Static type and value analysis by abstract interpretation of Python programs with native C libraries

Raphaël Monat

THÈSE DE DOCTORAT DE SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par
Raphaël Monat

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Sujet de la thèse :

Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries

Thèse soutenue le 22 Novembre 2021 devant le jury composé de :

Antoine Miné	Sorbonne Université & CNRS, France	Directeur de thèse
Isabella Mastroeni	Università di Verona, Italie	Rapportrice
Anders Møller	Aarhus Universitet, Danemark	Rapporteur
Emmanuel Chailloux	Sorbonne Université & CNRS, France	Président du jury
Francesco Logozzo	Facebook Seattle, États-Unis	Examinateur
Peter Müller	ETH Zürich, Suisse	Examinateur
Alan Schmitt	INRIA Rennes, France	Examinateur

Abstract

In this thesis, we aim at designing both theoretically and experimentally methods for the automatic detection of potential bugs in software – or the proof of the absence thereof. This detection is done statically by analyzing programs' source code without running them. We rely on the abstract interpretation framework to derive sound, computable semantics. In particular, we focus on analyzing dynamic programming languages. The target of this work is the analysis of Python programs combined with native C libraries.

The abstract interpretation workflow requires a concrete semantics, which will be approximated to create sound computable analyses. Our first step is thus to define the uncomputable, collecting semantics of Python formally. This semantics relies on previous work and retro-engineering from the source code of the reference interpreter. We took special care to make the semantics explainable by providing references to the actual implementation's source code for each case of the semantics.

We have implemented all the analyses presented in this thesis within the Mopsa framework. From a static analysis developer's perspective, Mopsa has a double goal. It aims at reducing the cost of lifting an abstract domain from a toy language to a real-world one. It also aspires to define relational analyses composed of loosely coupled abstract domains that can cooperate. We take particular care to define all our abstract domains independently from each other. Mopsa's core is language agnostic, making Mopsa suitable to analyze different programming languages. The analyses rely on some general use domains, such as numerical domains, the recency abstraction of dynamic memory allocation, and the loop iterator.

We present a first analysis aiming at detecting type-related errors in Python programs. It does so by inferring both the nominal and structural types of objects. Python's type annotations can be reused to support libraries. This analysis is refined into a value analysis which is more precise and infers numerical invariants for the manipulated builtin numeric values. These analyses rely on abstractions of dynamic memory allocation and data structures that we have adapted to the case of dynamic programming languages. Both analyses scale to real-world benchmarks a few thousand lines long used by Python developers.

Most Python programs rely on libraries written in C for the sake of performance or code reuse. We define a multilanguage analysis that can detect runtime errors in every part of a multilanguage program: in the Python part, in the C part, and at the boundary between the languages. Before defining this analysis, we provide an overview of the different interoperability mechanisms available between Python and C, and we define a collecting semantics for the interoperability mechanism provided by the reference interpreter. This analysis can tackle tests of popular, real-world libraries a few thousand lines of C and Python long in a few minutes.

Résumé

Dans cette thèse, nous avons pour objectif de concevoir, à la fois théoriquement et expérimentalement, des méthodes pour la détection automatique de bogues potentiels dans les logiciels – ou la preuve de leur absence. Ces méthodes sont statiques : elles analysent le code source des programmes sans les exécuter. Nos travaux s'inscrivent dans le cadre de l'interprétation abstraite pour dériver une sémantique sûre et décidable. Le principal objet de ce travail est l'analyse des langