Conference. pp. 466–471. DAC '09, ACM, New York, NY, USA (2009)

[80] Goubault, E., Putot, S.: Static analysis of finite precision computations. In: VMCAI. pp. 232–247 (2011)

[81] Goubault, E., Putot, S.: Robustness analysis of finite precision implementations. In: Programming Languages and Systems. pp. 50–57 (2013)

[82] Granger, P.: Static analysis of arithmetic congruences. Int. Journal of Computer Mathematics **30**, 165–199 (1989)

[83] Gross, J., Ganga, I., Sridhar, T.: Geneve: Generic network virtualization encapsulation. RFC 8926, RFC Editor (November 2020)

[84] Gupta, S., Rose, A., Bansal, S.: Counterexample-guided correlation algorithm for translation validation. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428289, https://doi.org/10.1145/3428289

[85] Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (Oct 1969)

[86] Howe, J.M., King, A.: Logahedra: A new weakly relational domain. In: ATVA. Lecture Notes in Computer Science, vol. 5799, pp. 306–320. Springer

[87] Hunt, J.W., Mcilroy, M.D.: An algorithm for differential file comparison. Computer Science (1975), http://www.cs.dartmouth.edu/%7Edoug/diff.pdf

[88] Nuclear power plants – Instrumentation and control systems important to safety – Software aspects for computer-based systems performing category A functions. Standard, International Electrotechnical Commission (2006)

[89] Medical device software – Software life cycle processes. Standard, International Electrotechnical Commission (2006)

[90] Software and systems engineering – Reference model for product line engineering and management. Standard, International Organization for Standardization (2015)

[91] ISO/IEC JTC1/SC22/WG14 working group: C standard. Tech. Rep. 1124, ISO & IEC (2007)

[92] Jackson, D., Ladd, D.A.: Semantic diff: A tool for summarizing the effects of modifications. In: Proceedings of ICSM '94. pp. 243–252 (1994)

[93] Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Proc. of CAV'09. vol. 5643, pp. 661–667 (June 2009)

[94] Jeannet, B.: Bddapron: A logico-numerical abstract domain library (2009), `http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/`

[95] Jourdan, J.: Verasco: a Formally Verified C Static Analyzer. (Verasco: un analyseur statique pour C formellement vérifié). Ph.D. thesis, Paris Diderot University, France

[96] Jourdan, J., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: POPL. pp. 247–259. ACM

[97] Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: Proc. of the 11th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE19). Lecture Notes in Computer Science (LNCS), vol. 12031, pp. 1–18. Springer (Jul 2019)

[98] Journault, M., Miné, A., Ouadjaout, A.: Modular static analysis of string manipulations in C programs. In: Proc. of the 25th International Static Analysis Symposium (SAS'18). Lecture Notes in Computer Science (LNCS), vol. 11002, pp. 243–262. Springer (Sep 2018). https://doi.org/10.1007/978-3-319-99725-4_16

[99] Journault, M.: Precise and modular static analysis by abstract interpretation for the automatic proof of program soundness and contracts inference. Theses, Sorbonne Université (Nov 2019), `https://tel.archives-ouvertes.fr/tel-02947214`

[100] Journault, M., Miné, A., Ouadjaout, A.: Modular static analysis of string manipulations in C programs. In: SAS (2018)

[101] Kápl, R., Parízek, P.: Endicheck: Dynamic analysis for detecting endianness bugs. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 254–270. Springer International Publishing, Cham (2020)

[102] Karr, M.: Affine relationships among variables of a program. Acta Inf. **6**, 133–151 (1976)

[103] Kästner, D., Barrho, J., Wünsche, U., Schlickling, M., Schommer, B., Schmidt, M., Ferdinand, C., Leroy, X., Blazy, S.: CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In: ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems. pp. 1–9. 3AF, SEE, SIE, Toulouse, France (Jan 2018), https://hal.inria.fr/hal-01643290

[104] Kästner, D., Ferdinand, C.: Proving the absence of stack overflows. In: Bondavalli, A., Giandomenico, F.D. (eds.) Computer Safety, Reliability, and Security - 33rd International Conference, SAFECOMP 2014, Florence, Italy, September 10-12, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8666, pp. 202–213. Springer (2014). https://doi.org/10.1007/978-3-319-10506-2_14, https://doi.org/10.1007/978-3-319-10506-2_14

[105] Kästner, D., Mauborgne, L., Wilhelm, S., Mallon, C., Ferdinand, C.: Static Data and Control Coupling Analysis. In: 11th Embedded Real Time Systems European Congress (ERTS2022). Toulouse, France (Jun 2022), https://hal.archives-ouvertes.fr/hal-03694546

[106] Kästner, D., Miné, A., Schmidt, A., Hille, H., Mauborgne, L., Wilhelm, S., Rival, X., Feret, J., Cousot, P., Ferdinand, C.: Finding All Potential Run-Time Errors and Data Races in Automotive Software. In: WCX™ 2017 - SAE World Congress Experience. pp. 1–9. SAE International, Detroit , United States (Apr 2017). https://doi.org/10.4271/2017-01-0054, https://hal.inria.fr/hal-01674831

[107] Kiefer, M., Klebanov, V., Ulbrich, M.: Relational program reasoning using compiler ir. J. Autom. Reason. **60**(3), 337–363 (mar 2018). https://doi.org/10.1007/s10817-017-9433-5, https://doi.org/10.1007/s10817-017-9433-5

[108] King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (jul 1976). https://doi.org/10.1145/360248.360252, https://doi.org/10.1145/360248.360252

[109] Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification of pointer programs by predicate abstraction. Formal Methods in System Design **52**(3), 229–259 (Jun 2018). https://doi.org/10.1007/s10703-017-0293-8

[110] Kovács, M., Seidl, H., Finkbeiner, B.: Relational abstract interpretation for the verification of 2-hypersafety properties. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. pp. 211–222. CCS '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2508859.2516721, http://doi.acm.org/10.1145/2508859.2516721

[111] Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: Symdiff: A language-agnostic semantic diff tool for imperative programs. In: CAV. pp. 712–717 (2012)

[112] Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: Proceedings of ESEC/FSE 2013. pp. 345–355 (2013)

[113] Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proc. of CGO'04 (Mar 2004)

[114] Ponce-de León, H., Furbach, F., Heljanko, K., Meyer, R.: Portability analysis for weak memory models porthos: One tool for all models. In: Ranzato, F. (ed.) Static Analysis. pp. 299–320. Springer International Publishing, Cham (2017)

[115] Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009), `http://xavierleroy.org/publi/compcert-CACM.pdf`

[116] LeVerge, H.: A note on Chernikova's algorithm. Tech. Rep. 635, IRISA (1992)

[117] Li, M., Grigg, A., Dickerson, C., Guan, L., Ji, S.: A product line systems engineering process for variability identification and reduction. IEEE Systems Journal **13**(4), 3663–3674 (2019). https://doi.org/10.1109/JSYST.2019.2897628

[118] Logozzo, F., Fahndrich, M.: On the relative completeness of bytecode analysis versus source code analysis. In: Proceedings of the International Conference on Compiler Construction. Springer Verlag (January 2008), `https://www.microsoft.com/en-us/research/publication/on-the-relative-completeness-of-bytecode-analysis-versus-source-code-analysis/`

[119] Logozzo, F., Fähndrich, M.: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. Sci. Comput. Program. **75**(9), 796–807

[120] Mahameed, S.: Mellanox, mlx5 rdma net device support (2017), `https://lwn.net/Articles/720074/`

[121] Maleki, S., Gao, Y., Garzar´n, M.J., Wong, T., Padua, D.A.: An evaluation of vectorizing compilers. In: 2011 International Conference on Parallel Architectures and Compilation Techniques. pp. 372–382 (2011). https://doi.org/10.1109/PACT.2011.68

[122] Marinescu, P.D., Cadar, C.: Katch: High-coverage testing of software patches. In: Proceedings of ESEC/FSE 2013. pp. 235–245 (2013)

[123] Martel, M.: Propagation of roundoff errors in finite precision computations: A semantics approach. In: Programming Languages and Systems. pp. 194–208 (2002)

[124] McMullen, P.: On zonotopes. Trans. Amer. Math. Soc. **159**, 91—-110

[125] Mellanox Technologies: mlx5 core library (2020), `https://github.com/torvalds/linux/tree/master/drivers/net/ethernet/mellanox/mlx5/core`

[126] Miné, A.: Weakly relational numerical abstract domains. Ph.D. thesis, École Polytechnique (Dec 2004)

[127] Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Proc. of the ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'06). pp. 54–63. ACM (June 2006)

[128] Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**(1), 31–100 (2006)

[129] Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Proc. of the 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'06). LNCS, vol. 3855, pp. 348–363. Springer (Jan 2006)

[130] Miné, A.: Abstract domains for bit-level machine integer and floating-point operations. In: Proc. of the 4th Int. Workshop on Invariant Generation (WING'12). p. 16. No. HW-MACS-TR-0097, Computer Science, School of Mathematical and Computer Science, Heriot-Watt University, UK (Jun 2012)

[131] Miné, A.: Static analysis by abstract interpretation of concurrent programs. Tech. rep., École normale supérieure (May 2013)

[132] Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Foundations and Trends in Programming Languages (FnTPL) **4**(3–4), 120–372 (2017). https://doi.org/10.1561/2500000034, `http://www-apr.lip6.fr/~mine/publi/article-mine-FTiPL17.pdf`

[133] Miné, A., Delmas, D.: Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. In: Proc. of the 15th International Conference on Embedded Software (EMSOFT'15). pp. 65–74. IEEE CS Press (Oct 2015)

[134] Miné, A., Ouadjaout, A., Journault, M.: Design of a Modular Platform for Static Analysis. In: The Ninth Workshop on Tools for Automatic Program Analysis (TAPAS'18). Fribourg-en-Brisgau, Germany (Aug 2018), `https://hal.sorbonne-universite.fr/hal-01870001`

[135] Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: PADO. Lecture Notes in Computer Science, vol. 2053, pp. 155–172. Springer

[136] Mohan, C.K.: Term rewriting with conditionals and priority orderings. Technical Reports 56, Electrical Engineering and Computer Science (1989), `https://surface.syr.edu/eecs_techreports/56`

[137] Monat, R., Ouadjaout, A., Miné, A.: Static type analysis by abstract interpretation of Python programs. In: Proc. of the 34th European Conference on Object-Oriented Programming (ECOOP'20). Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, pp. 17:1–17:29. Dagstuhl Publishing (Jul 2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.17, `http://www-apr.lip6.fr/~mine/publi/article-monat-al-ecoop20.pdf`

[138] Monat, R.: Static type and value analysis by abstract interpretation of Python programs with native C libraries. Theses, Sorbonne Université (Nov 2021), `https://tel.archives-ouvertes.fr/tel-03533030`

[139] Monat, R., Ouadjaout, A., Miné, A.: Value and Allocation Sensitivity in Static Python Analyses. In: 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. pp. 8–13. ACM, London, United Kingdom (Jun 2020). https://doi.org/10.1145/3394451.3397205, `https://hal.sorbonne-universite.fr/hal-02876667`

[140] Monat, R., Ouadjaout, A., Miné, A.: A multilanguage static analysis of python programs with native c extensions. In: Drăgoi, C., Mukherjee, S., Namjoshi, K. (eds.) Static Analysis. pp. 323–345. Springer International Publishing, Cham (2021)

[141] Mora, F., Li, Y., Rubin, J., Chechik, M.: Client-specific equivalence checking. In: Proceedings of ASE 2018. pp. 441–451 (2018)

[142] Moy, Y., Ledinot, E., Delseny, H., Wiels, V., Monate, B.: Testing or formal verification: Do-178c alternatives and industrial experience. IEEE software **30**(3), 50–57 (2013). https://doi.org/10/gf59z8

[143] Müller, C., Kovács, M., Seidl, H.: An analysis of universal information flow based on self-composition. In: CSF 2015. pp. 380–393 (July 2015)

[144] Namjoshi, K.S., Pavlinovic, Z.: The impact of program transformations on static program analysis. In: Static Analysis. pp. 306–325. Springer, Cham (2018)

[145] Naumann, D.A.: From coupling relations to mated invariants for checking information flow. In: Proceedings of the 11th European Conference on Research in Computer Security. p. 279–296. ESORICS'06, Springer-Verlag, Berlin, Heidelberg (2006), `https://doi.org/10.1007/11863908_18`

[146] Nita, M., Grossman, D.: Automatic transformation of bit-level C code to support multiple equivalent data layouts. In: Hendren, L.J. (ed.) Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4959, pp. 85–99. Springer (2008)

[147] Ouadjaout, A., Miné, A.: A library modeling language for the static analysis of C programs. In: Proc. of the 27th International Static Analysis Symposium (SAS'20). Lecture Notes in Computer Science (LNCS), vol. 12389, pp. 223–246. Springer (Nov 2020). https://doi.org/10.1007/978-3-030-65474-0_11

[148] Ouadjaout, A., Miné, A.: A library modeling language for the static analysis of C programs. In: SAS (2020)

[149] Oucheikh, R., Berrada, I., Hichami, O.E.: The 4-octahedron abstract domain. In: NETYS. Lecture Notes in Computer Science, vol. 9944, pp. 311–317. Springer

[150] Oucheikh, R., Berrada, I., Hichami, O.E.: A hypergraph based approach for the 4-constraint satisfaction problem tractability. In: arXiv:1905.09083. p. 23

[151] Oulamara, M., Venet, A.J.: Abstract interpretation with higher-dimensional ellipsoids and conic extrapolation. In: CAV (1). Lecture Notes in Computer Science, vol. 9206, pp. 415–430. Springer

[152] Paige, R.: Future directions in program transformations. ACM Comput. Surv. **28**(4es), 170–es (dec 1996). https://doi.org/10.1145/242224.242444

[153] Partush, N., Yahav, E.: Abstract semantic differencing for numerical programs. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. pp. 238–258. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[154] Partush, N., Yahav, E.: Abstract semantic differencing via speculative correlation. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. pp. 811–828. OOPSLA '14 (2014)

[155] Pei, Y., Biswas, S., Fussell, D.S., Pingali, K.: An elementary introduction to Kalman filtering. Commun. ACM **62**(11), 122–133

[156] Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 226–237. SIGSOFT '08/FSE-16 (2008)

[157] Red Hat, Inc.: Generic network virtualization encapsulation (2017), `https://github.com/torvalds/linux/blob/master/drivers/net/geneve.c`

[158] Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Trans. Amer. Math. Soc. **74**, 358–366 (1953)

[159] Rodríguez-Carbonell, E., Kapur, D.: Program verification using automatic generation of invariants,. In: Liu, Z., Araki, K. (eds.) Theoretical Aspects of Computing - ICTAC 2004. pp. 325–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

[160] Roux, P., Jobredeaux, R., Garoche, P., Éric Feron: A generic ellipsoid abstract domain for linear time invariant systems. In: HSCC. pp. 105–114. ACM

[161] Simon, A., King, A.: The two variable per inequality abstract domain. Higher-Order and Symbolic Computation **23**(1), 87–143

[162] Simon, A., King, A.: Exploiting sparsity in polyhedral analysis. In: Proceedings of the 12th International Conference on Static Analysis. p. 336–351. SAS'05, Springer-Verlag, Berlin, Heidelberg (2005)

[163] Singh, G., Püschel, M., Vechev, M.: Fast polyhedra abstract domain. SIGPLAN Not. **52**(1), 46–59 (jan 2017). https://doi.org/10.1145/3093333.3009885

[164] Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 57–69. PLDI '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2908080.2908092, `https://doi.org/10.1145/2908080.2908092`

[165] Souyris, J., Pavec, E.L., Himbert, G., Jégu, V., Borios, G.: Computing the worst case execution time of an avionics program by abstract interpretation. In: WCET. pp. 21–24 (2005)

[166] Souyris, J., Wiels, V., Delmas, D., Delseny, H.: Formal verification of avionics software products. In: FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings. pp. 532–546 (2009)

[167] Tarski, A.: A lattice theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics **5**, 285–310 (1955)

[168] development team, T.I.: Infer, a static analysis tool for java, c++, objective-c, and c. (2021), `https://github.com/facebook/infer`

[169] Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Proceedings of the 12th International Conference on Static Analysis. p. 352–367. SAS'05, Springer-Verlag, Berlin, Heidelberg (2005)

[170] The LineageOS Project: Lineageos (2020), `https://github.com/LineageOS/`

[171] The Squashfs Project: Squashfs (2020), `https://github.com/LineageOS/android_kernel_sony_msm8960t/tree/lineage-18.1/fs/squashfs`

[172] Trostanetski, A., Grumberg, O., Kroening, D.: Modular demand-driven analysis of semantic difference for program versions. In: Proceedings of SAS 2017. pp. 405–427 (2017)

[173] Venet, A.J.: The gauge domain: Scalable analysis of linear inequality invariants. In: CAV. Lecture Notes in Computer Science, vol. 7358, pp. 139–154. Springer

[174] Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM **21**, 168–173 (1974)

[175] Zaks, A., Pnueli, A.: CoVaC: Compiler validation by program analysis of the cross-product. In: International Symposium on Formal Methods (FM 2008). Turku, Finland (May 2008)

Appendix A

Double program semantics for Nimp$_2$

A.1 Abstract semantics with unbounded queues

The complete abstract semantics of simple Nimp programs $\hat{\mathbb{S}}_k[\![s]\!] \in \mathcal{P}(\hat{\mathcal{E}}) \to \mathcal{P}(\hat{\mathcal{E}})$ and of double Nimp$_2$ programs $\hat{\mathbb{D}}[\![s]\!] \in \mathcal{P}(\hat{\mathcal{D}}) \to \mathcal{P}(\hat{\mathcal{D}})$ are displayed on Fig. A.1, and Fig. A.2, respectively.

A.2 Abstract semantics with bounded queues

The complete abstract semantics of simple Nimp programs $\hat{\mathbb{S}}^p_k[\![s]\!] \in \mathcal{P}(\hat{\mathcal{E}}_p) \to \mathcal{P}(\hat{\mathcal{E}})_p$ and of double Nimp$_2$ programs $\hat{\mathbb{D}}^p[\![s]\!] \in \mathcal{P}(\hat{\mathcal{D}}_p) \to \mathcal{P}(\hat{\mathcal{D}}_p)$ are displayed on Fig. A.3, and Fig. A.4, respectively.

A.3 Abstracting away output sequences

The complete abstract semantics of simple Nimp programs without outputs $\tilde{\mathbb{S}}^p_k[\![s]\!] \in \mathcal{P}(\tilde{\mathcal{E}}_p) \to \mathcal{P}(\tilde{\mathcal{E}}_p)$ and of double Nimp$_2^-$ programs $\tilde{\mathbb{D}}^p[\![s]\!] \in \mathcal{P}(\tilde{\mathcal{D}}_p) \to \mathcal{P}(\tilde{\mathcal{D}}_p)$ are displayed on Fig. A.5, and Fig. A.6, respectively.

A.4 Abstracting away input sequences

The complete abstract semantics of simple Nimp programs without inputs and outputs $\tilde{\mathbb{S}}^0[\![s]\!] \in \mathcal{P}(\tilde{\mathcal{E}}_0) \to \mathcal{P}(\tilde{\mathcal{E}}_0)$ and of double Nimp$_2^\star$ programs $\tilde{\mathbb{D}}^0[\![s]\!] \in \mathcal{P}(\tilde{\mathcal{D}}_0) \to \mathcal{P}(\tilde{\mathcal{D}}_0)$ are displayed on Fig. A.7, and Fig. A.8, respectively.

$$\hat{\mathbb{S}}_k[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{E}}) \to \mathcal{P}(\hat{\mathcal{E}}) \quad ; \quad k \in \{1,2\}$$

$$\hat{\mathbb{S}}_k[\![\, \mathbf{skip} \,]\!]\, \hat{X} \triangleq \hat{X}$$

$$\hat{\mathbb{S}}_k[\![\, V \leftarrow e \,]\!]\, \hat{X} \triangleq \{\, ((\rho[V \mapsto v], o), \delta, q) \mid ((\rho, o), \delta, q) \in \hat{X} \wedge v \in \mathbb{E}[\![\, e \,]\!]\rho \,\}$$

$$\hat{\mathbb{S}}_1[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!]\, \hat{X} \triangleq \left\{ ((\rho[V \mapsto \nu], o), \delta - 1, \nu \cdot q) \;\middle|\; \begin{array}{l} ((\rho, o), \delta, q) \in \hat{X} \\ \delta \leq 0 \\ a \leq \nu \leq b \end{array} \right\}$$
$$\cup \left\{ ((\rho[V \mapsto v], o), \delta - 1, q) \;\middle|\; \begin{array}{l} ((\rho, o), \delta, q \cdot v) \in \hat{X} \\ \delta > 0 \\ a \leq \nu \leq b \end{array} \right\}$$

$$\hat{\mathbb{S}}_2[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!]\, \hat{X} \triangleq \left\{ ((\rho[V \mapsto \nu], o), \delta + 1, \nu \cdot q) \;\middle|\; \begin{array}{l} ((\rho, o), \delta, q) \in \hat{X} \\ \delta \geq 0 \\ a \leq \nu \leq b \end{array} \right\}$$
$$\cup \left\{ ((\rho[V \mapsto v], o), \delta + 1, q) \;\middle|\; \begin{array}{l} ((\rho, o), \delta, q \cdot v) \in \hat{X} \\ \delta < 0 \\ a \leq \nu \leq b \end{array} \right\}$$

$$\hat{\mathbb{S}}_k[\![\, \mathbf{output}(V) \,]\!]\, \hat{X} \triangleq \{\, ((\rho, o \cdot \rho(V)), \delta, q) \mid ((\rho, o), \delta, q) \in \hat{X} \,\}$$

$$\hat{\mathbb{S}}_k[\![\, \mathbf{assert}(c) \,]\!] \triangleq \hat{\mathbb{S}}[\![\, c? \,]\!]$$

$$\hat{\mathbb{S}}_k[\![\, \mathbf{if}\ c\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!] \triangleq \hat{\mathbb{S}}_k[\![\, s \,]\!] \circ \hat{\mathbb{S}}[\![\, c? \,]\!] \dot{\cup} \hat{\mathbb{S}}_k[\![\, t \,]\!] \circ \hat{\mathbb{S}}[\![\, \neg c? \,]\!]$$

$$\hat{\mathbb{S}}_k[\![\, \mathbf{while}\ c\ \mathbf{do}\ s \,]\!]\, \hat{X} \triangleq \hat{\mathbb{S}}[\![\, \neg c? \,]\!]\, \mathrm{lfp}\, (\lambda \hat{Y}.\, \hat{X} \cup \hat{\mathbb{S}}_k[\![\, s \,]\!] \circ \hat{\mathbb{S}}[\![\, c? \,]\!]\, \hat{Y})$$

$$\hat{\mathbb{S}}_k[\![\, s;\ t \,]\!] \triangleq \hat{\mathbb{S}}_k[\![\, t \,]\!] \circ \hat{\mathbb{S}}_k[\![\, s \,]\!]$$

where $\hat{\mathbb{S}}[\![\, c? \,]\!]\, \hat{X} \triangleq \{\, ((\rho, o), \delta, q) \in \hat{X} \mid \mathrm{true} \in \mathbb{C}[\![\, c \,]\!]\rho \,\}$

Figure A.1: Abstract semantics of simple programs P_1 and P_2 with unbounded queues

$\hat{\mathbb{D}}[\![s]\!] \in \mathcal{P}(\hat{\mathcal{D}}) \to \mathcal{P}(\hat{\mathcal{D}})$

$$
\begin{array}{ll}
\hat{\mathbb{D}}[\![\textbf{skip}]\!]\, \hat{R} & \triangleq \hat{R} \\
\hat{\mathbb{D}}[\![s_1 \parallel s_2]\!] & \triangleq \hat{\mathbb{D}}_2[\![s_2]\!] \circ \hat{\mathbb{D}}_1[\![s_1]\!] \\
\hat{\mathbb{D}}[\![V \leftarrow e_1 \parallel e_2]\!] & \triangleq \hat{\mathbb{D}}_2[\![V \leftarrow e_2]\!] \circ \hat{\mathbb{D}}_1[\![V \leftarrow e_1]\!] \\
\hat{\mathbb{D}}[\![V \leftarrow e]\!] & \triangleq \hat{\mathbb{D}}_2[\![V \leftarrow e]\!] \circ \hat{\mathbb{D}}_1[\![V \leftarrow e]\!] \\
\hat{\mathbb{D}}[\![\textbf{assert}(c)]\!] & \triangleq \hat{\mathbb{D}}_2[\![\textbf{assert}(c)]\!] \circ \hat{\mathbb{D}}_1[\![\textbf{assert}(c)]\!] \\
\hat{\mathbb{D}}[\![V \leftarrow \textbf{input}(a,b)]\!] & \triangleq \hat{\mathbb{D}}_2[\![V \leftarrow \textbf{input}(a,b)]\!] \circ \hat{\mathbb{D}}_1[\![V \leftarrow \textbf{input}(a,b)]\!] \\
\hat{\mathbb{D}}[\![\textbf{output}(V)]\!] & \triangleq \hat{\mathbb{D}}_2[\![\textbf{output}(V)]\!] \circ \hat{\mathbb{D}}_1[\![\textbf{output}(V)]\!] \\
\hat{\mathbb{D}}[\![\textbf{assert\_sync}]\!]\, \hat{R} & \triangleq \{\, ((\rho_1, o_1), (\rho_2, o_2), \delta, q) \in \hat{R} \mid o_1 = o_2 \,\} \\
\hat{\mathbb{D}}[\![s\,;\, t]\!] & \triangleq \hat{\mathbb{D}}[\![t]\!] \circ \hat{\mathbb{D}}[\![s]\!] \\
\hat{\mathbb{D}}[\![\textbf{if } c_1 \parallel c_2 \textbf{ then } s \textbf{ else } t]\!] & \triangleq \hat{\mathbb{D}}[\![s]\!] \circ \hat{\mathbb{F}}_2[\![c_2]\!] \circ \hat{\mathbb{F}}_1[\![c_1]\!] \\
& \quad \dot\cup\, \hat{\mathbb{D}}_2[\![\pi_2(t)]\!] \circ \hat{\mathbb{D}}_1[\![\pi_1(s)]\!] \circ \hat{\mathbb{F}}_2[\![\neg c_2]\!] \circ \hat{\mathbb{F}}_1[\![c_1]\!] \\
& \quad \dot\cup\, \hat{\mathbb{D}}_2[\![\pi_2(s)]\!] \circ \hat{\mathbb{D}}_1[\![\pi_1(t)]\!] \circ \hat{\mathbb{F}}_2[\![c_2]\!] \circ \hat{\mathbb{F}}_1[\![\neg c_1]\!] \\
& \quad \dot\cup\, \hat{\mathbb{D}}[\![t]\!] \circ \hat{\mathbb{F}}_2[\![\neg c_2]\!] \circ \hat{\mathbb{F}}_1[\![\neg c_1]\!] \\
\hat{\mathbb{D}}[\![\textbf{if } c \textbf{ then } s \textbf{ else } t]\!] & \triangleq \hat{\mathbb{D}}[\![\textbf{if } c \parallel c \textbf{ then } s \textbf{ else } t]\!] \\
\hat{\mathbb{D}}[\![\textbf{while } c_1 \parallel c_2 \textbf{ do } s]\!]\, \hat{R} & \triangleq \hat{\mathbb{F}}_2[\![\neg c_2]\!] \circ \hat{\mathbb{F}}_1[\![\neg c_1]\!]\, (\mathrm{lfp}\ H^{\hat{R}}) \\
\hat{\mathbb{D}}[\![\textbf{while } c \textbf{ do } s]\!] & \triangleq \hat{\mathbb{D}}[\![\textbf{while } c \parallel c \textbf{ do } s]\!]
\end{array}
$$

where

$$
\begin{array}{ll}
\hat{\mathbb{D}}_1[\![s]\!]\, \hat{R} & \triangleq \{\, (r_1', r_2, \delta', q') \mid (r_1', \delta', q') \in \hat{\mathbb{S}}_1[\![s]\!]\, \{\, (r_1, \delta, q) \,\} \wedge (r_1, r_2, \delta, q) \in \hat{R} \,\} \\
\hat{\mathbb{D}}_2[\![s]\!]\, \hat{R} & \triangleq \{\, (r_1, r_2', \delta', q') \mid (r_2', \delta', q') \in \hat{\mathbb{S}}_2[\![s]\!]\, \{\, (r_2, \delta, q) \,\} \wedge (r_1, r_2, \delta, q) \in \hat{R} \,\} \\
\hat{\mathbb{F}}_k[\![c]\!]\, \hat{R} & \triangleq \{\, ((\rho_1, o_1), (\rho_2, o_2), \delta, q) \in \hat{R} \mid \mathrm{true} \in \mathbb{C}[\![c]\!]\rho_k \,\}\ ;\ k \in \{1; 2\}
\end{array}
$$

and

$$
\begin{array}{ll}
H^{\hat{R}}(\hat{S}) & \triangleq \hat{R} \\
& \quad \cup\, \hat{\mathbb{D}}[\![s]\!] \circ \hat{\mathbb{F}}_2[\![c_2]\!] \circ \hat{\mathbb{F}}_1[\![c_1]\!]\, \hat{S} \\
& \quad \cup\, \hat{\mathbb{D}}_1[\![\pi_1(s)]\!] \circ \hat{\mathbb{F}}_2[\![\neg c_2]\!] \circ \hat{\mathbb{F}}_1[\![c_1]\!]\, \hat{S} \\
& \quad \cup\, \hat{\mathbb{D}}_2[\![\pi_2(s)]\!] \circ \hat{\mathbb{F}}_2[\![c_2]\!] \circ \hat{\mathbb{F}}_1[\![\neg c_1]\!]\, \hat{S}
\end{array}
$$

Figure A.2: Abstract semantics of double programs with unbounded queues

$$\hat{\mathbb{S}}^p_k[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{E}}_p) \to \mathcal{P}(\hat{\mathcal{E}}_p) \quad ; \quad k \in \{1,2\}$$

$$\hat{\mathbb{S}}^p_k[\![\, \mathbf{skip} \,]\!]\, \hat{X}_p \triangleq \hat{X}_p$$

$$\hat{\mathbb{S}}^p_k[\![\, V \leftarrow e \,]\!]\, \hat{X}_p \triangleq \{\, ((\rho[V \mapsto v], o), \delta, q) \mid ((\rho, o), \delta, q) \in \hat{X}_p \wedge v \in \mathbb{E}[\![\, e \,]\!]\rho \,\}$$

$$\hat{\mathbb{S}}^p_1[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!]\, \hat{X}_p \triangleq$$
$$\{\, ((\rho[V \mapsto \nu], o), \delta - 1, \nu \cdot q) \mid \delta \leq 0 \wedge \nu \in [a,b] \wedge ((\rho, o), \delta, q \cdot v) \in \hat{X}_p \wedge v \in \mathbb{Z} \,\}$$
$$\cup \left\{ ((\rho[V \mapsto v], o), \delta - 1, q \cdot 0 \cdot r) \;\middle|\; \begin{array}{l} \delta \in (0,p] \wedge v \in [a,b] \\ ((\rho, o), \delta, q \cdot v \cdot r) \in \hat{X}_p \wedge \forall n < p - \delta - 1 : r_n = 0 \end{array} \right\}$$
$$\cup \{\, ((\rho[V \mapsto \nu], o), \delta - 1, q) \mid \delta > p \wedge \nu \in [a,b] \wedge ((\rho, o), \delta, q) \in \hat{X}_p \,\}$$

$$\hat{\mathbb{S}}^p_2[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!]\, \hat{X}_p \triangleq$$
$$\{\, ((\rho[V \mapsto \nu], o), \delta + 1, \nu \cdot q) \mid \delta \geq 0 \wedge \nu \in [a,b] \wedge ((\rho, o), \delta, q \cdot v) \in \hat{X}_p \wedge v \in \mathbb{Z} \,\}$$
$$\cup \left\{ ((\rho[V \mapsto v], o), \delta + 1, q \cdot 0 \cdot r) \;\middle|\; \begin{array}{l} \delta \in [-p,0) \wedge v \in [a,b] \\ ((\rho, o), \delta, q \cdot v \cdot r) \in \hat{X}_p \wedge \forall n < p - \delta - 1 : r_n = 0 \end{array} \right\}$$
$$\cup \{\, ((\rho[V \mapsto \nu], o), \delta + 1, q) \mid \delta < -p \wedge \nu \in [a,b] \wedge ((\rho, o), \delta, q) \in \hat{X}_p \,\}$$

$$\hat{\mathbb{S}}^p_k[\![\, \mathbf{output}(V) \,]\!]\, \hat{X}_p \triangleq \{\, ((\rho, o \cdot \rho(V)), \delta, q) \mid ((\rho, o), \delta, q) \in \hat{X}_p \,\}$$

$$\hat{\mathbb{S}}^p_k[\![\, \mathbf{assert}(c) \,]\!] \triangleq \hat{\mathbb{S}}^p[\![\, c? \,]\!]$$

$$\hat{\mathbb{S}}^p_k[\![\, \mathbf{if}\ c\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!] \triangleq \hat{\mathbb{S}}^p_k[\![\, s \,]\!] \circ \hat{\mathbb{S}}^p[\![\, c? \,]\!] \;\dot\cup\; \hat{\mathbb{S}}^p_k[\![\, t \,]\!] \circ \hat{\mathbb{S}}^p[\![\, \neg c? \,]\!]$$

$$\hat{\mathbb{S}}^p_k[\![\, \mathbf{while}\ c\ \mathbf{do}\ s \,]\!]\, \hat{X}_p \triangleq \hat{\mathbb{S}}^p[\![\, \neg c? \,]\!]\, \mathrm{lfp}\, (\lambda \hat{Y}_p .\, \hat{X}_p \cup \hat{\mathbb{S}}^p_k[\![\, s \,]\!] \circ \hat{\mathbb{S}}^p[\![\, c? \,]\!]\, \hat{Y}_p)$$

$$\hat{\mathbb{S}}^p_k[\![\, s;\ t \,]\!] \triangleq \hat{\mathbb{S}}^p_k[\![\, t \,]\!] \circ \hat{\mathbb{S}}^p_k[\![\, s \,]\!]$$

where $\hat{\mathbb{S}}^p[\![\, c? \,]\!]\, \hat{X}_p \triangleq \{\, ((\rho, o), \delta, q) \in \hat{X}_p \mid \mathrm{true} \in \mathbb{C}[\![\, c \,]\!]\rho \,\}$

Figure A.3: Abstract semantics of simple programs P_1 and P_2 with queues of length $p \geq 1$.

$\hat{\mathbb{D}}^p[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{D}}_p) \to \mathcal{P}(\hat{\mathcal{D}}_p)$

$$
\begin{array}{ll}
\hat{\mathbb{D}}^p[\![\, \mathbf{skip} \,]\!]\, \hat{R}_p & \triangleq \hat{R}_p \\
\hat{\mathbb{D}}^p[\![\, s_1 \parallel s_2 \,]\!] & \triangleq \hat{\mathbb{D}}^p_2[\![\, s_2 \,]\!] \circ \hat{\mathbb{D}}^p_1[\![\, s_1 \,]\!] \\
\hat{\mathbb{D}}^p[\![\, V \leftarrow e_1 \parallel e_2 \,]\!] & \triangleq \hat{\mathbb{D}}^p_2[\![\, V \leftarrow e_2 \,]\!] \circ \hat{\mathbb{D}}^p_1[\![\, V \leftarrow e_1 \,]\!] \\
\hat{\mathbb{D}}^p[\![\, V \leftarrow e \,]\!] & \triangleq \hat{\mathbb{D}}^p_2[\![\, V \leftarrow e \,]\!] \circ \hat{\mathbb{D}}^p_1[\![\, V \leftarrow e \,]\!] \\
\hat{\mathbb{D}}^p[\![\, \mathbf{assert}(c) \,]\!] & \triangleq \hat{\mathbb{D}}^p_2[\![\, \mathbf{assert}(c) \,]\!] \circ \hat{\mathbb{D}}^p_1[\![\, \mathbf{assert}(c) \,]\!] \\
\hat{\mathbb{D}}^p[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!] & \triangleq \hat{\mathbb{D}}^p_2[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!] \circ \hat{\mathbb{D}}^p_1[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!] \\
\hat{\mathbb{D}}^p[\![\, \mathbf{output}(V) \,]\!] & \triangleq \hat{\mathbb{D}}^p_2[\![\, \mathbf{output}(V) \,]\!] \circ \hat{\mathbb{D}}^p_1[\![\, \mathbf{output}(V) \,]\!] \\
\hat{\mathbb{D}}^p[\![\, \mathbf{assert\_sync} \,]\!]\, \hat{R}_p & \triangleq \{\, ((\rho_1,o_1),(\rho_2,o_2),\delta,q) \in \hat{R}_p \mid o_1 = o_2 \,\} \\
\hat{\mathbb{D}}^p[\![\, s\,;\, t \,]\!] & \triangleq \hat{\mathbb{D}}^p[\![\, t \,]\!] \circ \hat{\mathbb{D}}^p[\![\, s \,]\!] \\
\hat{\mathbb{D}}^p[\![\, \mathbf{if}\ c_1 \parallel c_2\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!] & \triangleq \hat{\mathbb{D}}^p[\![\, s \,]\!] \circ \hat{\mathbb{F}}^p_2[\![\, c_2 \,]\!] \circ \hat{\mathbb{F}}^p_1[\![\, c_1 \,]\!] \\
& \quad \dot\cup\, \hat{\mathbb{D}}^p_2[\![\, \pi_2(t) \,]\!] \circ \hat{\mathbb{D}}^p_1[\![\, \pi_1(s) \,]\!] \circ \hat{\mathbb{F}}^p_2[\![\, \neg c_2 \,]\!] \circ \hat{\mathbb{F}}^p_1[\![\, c_1 \,]\!] \\
& \quad \dot\cup\, \hat{\mathbb{D}}^p_2[\![\, \pi_2(s) \,]\!] \circ \hat{\mathbb{D}}^p_1[\![\, \pi_1(t) \,]\!] \circ \hat{\mathbb{F}}^p_2[\![\, c_2 \,]\!] \circ \hat{\mathbb{F}}^p_1[\![\, \neg c_1 \,]\!] \\
& \quad \dot\cup\, \hat{\mathbb{D}}^p[\![\, t \,]\!] \circ \hat{\mathbb{F}}^p_2[\![\, \neg c_2 \,]\!] \circ \hat{\mathbb{F}}^p_1[\![\, \neg c_1 \,]\!] \\
\hat{\mathbb{D}}^p[\![\, \mathbf{if}\ c\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!] & \triangleq \hat{\mathbb{D}}^p[\![\, \mathbf{if}\ c \parallel c\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!] \\
\hat{\mathbb{D}}^p[\![\, \mathbf{while}\ c_1 \parallel c_2\ \mathbf{do}\ s \,]\!]\, \hat{R}_p & \triangleq \hat{\mathbb{F}}^p_2[\![\, \neg c_2 \,]\!] \circ \hat{\mathbb{F}}^p_1[\![\, \neg c_1 \,]\!]\, (\mathrm{lfp}\ H^{\hat{R}_p}) \\
\hat{\mathbb{D}}^p[\![\, \mathbf{while}\ c\ \mathbf{do}\ s \,]\!] & \triangleq \hat{\mathbb{D}}^p[\![\, \mathbf{while}\ c \parallel c\ \mathbf{do}\ s \,]\!]
\end{array}
$$

where

$$
\begin{array}{lll}
\hat{\mathbb{D}}^p_1[\![\, s \,]\!]\, \hat{R}_p & \triangleq \{\, (r_1',r_2,\delta',q') \mid (r_1',\delta',q') \in \hat{\mathbb{S}}^p_1[\![\, s \,]\!]\, \{\, (r_1,\delta,q) \,\} \wedge (r_1,r_2,\delta,q) \in \hat{R}_p \,\} \\
\hat{\mathbb{D}}^p_2[\![\, s \,]\!]\, \hat{R}_p & \triangleq \{\, (r_1,r_2',\delta',q') \mid (r_2',\delta',q') \in \hat{\mathbb{S}}^p_2[\![\, s \,]\!]\, \{\, (r_2,\delta,q) \,\} \wedge (r_1,r_2,\delta,q) \in \hat{R}_p \,\} \\
\hat{\mathbb{F}}^p_k[\![\, c \,]\!]\, \hat{R}_p & \triangleq \{\, ((\rho_1,o_1),(\rho_2,o_2),\delta,q) \in \hat{R}_p \mid \mathrm{true} \in \mathbb{C}[\![\, c \,]\!]\rho_k \,\}\ ;\ k \in \{1;2\}
\end{array}
$$

and

$$
\begin{array}{ll}
H^{\hat{R}_p}(\hat{S}_p) & \triangleq \hat{R}_p \\
& \quad \cup\, \hat{\mathbb{D}}^p[\![\, s \,]\!] \circ \hat{\mathbb{F}}^p_2[\![\, c_2 \,]\!] \circ \hat{\mathbb{F}}^p_1[\![\, c_1 \,]\!]\, \hat{S}_p \\
& \quad \cup\, \hat{\mathbb{D}}^p_1[\![\, \pi_1(s) \,]\!] \circ \hat{\mathbb{F}}^p_2[\![\, \neg c_2 \,]\!] \circ \hat{\mathbb{F}}^p_1[\![\, c_1 \,]\!]\, \hat{S}_p \\
& \quad \cup\, \hat{\mathbb{D}}^p_2[\![\, \pi_2(s) \,]\!] \circ \hat{\mathbb{F}}^p_2[\![\, c_2 \,]\!] \circ \hat{\mathbb{F}}^p_1[\![\, \neg c_1 \,]\!]\, \hat{S}_p
\end{array}
$$

Figure A.4: Abstract semantics of double programs with queues of length $p \geq 1$

$$\tilde{\mathbb{S}}^p_k[\![\, s \,]\!] \in \mathcal{P}(\tilde{\mathcal{E}}_p) \to \mathcal{P}(\tilde{\mathcal{E}}_p) \quad ; \quad k \in \{1, 2\}$$

$$\tilde{\mathbb{S}}^p_k[\![\, \mathbf{skip} \,]\!]\, \tilde{X}_p \triangleq \tilde{X}_p$$

$$\tilde{\mathbb{S}}^p_k[\![\, V \leftarrow e \,]\!]\, \tilde{X}_p \triangleq \{\, (\rho[V \mapsto v], \delta, q) \mid (\rho, \delta, q) \in \tilde{X}_p \wedge v \in \mathbb{E}[\![\, e \,]\!]\rho \,\}$$

$$\begin{aligned}
\tilde{\mathbb{S}}^p_1[\![\, V \leftarrow \mathbf{input}(a, b) \,]\!]\, \tilde{X}_p \triangleq\ & \\
& \{\, (\rho[V \mapsto \nu], \delta - 1, \nu \cdot q) \mid \delta \leq 0 \wedge \nu \in [a, b] \wedge (\rho, \delta, q \cdot v) \in \tilde{X}_p \wedge v \in \mathbb{Z} \,\} \\
\cup & \left\{ (\rho[V \mapsto v], \delta - 1, q \cdot 0 \cdot r) \;\middle|\; \begin{array}{l} \delta \in (0, p] \wedge v \in [a, b] \\ (\rho, \delta, q \cdot v \cdot r) \in \tilde{X}_p \wedge \forall n < p - \delta - 1 : r_n = 0 \end{array} \right\} \\
\cup & \{\, (\rho[V \mapsto \nu], \delta - 1, q) \mid \delta > p \wedge \nu \in [a, b] \wedge (\rho, \delta, q) \in \tilde{X}_p \,\}
\end{aligned}$$

$$\begin{aligned}
\tilde{\mathbb{S}}^p_2[\![\, V \leftarrow \mathbf{input}(a, b) \,]\!]\, \tilde{X}_p \triangleq\ & \\
& \{\, (\rho[V \mapsto \nu], \delta + 1, \nu \cdot q) \mid \delta \geq 0 \wedge \nu \in [a, b] \wedge (\rho, \delta, q \cdot v) \in \tilde{X}_p \wedge v \in \mathbb{Z} \,\} \\
\cup & \left\{ (\rho[V \mapsto v], \delta + 1, q \cdot 0 \cdot r) \;\middle|\; \begin{array}{l} \delta \in [-p, 0) \wedge v \in [a, b] \\ (\rho, \delta, q \cdot v \cdot r) \in \tilde{X}_p \wedge \forall n < p - \delta - 1 : r_n = 0 \end{array} \right\} \\
\cup & \{\, (\rho[V \mapsto \nu], \delta + 1, q) \mid \delta < -p \wedge \nu \in [a, b] \wedge (\rho, \delta, q) \in \tilde{X}_p \,\}
\end{aligned}$$

$$\tilde{\mathbb{S}}^p_k[\![\, \mathbf{assert}(c) \,]\!] \triangleq \tilde{\mathbb{S}}^p[\![\, c? \,]\!]$$

$$\tilde{\mathbb{S}}^p_k[\![\, \mathbf{if}\ c\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!] \triangleq \tilde{\mathbb{S}}^p_k[\![\, s \,]\!] \circ \tilde{\mathbb{S}}^p[\![\, c? \,]\!] \dot{\cup} \tilde{\mathbb{S}}^p_k[\![\, t \,]\!] \circ \tilde{\mathbb{S}}^p[\![\, \neg c? \,]\!]$$

$$\tilde{\mathbb{S}}^p_k[\![\, \mathbf{while}\ c\ \mathbf{do}\ s \,]\!]\, \tilde{X}_p \triangleq \tilde{\mathbb{S}}^p[\![\, \neg c? \,]\!]\, \mathrm{lfp}\, (\lambda \tilde{Y}_p.\, \tilde{X}_p \cup \tilde{\mathbb{S}}^p_k[\![\, s \,]\!] \circ \tilde{\mathbb{S}}^p[\![\, c? \,]\!]\, \tilde{Y}_p)$$

$$\tilde{\mathbb{S}}^p_k[\![\, s;\ t \,]\!] \triangleq \tilde{\mathbb{S}}^p_k[\![\, t \,]\!] \circ \tilde{\mathbb{S}}^p_k[\![\, s \,]\!]$$

where $\tilde{\mathbb{S}}^p[\![\, c? \,]\!]\, \tilde{X}_p \triangleq \{\, (\rho, \delta, q) \in \tilde{X}_p \mid \mathrm{true} \in \mathbb{C}[\![\, c \,]\!]\rho \,\}$

Figure A.5: Abstract semantics of simple NIMP programs P_1 and P_2 without outputs with input queues of length $p \geq 1$.

$\tilde{\mathbb{D}}^p[\![\, s \,]\!] \in \mathcal{P}(\tilde{\mathcal{D}}_p) \to \mathcal{P}(\tilde{\mathcal{D}}_p)$

$$
\begin{array}{ll}
\tilde{\mathbb{D}}^p[\![\, \mathbf{skip} \,]\!]\, \tilde{R}_p & \triangleq \tilde{R}_p \\
\tilde{\mathbb{D}}^p[\![\, s_1 \parallel s_2 \,]\!] & \triangleq \tilde{\mathbb{D}}^p_2[\![\, s_2 \,]\!] \circ \tilde{\mathbb{D}}^p_1[\![\, s_1 \,]\!] \\
\tilde{\mathbb{D}}^p[\![\, V \leftarrow e_1 \parallel e_2 \,]\!] & \triangleq \tilde{\mathbb{D}}^p_2[\![\, V \leftarrow e_2 \,]\!] \circ \tilde{\mathbb{D}}^p_1[\![\, V \leftarrow e_1 \,]\!] \\
\tilde{\mathbb{D}}^p[\![\, V \leftarrow e \,]\!] & \triangleq \tilde{\mathbb{D}}^p_2[\![\, V \leftarrow e \,]\!] \circ \tilde{\mathbb{D}}^p_1[\![\, V \leftarrow e \,]\!] \\
\tilde{\mathbb{D}}^p[\![\, \mathbf{assert}(c) \,]\!] & \triangleq \tilde{\mathbb{D}}^p_2[\![\, \mathbf{assert}(c) \,]\!] \circ \tilde{\mathbb{D}}^p_1[\![\, \mathbf{assert}(c) \,]\!] \\
\tilde{\mathbb{D}}^p[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!] & \triangleq \tilde{\mathbb{D}}^p_2[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!] \circ \tilde{\mathbb{D}}^p_1[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!] \\
\tilde{\mathbb{D}}^p[\![\, \mathbf{assert\_sync}(V) \,]\!]\, \tilde{R}_p & \triangleq \{\, (\rho_1, \rho_2, \delta, q) \in \tilde{R}_p \mid \rho_1(V) = \rho_2(V) \,\} \\
\tilde{\mathbb{D}}^p[\![\, s \,;\, t \,]\!] & \triangleq \tilde{\mathbb{D}}^p[\![\, t \,]\!] \circ \tilde{\mathbb{D}}^p[\![\, s \,]\!] \\
\tilde{\mathbb{D}}^p[\![\, \mathbf{if}\ c_1 \parallel c_2\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!] & \triangleq \tilde{\mathbb{D}}^p[\![\, s \,]\!] \circ \tilde{\mathbb{F}}^p_2[\![\, c_2 \,]\!] \circ \tilde{\mathbb{F}}^p_1[\![\, c_1 \,]\!] \\
& \quad \dot\cup\, \tilde{\mathbb{D}}^p_2[\![\, \pi_2(t) \,]\!] \circ \tilde{\mathbb{D}}^p_1[\![\, \pi_1(s) \,]\!] \circ \tilde{\mathbb{F}}^p_2[\![\, \neg c_2 \,]\!] \circ \tilde{\mathbb{F}}^p_1[\![\, c_1 \,]\!] \\
& \quad \dot\cup\, \tilde{\mathbb{D}}^p_2[\![\, \pi_2(s) \,]\!] \circ \tilde{\mathbb{D}}^p_1[\![\, \pi_1(t) \,]\!] \circ \tilde{\mathbb{F}}^p_2[\![\, c_2 \,]\!] \circ \tilde{\mathbb{F}}^p_1[\![\, \neg c_1 \,]\!] \\
& \quad \dot\cup\, \tilde{\mathbb{D}}^p[\![\, t \,]\!] \circ \tilde{\mathbb{F}}^p_2[\![\, \neg c_2 \,]\!] \circ \tilde{\mathbb{F}}^p_1[\![\, \neg c_1 \,]\!] \\
\tilde{\mathbb{D}}^p[\![\, \mathbf{if}\ c\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!] & \triangleq \tilde{\mathbb{D}}^p[\![\, \mathbf{if}\ c \parallel c\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!] \\
\tilde{\mathbb{D}}^p[\![\, \mathbf{while}\ c_1 \parallel c_2\ \mathbf{do}\ s \,]\!]\, \tilde{R}_p & \triangleq \tilde{\mathbb{F}}^p_2[\![\, \neg c_2 \,]\!] \circ \tilde{\mathbb{F}}^p_1[\![\, \neg c_1 \,]\!]\, (\mathrm{lfp}\ H^{\tilde{R}_p}) \\
\tilde{\mathbb{D}}^p[\![\, \mathbf{while}\ c\ \mathbf{do}\ s \,]\!] & \triangleq \tilde{\mathbb{D}}^p[\![\, \mathbf{while}\ c \parallel c\ \mathbf{do}\ s \,]\!]
\end{array}
$$

where

$$
\begin{array}{lll}
\tilde{\mathbb{D}}^p_1[\![\, s \,]\!]\, \tilde{R}_p & \triangleq \left\{ (\rho_1', \rho_2, \delta', q') \;\middle|\; \begin{array}{l} (\rho_1', \delta', q') \in \tilde{\mathbb{S}}^p_1[\![\, s \,]\!]\, \{\, (\rho_1, \delta, q) \,\} \\ (\rho_1, \rho_2, \delta, q) \in \tilde{R}_p \end{array} \right\} \\
\tilde{\mathbb{D}}^p_2[\![\, s \,]\!]\, \tilde{R}_p & \triangleq \left\{ (\rho_1, \rho_2', \delta', q') \;\middle|\; \begin{array}{l} (\rho_2', \delta', q') \in \tilde{\mathbb{S}}^p_2[\![\, s \,]\!]\, \{\, (\rho_2, \delta, q) \,\} \\ (\rho_1, \rho_2, \delta, q) \in \tilde{R}_p \end{array} \right\} \\
\tilde{\mathbb{F}}^p_k[\![\, c \,]\!]\, \tilde{R}_p & \triangleq \{\, (\rho_1, \rho_2, \delta, q) \in \tilde{R}_p \mid \mathrm{true} \in \mathbb{C}[\![\, c \,]\!]\rho_k \,\}\ ;\ k \in \{1; 2\}
\end{array}
$$

and

$$
\begin{array}{ll}
H^{\tilde{R}_p}(\tilde{S}_p) & \triangleq \tilde{R}_p \\
& \quad \cup\, \tilde{\mathbb{D}}^p[\![\, s \,]\!] \circ \tilde{\mathbb{F}}^p_2[\![\, c_2 \,]\!] \circ \tilde{\mathbb{F}}^p_1[\![\, c_1 \,]\!]\, \tilde{S}_p \\
& \quad \cup\, \tilde{\mathbb{D}}^p_1[\![\, \pi_1(s) \,]\!] \circ \tilde{\mathbb{F}}^p_2[\![\, \neg c_2 \,]\!] \circ \tilde{\mathbb{F}}^p_1[\![\, c_1 \,]\!]\, \tilde{S}_p \\
& \quad \cup\, \tilde{\mathbb{D}}^p_2[\![\, \pi_2(s) \,]\!] \circ \tilde{\mathbb{F}}^p_2[\![\, c_2 \,]\!] \circ \tilde{\mathbb{F}}^p_1[\![\, \neg c_1 \,]\!]\, \tilde{S}_p
\end{array}
$$

Figure A.6: Abstract semantics of double NIMP$_2^-$ programs with input queues of length $p \geq 1$.

$\tilde{\mathbb{S}}^0[\![\, s \,]\!] \in \mathcal{P}(\tilde{\mathcal{E}}_0) \to \mathcal{P}(\tilde{\mathcal{E}}_0)$; $k \in \{1, 2\}$

$$
\begin{array}{ll}
\tilde{\mathbb{S}}^0[\![\, \mathbf{skip} \,]\!]\, \tilde{X}_0 & \triangleq \tilde{X}_0 \\
\tilde{\mathbb{S}}^0[\![\, V \leftarrow e \,]\!]\, \tilde{X}_0 & \triangleq \{\, \rho[V \mapsto v] \mid \rho \in \tilde{X}_0 \wedge v \in \mathbb{E}[\![\, e \,]\!]\rho \,\} \\
\tilde{\mathbb{S}}^0[\![\, \mathbf{assert}(c) \,]\!] & \triangleq \tilde{\mathbb{S}}^p[\![\, c? \,]\!] \\
\tilde{\mathbb{S}}^0[\![\, \mathbf{if}\ c\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!] & \triangleq \tilde{\mathbb{S}}^p[\![\, s \,]\!] \circ \tilde{\mathbb{S}}^0[\![\, c? \,]\!] \dot{\cup} \tilde{\mathbb{S}}^0[\![\, t \,]\!] \circ \tilde{\mathbb{S}}^0[\![\, \neg c? \,]\!] \\
\tilde{\mathbb{S}}^0[\![\, \mathbf{while}\ c\ \mathbf{do}\ s \,]\!]\, \tilde{X}_0 & \triangleq \tilde{\mathbb{S}}^0[\![\, \neg c? \,]\!]\, \mathrm{lfp}\, (\lambda \tilde{Y}_0.\, \tilde{X}_0 \cup \tilde{\mathbb{S}}^0[\![\, s \,]\!] \circ \tilde{\mathbb{S}}^0[\![\, c? \,]\!]\, \tilde{Y}_0) \\
\tilde{\mathbb{S}}^0[\![\, s;\ t \,]\!] & \triangleq \tilde{\mathbb{S}}^0[\![\, t \,]\!] \circ \tilde{\mathbb{S}}^0[\![\, s \,]\!] \\
\quad \text{where } \tilde{\mathbb{S}}^0[\![\, c? \,]\!]\, \tilde{X}_0 & \triangleq \{\, \rho \in \tilde{X}_0 \mid \mathrm{true} \in \mathbb{C}[\![\, c \,]\!]\rho \,\}
\end{array}
$$

Figure A.7: Abstract semantics of simple NIMP programs P_1 and P_2 without inputs and outputs.

$\tilde{\mathbb{D}}^0[\![\, s \,]\!] \in \mathcal{P}(\tilde{\mathcal{D}}_0) \to \mathcal{P}(\tilde{\mathcal{D}}_0)$

$$
\begin{array}{ll}
\tilde{\mathbb{D}}^0[\![\, \mathbf{skip} \,]\!]\, \tilde{R}_0 & \triangleq \tilde{R}_0 \\
\tilde{\mathbb{D}}^0[\![\, s_1 \parallel s_2 \,]\!] & \triangleq \tilde{\mathbb{D}}^0_2[\![\, s_2 \,]\!] \circ \tilde{\mathbb{D}}^0_1[\![\, s_1 \,]\!] \\
\tilde{\mathbb{D}}^0[\![\, V \leftarrow e_1 \parallel e_2 \,]\!] & \triangleq \tilde{\mathbb{D}}^0_2[\![\, V \leftarrow e_2 \,]\!] \circ \tilde{\mathbb{D}}^0_1[\![\, V \leftarrow e_1 \,]\!] \\
\tilde{\mathbb{D}}^0[\![\, V \leftarrow e \,]\!] & \triangleq \tilde{\mathbb{D}}^0_2[\![\, V \leftarrow e \,]\!] \circ \tilde{\mathbb{D}}^0_1[\![\, V \leftarrow e \,]\!] \\
\tilde{\mathbb{D}}^0[\![\, \mathbf{assert}(c) \,]\!] & \triangleq \tilde{\mathbb{D}}^0_2[\![\, \mathbf{assert}(c) \,]\!] \circ \tilde{\mathbb{D}}^0_1[\![\, \mathbf{assert}(c) \,]\!] \\
\tilde{\mathbb{D}}^0[\![\, V \leftarrow \mathbf{input\_sync}(a, b) \,]\!]\, \tilde{R}_0 & \triangleq \{\, (\rho_1[V \mapsto v], \rho_2[V \mapsto v]) \mid v \in [a, b] \wedge (\rho_1, \rho_2) \in \tilde{R}_0 \,\} \\
\tilde{\mathbb{D}}^0[\![\, \mathbf{assert\_sync}(V) \,]\!]\, \tilde{R}_0 & \triangleq \{\, (\rho_1, \rho_2) \in \tilde{R}_0 \mid \rho_1(V) = \rho_2(V) \,\} \\
\tilde{\mathbb{D}}^0[\![\, s\,;\, t \,]\!] & \triangleq \tilde{\mathbb{D}}^0[\![\, t \,]\!] \circ \tilde{\mathbb{D}}^0[\![\, s \,]\!] \\
\tilde{\mathbb{D}}^0[\![\, \mathbf{if}\ c_1 \parallel c_2\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!] & \triangleq \tilde{\mathbb{D}}^0[\![\, s \,]\!] \circ \tilde{\mathbb{F}}^0_2[\![\, c_2 \,]\!] \circ \tilde{\mathbb{F}}^0_1[\![\, c_1 \,]\!] \\
& \dot{\cup}\, \tilde{\mathbb{D}}^0_2[\![\, \pi_2(t) \,]\!] \circ \tilde{\mathbb{D}}^0_1[\![\, \pi_1(s) \,]\!] \circ \tilde{\mathbb{F}}^0_2[\![\, \neg c_2 \,]\!] \circ \tilde{\mathbb{F}}^0_1[\![\, c_1 \,]\!] \\
& \dot{\cup}\, \tilde{\mathbb{D}}^0_2[\![\, \pi_2(s) \,]\!] \circ \tilde{\mathbb{D}}^0_1[\![\, \pi_1(t) \,]\!] \circ \tilde{\mathbb{F}}^0_2[\![\, c_2 \,]\!] \circ \tilde{\mathbb{F}}^0_1[\![\, \neg c_1 \,]\!] \\
& \dot{\cup}\, \tilde{\mathbb{D}}^0[\![\, t \,]\!] \circ \tilde{\mathbb{F}}^0_2[\![\, \neg c_2 \,]\!] \circ \tilde{\mathbb{F}}^0_1[\![\, \neg c_1 \,]\!] \\
\tilde{\mathbb{D}}^0[\![\, \mathbf{if}\ c\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!] & \triangleq \tilde{\mathbb{D}}^0[\![\, \mathbf{if}\ c \parallel c\ \mathbf{then}\ s\ \mathbf{else}\ t \,]\!] \\
\tilde{\mathbb{D}}^0[\![\, \mathbf{while}\ c_1 \parallel c_2\ \mathbf{do}\ s \,]\!]\, \tilde{R}_0 & \triangleq \tilde{\mathbb{F}}^0_2[\![\, \neg c_2 \,]\!] \circ \tilde{\mathbb{F}}^0_1[\![\, \neg c_1 \,]\!]\, (\mathrm{lfp}\ H^{\tilde{R}_0}) \\
\tilde{\mathbb{D}}^0[\![\, \mathbf{while}\ c\ \mathbf{do}\ s \,]\!] & \triangleq \tilde{\mathbb{D}}^0[\![\, \mathbf{while}\ c \parallel c\ \mathbf{do}\ s \,]\!] \\
\quad \text{where} \quad \tilde{\mathbb{D}}^0_1[\![\, s \,]\!]\, \tilde{R}_0 & \triangleq \{\, (\rho_1', \rho_2) \mid \rho_1' \in \tilde{\mathbb{S}}^0[\![\, s \,]\!]\, \{\, \rho_1 \,\} \wedge (\rho_1, \rho_2) \in \tilde{R}_0 \,\} \\
\tilde{\mathbb{D}}^0_2[\![\, s \,]\!]\, \tilde{R}_0 & \triangleq \{\, (\rho_1, \rho_2') \mid \rho_2' \in \tilde{\mathbb{S}}^0[\![\, s \,]\!]\, \{\, \rho_2 \,\} \wedge (\rho_1, \rho_2) \in \tilde{R}_0 \,\} \\
\tilde{\mathbb{F}}^0_k[\![\, c \,]\!]\, \tilde{R}_0 & \triangleq \{\, (\rho_1, \rho_2) \in \tilde{R}_0 \mid \mathrm{true} \in \mathbb{C}[\![\, c \,]\!]\rho_k \,\}\ ;\ k \in \{1; 2\} \\
\quad \text{and} \quad H^{\tilde{R}_0}(\tilde{S}_0) & \triangleq \tilde{R}_0 \\
& \cup\, \tilde{\mathbb{D}}^0[\![\, s \,]\!] \circ \tilde{\mathbb{F}}^0_2[\![\, c_2 \,]\!] \circ \tilde{\mathbb{F}}^0_1[\![\, c_1 \,]\!]\, \tilde{S}_0 \\
& \cup\, \tilde{\mathbb{D}}^0_1[\![\, \pi_1(s) \,]\!] \circ \tilde{\mathbb{F}}^0_2[\![\, \neg c_2 \,]\!] \circ \tilde{\mathbb{F}}^0_1[\![\, c_1 \,]\!]\, \tilde{S}_0 \\
& \cup\, \tilde{\mathbb{D}}^0_2[\![\, \pi_2(s) \,]\!] \circ \tilde{\mathbb{F}}^0_2[\![\, c_2 \,]\!] \circ \tilde{\mathbb{F}}^0_1[\![\, \neg c_1 \,]\!]\, \tilde{S}_0
\end{array}
$$

Figure A.8: Abstract semantics of double NIMP$^\star_2$ programs.

Appendix B

Double program semantics for C

B.1 Semantics of endian-diverse simple and double statements

A brief overview of the transfer functions of statements is available in 7.3.4. This section provides a complete description.

B.1.1 Semantics of simple statements

Before defining the semantics for double statements in this domain, we first define the semantics $\mathbb{E}^\flat_\alpha[\![*_t e]\!] \in \mathcal{D}^\flat \to \mathcal{D}^\flat \times \mathcal{P}(\mathbb{V})$ and $\mathbb{S}^\flat_\alpha[\![*_t e_1 \leftarrow e_2]\!] \in \mathcal{D}^\flat \to \mathcal{D}^\flat$ for simple memory reads and writes, in program version P_α ($\alpha \in \mathcal{A}$).

Evaluations.

We describe the semantics of $\mathbb{E}^\flat_\alpha[\![*_t e]\!] \langle C, R\rangle$, assuming the expression e does not contain any dereference. This is not restrictive, as expressions can be transformed into purely scalar expressions by resolving left-values bottom up. To compute $\mathbb{E}^\flat_\alpha[\![*_t e]\!] \langle C, R\rangle$, we first resolve $*_t e$ into a set L_α of single cells of program P_α, by evaluating e into a set of pointer values, and gathering single cells corresponding to valid pointers:

$$L_\alpha \triangleq \{ \langle V, o, t, \alpha\rangle \in \widetilde{\mathcal{C}ell}_\alpha \mid \langle V, o\rangle \in \mathbb{E}_\alpha[\![e]\!] \rho,\ \rho \in R \}$$

Then, we call $\mathit{add\text{-}cell}^\flat_\alpha$ to ensure that all the target cells in L_α occur in the abstract environment, either directly or via a suitable shared bi-cell. This updates the abstract state $\langle C, R\rangle$ to $\langle C_0, R_0\rangle$. The semantics of $\mathit{add\text{-}cell}^\flat_\alpha$ ensures that $\mathit{occ}(c, C_0) \neq \emptyset$ for all $c \in L_\alpha$.
Finally,

$$\mathbb{E}^\flat_\alpha[\![*_t e]\!] \langle C, R\rangle = \langle \langle C_0, R_0\rangle, \{ \rho_0(c_0) \mid \rho_0 \in R_0,\ c_0 \in \mathit{occ}(c, C_0),\ c \in L_\alpha \}\rangle$$

Assignments.

The semantics of assignments $\mathbb{S}^\flat_\alpha[\![*_t e_1 \leftarrow e_2]\!] \langle C, R\rangle$ involves more steps.

We first evaluate e_2 into a set of values $\mathbb{V}_{e_2} \in \mathcal{P}(\mathbb{V})$. This may involve synthesizing bi-cells for the dereferences of e_2, which updates the state $\langle C, R\rangle$ to $\langle C', R'\rangle$: $\langle\langle C', R'\rangle, \mathbb{V}_{e_2}\rangle = \mathbb{E}^\flat_\alpha[\![e_2]\!] \langle C, R\rangle$. Like for evaluations, we assume the expression e_1 does not contain any dereference, and start with resolving $*_t e_1$ into a set L_α of single cells in the memory of P_α. Then, we realize the bi-cells in L_α using *add-cell*$^\flat_\alpha$: let $\langle C_0, R_0\rangle$ be the updated environment. Some of the single cells in L_α may have been realized into shared bi-cells in C_0. Let $S_0 \triangleq (C_0 \setminus C) \cap \widetilde{\mathcal{C}ell}^2$ be the set of such shared bi-cells. Elements of S_0 represent equalities between cells in the memory of P_α, and cells in the memory of the other program version P_β ($\beta \in \mathcal{A} \setminus \{\alpha\}$). Such equalities may no longer hold, after assignment by the simple program P_α only. Therefore, we split shared bi-cells of S_0 into their left and right projections, in a copy-on-write strategy. The updated environment is

$$\langle C'_0, R'_0\rangle = \langle C_0 \cup \bigcup_{\langle c,c'\rangle \in S} \{c, c'\}, \{ c \mapsto \begin{cases} \rho(x) \text{ if } \exists x \in occ(c,S) \neq \emptyset \\ \rho(c) \text{ otherwise} \end{cases} \mid \rho \in R_0 \}\rangle$$

Finally, we update the environment for the single cells written (elements of L_α), with the possible values of e_2. However, this is not sufficient: it is also necessary to update the environment for any overlapping bi-cells, including shared bi-cells that have been split into pairs of single cells. A sound and efficient (though possibly coarse) solution is to simply remove them. Indeed, removing any bi-cell is always sound in our memory model: it amounts to losing information, as we lose constraints on the byte-representation of the memory. Let $\Omega'_0 \subseteq C'_0 \setminus L_\alpha$ be the set of such bi-cells: elements of Ω'_0 are shared bi-cells and single cells from the memory of P_α, with base variables, offsets and sizes such that they overlap some element of L_α. The updated environment is:

$$\begin{aligned}\mathbb{S}^\flat_\alpha[\![*_t e_1 \leftarrow e_2]\!] \langle C, R\rangle = \langle & C'_0 \setminus \Omega, \\ & \{ \rho_{|C'_0 \setminus \Omega}[\forall c \in L_\alpha : c \mapsto v] \mid \rho \in R'_0,\ v \in \mathbb{V}_{e_2} \}\rangle\end{aligned}$$

B.1.2 Semantics of double statements

We are now ready to define the semantics $\mathbb{D}^\flat[\![dstat]\!] \in \mathcal{D}^\flat \to \mathcal{D}^\flat$ of double statements in this domain. Like $\mathbb{D}$, $\mathbb{D}^\flat$ is defined by induction on the syntax. We focus on assignments, as other the semantics of statements is the same as in Sec. 6.2.5.

In an assignment $\mathbb{D}^\flat[\![*_t e_1 \leftarrow e_2]\!] \langle C, R\rangle$, although both programs execute the same syntactic assignment, their semantics are different, as are their endiannesses. For instance, recall Example 3 from Sec. 1.1.4. The value assigned to field `s.x` by the statement `p[4]=1` depends on the endianness of the platform. In addition, available bi-cells may be different. By default, double assignments are straightforward extensions of simple assignments: $\mathbb{D}^\flat[\![*_t e_1 \leftarrow e_2]\!] = \mathbb{S}^\flat_2[\![*_t e_1 \leftarrow e_2]\!] \circ \mathbb{S}^\flat_1[\![*_t e_1 \leftarrow e_2]\!]$. As in Sec. B.1.2, we introduce two precision optimizations, taking advantage of implicit equalities represented by shared bi-cells. We first transform $*_t e_1$ and the dereferences in e_2 into sets

of bi-cells L and R, respectively. Note that R may be empty, as e_2 may be a constant expression. Then, we realize the cells in L and R, using *add-cell*$^\flat$. Let $\langle C_0, R_0\rangle$ be the updated environment. Two optimizations are possible, depending on e_1, e_2, L, and R.

Optimization 1: Assignment of shared bi-cells. If e_1 and e_2 are deterministic expressions, and if they evaluate to bi-cells that are all shared ($L \cup R \subseteq \widetilde{\mathcal{C}ell}^2$), then $P_\mathcal{L}$ and $P_\mathcal{B}$ write the same value to the same destination. We thus update shared destination bi-cells (in L), and remove any overlapping bi-cells. Formally,

$$\mathbb{D}^\flat[\![*_t e_1 \leftarrow e_2]\!] \langle C, R\rangle = \langle C_0 \setminus \Omega, \\ \{\, \rho_{|C_0\setminus\Omega}[\forall c \in L : c \mapsto v] \mid \rho \in R_0,\, v \in \mathbb{E}^\flat_\mathcal{L}[\![e_2]\!] \langle C_0, R_0\rangle \,\}\rangle$$

where $\Omega_0 \subseteq \mathcal{C}_0 \setminus L$ is the set of (shared or single) bi-cells overlapping elements of L. The choice of evaluating $\mathbb{E}^\flat_\mathcal{L}[\![e_2]\!]$ (rather than $\mathbb{E}^\flat_\mathcal{B}[\![e_2]\!]$) is arbitrary, as they are equal. Indeed, the endianness $\mathcal{L}$ is not used by $\mathbb{E}^\flat_\mathcal{L}[\![e_2]\!]$, all the necessary cells are materialized before evaluating expression e_2.

Optimization 2: Copy assignment. If the conditions for optimization *1* are satisfied, and if, in addition, $e_2 = *_t e'_2$, and both $*_t e_1$ and $*_t e'_2$ evaluate to single bi-cells ($|L| = |R| = 1$) in every state $\langle C_0, \rho_0\rangle \in X_0$, then we are dealing with a copy assignment, as in `y=x;`. We may thus soundly copy any memory information from the source $\{r\} = R$, to the destination $\{l\} = L$, so as to further improve precision. We therefore remove l, and create a copy of r, and of any smaller bi-cell $r' \in C_0$ for the same bytes, to a corresponding bi-cell for the bytes of l. Newly created destination bi-cells have the sides and endiannesses of their sources. The environment is updated accordingly, to reflect equalities between sources and destinations.

B.2 Symbolic domain of bit-slice predicates

Abstract lattice operators and transfer functions of the $\mathcal{P}red^\sharp$ domain are described in Sec. 7.4.1. The transfer function for assignments translates general expressions into $\mathcal{B}its$ expressions. The translation relies on the function $nf \in \mathcal{N}um^\sharp \times \mathcal{C}_{\mathcal{B}its} \to \mathcal{B}its$ to rewrite bit-slice expressions to normal forms. Fig. B.1 shows the definition of nf. Constants are normalized to the smallest possible slice of themselves. So are variables, using information from the numerical abstraction. Bit-slices are pushed down into bitwise OR of terms, only if each term represents a disjoint interval of bits. nf relies on the predicate $disjoint \in \mathcal{B}its \to \mathbb{B}$ to check this property. The definition of *disjoint* is shown on Fig. B.2. Sec. 7.4.1 assumes a version of nf that additionally sorts the disjoint intervals of bits of each term of a bitwise OR of bit-slices by increasing lower bounds. We do not show this feature in Fig. B.1 for simplicity.

$nf \in \mathcal{N}um^\sharp \times \mathcal{C}_{\mathcal{B}its} \to \mathcal{B}its$

$$
\begin{array}{rcll}
\text{Notation: } n & \triangleq & 8 \times s & \text{where } s = \mathit{sizeof}(\mathit{typeof}(e)) \\
 & & & \quad e \in \mathcal{C}_{\mathcal{B}its} \text{ is the argument of } nf \\
nf\langle I, 0\rangle & \triangleq & 0 & \\
nf\langle I, c \in \mathbb{N} \setminus \{\,0\,\}\rangle & \triangleq & \overrightarrow{c[i,j)}^{i} & \text{where } i = \max\{\,0 \le p < n \mid \overrightarrow{c[p,n)}^{p} = c\,\} \\
 & & & \quad \text{and } j = \min\{\,i < q \le n \mid \overrightarrow{c[i,q)}^{i} = c\,\} \\
nf\langle I, x \in \mathcal{V}\rangle & \triangleq & \overrightarrow{x[0,j)}^{0} & \text{where } j = \min\{\,0 \le q < n \mid \mathbb{C}_{\mathcal{N}um}[\![\, x \ge 2^q \,]\!]^\sharp I = \bot\,\} \\
nf\langle I, \overrightarrow{x[i,j)}^{k}\,\rangle & \triangleq & \overrightarrow{x[i,j)}^{k} & \textbf{if } 0 \le i < j \le n \wedge k \ge 0 \wedge k + j - i \le n \\
 & & & \quad \wedge\ (i,j,k) \ne (0,0,n) \\
nf\langle I, \overrightarrow{e[0,N)}^{0}\,\rangle & \triangleq & nf\langle I, e\rangle & \textbf{if } N \ge n \\
nf\langle I, \overrightarrow{e[p,q)}^{r}\,\rangle & \triangleq & 0 & \textbf{if } p \ge q \vee r \ge n \\
nf\langle I, \overrightarrow{e[p,q)}^{r}\,\rangle & \triangleq & nf\langle I, \overrightarrow{e[p-r,q)}^{0}\,\rangle & \textbf{if } p < q \wedge r < 0 \\
nf\langle I, \overrightarrow{e[p,q)}^{r}\,\rangle & \triangleq & nf\langle I, \overrightarrow{e[p,n-r)}^{r}\,\rangle & \textbf{if } p < q \wedge 0 \le r < n \le r + q - p \\
nf\langle I, \overrightarrow{e[p,q)}^{r}\,\rangle & \triangleq & nf\langle I, \overrightarrow{e[0,q)}^{r}\,\rangle & \textbf{if } p < 0 \le q \wedge r \ge 0 \wedge r + q - p \le n \\
nf\langle I, \overrightarrow{e[i,j)}^{k}[p,q)\,\rangle & \triangleq & \overrightarrow{e[u,v)}^{w} & \text{where } u = i + \max\{\,0, p + k\,\} \wedge \\
 & & & \quad \text{and } v = i + \min\{\,q - k, j - i\,\} \wedge \\
 & & & \quad \text{and } w = r + \max\{\,0, k - p\,\} \\
nf\langle I, \overrightarrow{(e_1 \mid e_2)[p,q)}^{r}\,\rangle & \triangleq & nf\langle I, e_1' \mid e_2'\rangle & \textbf{if } \mathit{disjoint}(nf\langle I, e_1\rangle, nf\langle I, e_2\rangle) \\
 & & & \quad \text{and } e_k' = nf\langle I, \overrightarrow{e_k[p,q)}^{r}\,\rangle \\
nf\langle I, \overrightarrow{\top[p,q)}^{r}\,\rangle & \triangleq & \top & \\
nf\langle I, e \mid \top\rangle & \triangleq & \top & = nf\langle I, \top \mid e\rangle \\
nf\langle I, e \mid 0\rangle & \triangleq & nf\langle I, e\rangle & = nf\langle I, 0 \mid e\rangle \\
nf\langle I, e\rangle & \triangleq & \top & \text{in all other cases.}
\end{array}
$$

Figure B.1: Rewriting bit-slices to a normal form

$\mathit{disjoint} \in \mathcal{B}its \to \mathbb{B}$

$$
\begin{array}{rcll}
\mathit{disjoint}(0, e) \triangleq \mathit{disjoint}(0, e) & \triangleq & \text{true} & \\
\mathit{disjoint}(\top, e) \triangleq \mathit{disjoint}(\top, e) & \triangleq & \text{false} & \\
\mathit{disjoint}(\overrightarrow{x[i,j)}^{k}, \overrightarrow{y[p,q)}^{r}) & \overset{\triangle}{\iff} & [k, k + j - i] \cap [r, r + q - p] \ne \emptyset & x, y \in \mathbb{N} \cup \mathcal{V} \\
\mathit{disjoint}(e, e' \mid e'') & \overset{\triangle}{\iff} & \mathit{disjoint}(e, e') \wedge \mathit{disjoint}(e, e'') \wedge \mathit{disjoint}(e', e'') &
\end{array}
$$

Figure B.2: Disjoint normal form predicate

Appendix C

Examples

C.1 Patch analysis for Nimp_2^- programs

C.1.1 `Comp` example from [172]

```
{
  int x; int y; int z;
  x = input (-10,10);
  y = input (-100,100);

  if ( x>y || x<y ) z=1; else z=0;

  if ( z==0 || z!=0 ) {
    int tmp;
    tmp=y;
    y=x;
    x=tmp;
  }
  assert_sync(y);
}
```

C.1.2 `Const` example from [172]

```
{
    int a; int b; int c; int d; int r;
    a = input (-10,10);
    b = input (-100,100);

    {/*skip*/} || d=3;
    c = a+b || b+a;
    r = c+3 || c+d;
    assert_sync(r);
}
```

C.1.3 Modified `Fig.2` example (from [172])

```
{
    int x;
    x = input (-100,100);

    if ( x<0 ) {
      x=-1;
    }
    else {
      if ( x>=2 || x>=4 ) {}
      else {
  while (x==2) x=2;
  x=3;
      }
    }

    assert_sync(x); // x=2 ignored
}
```

$x \geq 2 \parallel x > 4$ in [172]

C.1.4 `LoopMult` example from [172]

```
{
  int a; int b; int c; int i;

  a = input(18,20);
  b = input(17,21);

  c=0;
  i=1;
  while  ( i<=b || i<=a ) {
    c = c+a || c+b;
    i=i+1;
  }

  assert_sync(c);
}
```

C.1.5 Variables switch roles: `LoopSub` example from [172]

```
{
  int x; int y; int a; int b; int c; int i;
  x = input (-100,100);
  y = input (-10,1);

  a = x || y;
  b = y || x;

  c = a || b;                                   a1 = b2 ∧ a2 = b1
  i=0;
  while  ( i<3 ) {
    c = c-b || c-a;
    i=i+1;
  }

  assert_sync(c);
}
```

C.1.6 `UnchLoop` example from [172]

```
{
  int a; int b; int c; int i; int r;

  a = input (-1000,1000);
  b = input (-1000,1000);
  c = 1 || 0;

  i=0;
  while  ( i<a ) {
    c=c+b;
    i=i+1;
  }

  r = c || c+1;
  assert_sync(r);
}
```

C.1.7 `sign` example from [153]

```
{
    int x; int sgn; int sgn2;

    x = input (-1000,1000);
    sgn = input (-100,100);

    if (x<0) sgn = -1;
    else sgn = 1;
    {/* skip */} || if (x==0) sgn=0;

    if ( x!=0 ) {
      sgn2=sgn;
      }
    assert_sync(sgn2);
}
```

C.1.8 `sum` example from [153]

```
{
  int arr; int len;
  int i; int result;

  bool b0; bool b1; bool b2; bool b3; bool b4;
  int  v0; int  v1; int  v2; int  v3; int  v4;
  b0=input(0,1); if (b0==1) v0=1; else v0=0;
  b1=input(0,1); if (b1==1) v1=1; else v1=0;
  b2=input(0,1); if (b2==1) v2=1; else v2=0;
  b3=input(0,1); if (b3==1) v3=1; else v3=0;
  b4=input(0,1); if (b4==1) v4=1; else v4=0;

  arr = input (-1000,1000);
//  len = input (0,100); // works with polyhedra and octagons
  len = v0 + 2*v1 + 4*v2 + 8*v3 + 16*v4; // len ∈ [0,31]

  result = 0;
  i = 1 || 0;
  while  ( i<len || i+1<len ) {
    {/* skip */} || i=i+1;
    result = result + arr;
    i = i+2 || i+1;
  }
  assert_sync(result);
}
```

C.1.9 `copy` example from Coreutils, and [153]

```
{
  int dest_desc;
  bool HAVE_FCHOWN;
  int fchown_dest_desc_uid_gid;
  int chown_dst_name_uid_gid;
  bool chown_failure_ok_x;
  int x_require_preserve;
  int r;

  dest_desc                = input(-1,1000);
  HAVE_FCHOWN              = input(0,1);
  chown_failure_ok_x       = rand(0,1);
  x_require_preserve       = rand(0,1);

  // equivalence only if if the syscall succeeds
  fchown_dest_desc_uid_gid = 0;
  chown_dst_name_uid_gid   = 0;

  r = 1 || 0;

  if  ( HAVE_FCHOWN == 1 && dest_desc != -1 ) {
      if ( fchown_dest_desc_uid_gid == 0 ) {
        r = 1;
      } else {
        if  ( chown_failure_ok_x == 0 ) {
          if ( x_require_preserve == 1 ) {
            r = 0 || -1;
          }
        }
      }
 } else {
      if ( chown_dst_name_uid_gid == 0 ) {
        r = 1;
      } else {
        if  ( chown_failure_ok_x == 0 ) {
          if ( x_require_preserve == 1 ) {
            r = 0 || -1;
          }
        }
      }
  }

  assert_sync(r);
}
```

C.1.10 `remove` example from Coreutils, and [153]

```
*/
/* Like fstatat, but cache the result.
    If st_size is -1, the status has not been gotten yet.
    If less than -1, fstatat failed with errno == -1 - st_size || st_ino.
    Otherwise, the status has already been gotten, so return 0.
static int
cache_fstatat (int fd, char const *file, struct stat *st, int flag)*/
{
  int r;
  int st_size; int st_ino;
  int errno; bool fstatat_ok;// true iff fstatat() returns 0
  int continue;

  st_ino = input(-1000,1000);
  st_size = -1;// before first call: the status has not been gotten yet.

  continue=1;
  while ( continue==1 ) {// cache_fstatat is called n times

  if ( st_size == -1 ) {// call fstatat
    fstatat_ok = input(0,1); // 1 + fstatat(.)
    if ( fstatat_ok == 1 ) { // fstatat behaviour
       st_size = input(0,1000);
       st_ino = input(-1000,1000);
    }
    else {
      // fstatat behaviour on systems with positive errno values
      errno = input(1,1000);
      // goal: prove non regression on systems with positive errno values
      st_size = -1 - errno || -2;
      {} || st_ino = errno;
    }
  }
  if ( 0 <= st_size ) r = 0;
  else {
    errno = -1 - st_size || st_ino;
    r = -1;
  }

  continue=input(0,1);
  }

  assert_sync(r,errno);
}
```

C.1.11 Loop rearrangements: `seq` example from Coreutils, and [153, 154]

```
{
  int first; int last;
  int x; int i; int step;
  int separator; int terminator;

  int out_of_range; int break; int print_extra_number;
  int out1; int out2;
  first = input (0,100);
  last = input (0,100);

  step = 2;
  separator = 128;
  terminator = 1024;
  print_extra_number = 0;

  {/* skip */} || if (last < first) out_of_range = 1;

  if  ( 1==1 || out_of_range == 0 ) {
    {/* skip */} || x=first;
    i = 0 || 1;
    while  ( break == 0 ) {
    {
      x = first + i * step;
      if (last < x) break = 1;
      if (break == 0) {
  if (i!=0) out1 = separator;
  out2 = x;
      }
    } || {
      out2 = x;
      if (out_of_range == 1) break = 1;
      else {
  x = first + i * step;
  if (last < x) out_of_range = 1;
  if (out_of_range == 1) if (print_extra_number == 0) break = 1;
  if (break == 0) out1 = separator;
      }
    }
    i=i+1;
    }
 }
  assert_sync(out1,out2);
  if (i!=0) out1 = terminator; || out1 = terminator;

  assert_sync(out1);
}
```

C.1.12 `test` example from Coreutils

```
// coreutils adjacent commits
// between 88c32fa68ee7057744bfb6d41f6e8eb68801306f
// and 7fd7709a7a5f3537f2f373dcc57e17001830591e (test: simplify redundant code)
//static bool  two_arguments (void)
{
  int argc; int argv_2_0; int pos;
  int cur; int next; int nextnext;
  int value;
  argc = 3;
  argv_2_0 = input(0,255); //argv[pos - 1][0] in unary_op
  pos = 1;
  cur = input(0,255);
  next = input(0,255);
  nextnext = input(0,255);

  if (cur == 97 && next == 0) //STREQ (argv[pos], "!")
    {
      pos = pos + 1;//advance (false);
      if (argv_2_0 != 0) value = 1;
      else value = 0;
      pos = pos + 1;//! one_argument ();
    }
  else if ( cur == 45         // argv[pos][0] == '-'
           && next != 0       // argv[pos][1] != '\0'
           && nextnext == 0 )// argv[pos][2] == '\0'
     {
      if  ( cur == 45         //test_unop (argv[pos])
    && (next==98||next==99||next==100||next==101||next==102||next==103||next
    ==104
        ||next == 107
        ||next==110||next==111||next==112
        ||next==114||next==115||next==116||next==117
        ||next==119||next==120
        ||next==122
        ||next==71||next==76||next==79||next==83||next==78
     ) || 1==1 ) {
        // value = unary_operator ();
  if (next == 1001) {// e
    pos = pos + 1;
    if (pos >= argc) halt;
    pos = pos + 1;
          value = input (0,1);
  }
  else if (next == 114) {// r
    pos = pos + 1;
    if (pos >= argc) halt;
    pos = pos + 1;
```

```
    value = input (0,1);
  }
  else if (next == 119) {// w
    pos = pos + 1;
    if (pos >= argc) halt;
    pos = pos + 1;
    value = input (0,1);
  }
  else if (next == 120) {// x
    pos = pos + 1;
    if (pos >= argc) halt;
    pos = pos + 1;
    value = input (0,1);
  }
  else if (next == 79) {// O
    pos = pos + 1;
    if (pos >= argc) halt;
    pos = pos + 1;
    value = input (0,1);
  }
        else if (next == 71) {//G
    pos = pos + 1;
    if (pos >= argc) halt;
    pos = pos + 1;
    value = input (0,1);
  }
⋮
  else if (next == 122) {// z
    pos = pos + 1;
    if (pos >= argc) halt;
    pos = pos + 1;
    if (argv_2_0 == 0) value = 1;
    else value = 0;
  }
  else {
          {/* skip */} || halt;
    value = 0; // default: return false;
       }
    } else {
  halt; || {/* skip */} //test_syntax_error (_("%s: unary operator expected"),
    quote (argv[pos]));
    }
  }
  else
    halt; //beyond ();

  assert_sync(value,pos);
}
```

Section	Example	swapping technique	LOC	bytes swapped 2	4	8
C.2.1	44	pointer	54	58	77	111
C.2.1	45	union	54	61	85	132
C.2.2	46	bitwise	82	51	73	96

(a) Integer byte-swaps

Section	Example	LOC	Time
7.4	41	16	30 ms
C.2.3	48	88	195 ms
C.2.3	49	19	35 ms
C.2.4	50	218	1060 ms
C.2.4	51	125	155 ms
C.2.4	52	110	150 ms

(b) Other benchmarks

Figure C.1: Small benchmarks

C.2 Endian portability analysis for C programs

In this section, we further develop the benchmarks introduced in Sec. 7.6 of Chapter 7.

Fig. C.1 shows analysis times for the benchmarks introduced in Sec. 7.6.1 and 7.6.2 of Chapter 7. It also refers to the sections of the current appendix showing related source codes, or source code excerpts. Fig. C.1(a) shows analysis times, in milliseconds, for the 9 idiomatic examples of Sec. 7.6.1, illustrating network data processing. Fig. C.1(b) shows analysis times for the other idiomatic examples of Sec. 7.6.1, as well as open source benchmarks of Sec. 7.6.2.

In the following of this section, we show source codes for benchmarks introduced in Sec. 7.6.1, and relevant excerpts from open source benchmarks introduced in Sec. 7.6.2. Note that these source codes exhibit slight differences between the notations used in Chapter 7, and the C syntax supported by the analyzer. Mainly, the static analysis primitives we use are `_mopsa_assert`, together with predicate `_sync`, rather than **assert_sync**. For instance, Examples 40 and 41 from Sec. 7.1 and 7.4 of the Chapter 7 are re-written as follows:

Example (40)**.** *(in MOPSA syntax)*

```
  read_from_network((uint8_t *)&x, sizeof(x));
# if __BYTE_ORDER == __LITTLE_ENDIAN
    uint8_t *px = (uint8_t *)&x, *py = (uint8_t *)&y;
    for (int i=0; i<sizeof(x); i++) py[i] = px[sizeof(x)-i-1];
# else
    y = x;
# endif
  _mopsa_assert(_sync(y));
```

Example (41)**.** *(in MOPSA syntax)*

```
  u16 x; u8 *p = (u8 *)&x;
  u8 y = input_sync(0,255);
# if __BYTE_ORDER == __LITTLE_ENDIAN
    x = y | 0xff00;
# else
    x = (y << 8) | 0xff;
# endif
  _mopsa_assert(_sync(p[0], p[1]));
```

Benchmarks related to network communication share a small set of stub functions, shown in Example 43. Function `read_from_network` reads a stream of bytes from an external source, and writes it into a buffer. The same stream is read by the left (little-endian) and right (big-endian) versions of a double program. Function `write_to_network` writes bytes to an external destination. The analysis should prove that the same bytes are written, by the little- and big-endian callers.

Example 43. Network stubs.

```
void read_from_network(u8 buf[], u32 size) {
  for (int i=0; i<size; i++) {
    buf[i] = input_sync(0,255);
  }
}

void write_to_network(u8 buf[], u32 size) {
  for (int i=0; i<size; i++)
    _mopsa_assert(_sync( buf[i] ));
}
```

C.2.1 Type-punning

Examples 44 and 45, introduced in Sec. 7.6.1, use type-punning to byte-swap (big-endian) network input data of type $\tau \in \{\, \mathbf{u16}, \mathbf{u32}, \mathbf{u64} \,\}$.

Type-punning with pointers.

Example 44 shows an example using pointer casts to implement type-punning.

Example 44. Type-punning with pointers.

```
# if __BYTE_ORDER == __LITTLE_ENDIAN
# define NTOH(x,y) {                                              \
    uint8_t *px = (uint8_t *)&x, *py = (uint8_t *)&y;             \
    for (int i=0; i<sizeof(x); i++) py[i] = px[sizeof(x)-i-1];    \
  }
# else
# define NTOH(x,y) {                                 \
    y=x;                                             \
  }
# endif

T x, y, z; // u8, u16, u32, or u64 integers

void main(void)
{
  read_from_network((uint8_t *)&x, sizeof(x));
  NTOH(x, y);
  y++;
  NTOH(y, z);
  write_to_network((uint8_t *)&z, sizeof(z));
}
```

Type-punning with unions.

Example 45 shows an example using union types to implement type-punning.

Example 45. Type-punning with unions

```
# if __BYTE_ORDER == __LITTLE_ENDIAN
# define NTOH(x,y) {                                              \
    for (int i=0; i<sizeof(x); i++)                               \
      y.b[i] = x.b[sizeof(x)-i-1];                                \
  }
# else
# define NTOH(x,y) {                                 \
    y.i = x.i;                                       \
  }
# endif
```

```
U x, y, z; // union of bytes and u8, u16, u32, or u64 integers

void main(void)
{
  read_from_network(x.b, sizeof(x));
  NTOH(x, y);
  y.i++;
  NTOH(y, z);
  write_to_network(z.b, sizeof(z));
}
```

C.2.2 Bitwise arithmetics

Rather than type-punning, Example 46 uses bitwise arithmetics.

Example 46. Bitwise arithmetics

```
# if __BYTE_ORDER == __LITTLE_ENDIAN
# define NTOH(x,y) {      \
    y = BITWISE_SWAP(x);  \
  }
# else
# define NTOH(x,y) {                          \
    y = x;                                    \
  }
# endif

T x, y, z; // u8, u16, u32, or u64 integers

void main(void)
{
  read_from_network((uint8_t *)&x, sizeof(x));
  NTOH(x, y);
  y++;
  NTOH(y, z);
  write_to_network((uint8_t *)&z, sizeof(z));
}
```

Example 46 relies on byte-swapping macros defined as example 47.

Example 47. Byte-swapping macros

```
/* Swap bytes in 16 bit value.  */
#define __bswap_constant_16(x)                                          \
  ((unsigned short int) ((((x) >> 8) & 0xff) | (((x) & 0xff) << 8)))

/* Swap bytes in 32 bit value.  */
#define __bswap_constant_32(x)                                  \
  ((((x) & 0xff000000) >> 24) | (((x) & 0x00ff0000) >>  8) |    \
```

```
   (((x) & 0x0000ff00) <<  8) | (((x) & 0x000000ff) << 24))

/* Swap bytes in 64 bit value.  */
# define __bswap_constant_64(x)                 \
  ((((x) & 0xff00000000000000ull) >> 56)        \
   | (((x) & 0x00ff000000000000ull) >> 40)      \
   | (((x) & 0x0000ff0000000000ull) >> 24)      \
   | (((x) & 0x000000ff00000000ull) >> 8)       \
   | (((x) & 0x00000000ff000000ull) << 8)       \
   | (((x) & 0x0000000000ff0000ull) << 24)      \
   | (((x) & 0x000000000000ff00ull) << 40)      \
   | (((x) & 0x00000000000000ffull) << 56))

# ifdef __U16__
# define BITWISE_SWAP __bswap_constant_16
# else
# ifdef __U32__
# define BITWISE_SWAP __bswap_constant_32
# else
# ifdef __U64__
# define BITWISE_SWAP __bswap_constant_64
# else
# define BITWISE_SWAP(x) (x)
# endif
# endif
# endif
```

C.2.3 Endianness of floats

Byte-swapping floats.

Example 48 extracts a double precision float from an array of bytes read from the network. Floats are byte-swapped using a combination of type-punning and bitwise arithmetics on 64-bits integers. The implementation relies on the assumption that the order of bytes is the same for integers and floats, which is the case for most machines.

Example 48. Byte-swapping floats

```
u64 ntohll(u64 x) {
  u64 y;
# if __BYTE_ORDER == __LITTLE_ENDIAN
  y = __bswap_constant_64(x);
# else
  y = x;
# endif
  return y;
}

u64 htonll(u64 x) { return ntohll(x); }

#define SIZE 20
u8 zone[SIZE];

void main(void)
{
  read_from_network(zone,SIZE);

  double y;
  _memcpy(&y, zone+10, sizeof(y));
  * (u64 *) &y = htonll(* (u64 *) &y);

  y++;

  * (u64 *) &y = htonll(* (u64 *) &y);
  _memcpy(zone+10, &y, 8);

  write_to_network(zone,SIZE);
}
```

Extracting fields from floats.

Example 49 features two ways of extracting the exponent from a double-precision float. The first is portable, the second works only for big-endian machines.

Example 49. Extracting fields from floats

```
u16 portable_exp(double x)
{
  return ((*(u64 *)&x) >> 52) & 0x7FF;
}

// works for big-endian only
u16 non_portable_exp(double x)
{
  return ((*(u32 *)&x) >> 20) & 0x7FF;
}

void main(void)
{
  double x = _mopsa_rand_double();
  _mopsa_assume(_sync(x));

  u16 e = portable_exp(x);
  _mopsa_assert( _sync( e ) ); // success

  u16 should_fail = non_portable_exp(x);
  _mopsa_assert( _sync( should_fail ) ); // should fail
}
```

C.2.4 Open source benchmarks

The benchmarks below are introduced in Sec. 7.6.2 of Chapter 7.

GENEVE.

The first benchmark is an implementation of a tunneling driver [157] based on an encapsulation network protocol [83]. It uses big-endian integers as tunnel identifiers, and the three LSBs thereof as virtual network identifiers (VNI). Therefore byte-per-byte comparisons may be used, whatever the endianness.

The driver was introduced in 2014[1] in the Linux kernel, and patched several times[2] for endianness-related issues detected by Sparse. Then, a performance optimization

[1] https://github.com/torvalds/linux/commit/0b5e8b8eeae40bae6ad7c7e91c97c3c0d0e57882

[2] https://github.com/torvalds/linux/commit/42350dcaaf1d8d95d58e8b43aee006d62c84bc2e
https://github.com/torvalds/linux/commit/0a5d1c55faa5414858857875496f6f6a9926fa51

introduced in 2016[3] a new endianness portability bug, which SPARSE failed to detect. It was fixed a year later[4].

Example 50. Geneve: device lookup.

```
static __be64 vni_to_tunnel_id(const __u8 *vni)
{
  if __IS_BIG_ENDIAN__
//#ifdef __BIG_ENDIAN
    return (vni[0] << 16) | (vni[1] << 8) | vni[2];
  else
//#else
    return (__force __be64)(((__force u64)vni[0] << 40) |
                            ((__force u64)vni[1] << 48) |
                            ((__force u64)vni[2] << 56));
//#endif
}
static bool eq_tun_id_and_vni(u8 *tun_id, u8 *vni)
{
#if 0 // was a geneve bug.
//#ifdef __BIG_ENDIAN
    return (vni[0] == tun_id[2]) &&
      (vni[1] == tun_id[1]) &&
      (vni[2] == tun_id[0]);
#else
    return !memcmp(vni, &tun_id[5], 3);
#endif
}
static struct geneve_dev *geneve_lookup(struct geneve_sock *gs,
          __be32 addr, u8 vni[])
{
  struct hlist_head *vni_list_head;
  struct geneve_dev *geneve;
  __u32 hash;

  /* Find the device for this VNI */
  hash = geneve_net_vni_hash(vni);
  vni_list_head = &gs->vni_list[hash];
  hlist_for_each_entry_rcu(geneve, vni_list_head, hlist) {
    if (eq_tun_id_and_vni((u8 *)&geneve->info.key.tun_id, vni) &&
                    addr == geneve->info.key.u.ipv4.dst)
      return geneve;
  }
  return NULL;
}
```

[3] https://github.com/torvalds/linux/commit/2e0b26e1035253bda7587f705f346385352e942d
[4] https://github.com/torvalds/linux/commit/772e97b57a4aa00170ad505a40ffad31d987ce1d

MLX5.

The second benchmark is a core library of a Linux driver [125] for ethernet and RDMA net devices [120].

An endianness issue was detected on code committed in 2017[5]. A fix was committed in 2020[6]. A second patch of the same code was committed 6 months later[7], as the first fix was incomplete.

Example 51. mlx5 net device driver.

```
static u64 mask_to_le(u64 mask, int size)
{
  __be32 mask_be32;
  __be16 mask_be16;

  if (size == 32) {
    mask_be32 = (__force __be32)(mask);
    mask = (__force unsigned long)cpu_to_le32(be32_to_cpu(mask_be32));
  } else if (size == 16) {
    mask_be32 = (__force __be32)(mask);
    mask_be16 = *(__be16 *)&mask_be32;
    mask = (__force unsigned long)cpu_to_le16(be16_to_cpu(mask_be16));
  }

  return mask;
}
```

[5] https://github.com/torvalds/linux/commit/2b64beba025109f64e688ae675985bbf72196b8c
[6] https://github.com/torvalds/linux/commit/404402abd5f90aa90a134eb9604b1750c1941529
[7] https://github.com/torvalds/linux/commit/82198d8bcdeff01d19215d712aa55031e21bccbc

Squashfs.

The third benchmark is extracted from a version of a compressed read-only filesystem [171] for Linux, used as part of the LineageOS [170] alternative operating system for Android devices. A block-processing function extracts 2-byte fields encoded in little-endian out of metadata of read request buffers. An endianness issue was introduced and fixed in patches committed in 2020[8].

Example 52. Squashfs.

```
/* Extract the length of the metadata block */
if (req->offset != msblk->devblksize - 1) {
  length = le16_to_cpup((__le16 *)
      (bh[0]->b_data + req->offset));
} else {
  length = (unsigned char)bh[0]->b_data[req->offset];
  length |= (unsigned char)bh[1]->b_data[0] << 8;
}
```

[8] https://github.com/LineageOS/android_kernel_sony_msm8960t/commit/5f61dc71accfea6c9467499d0e3eb5462dab8d63

List of Figures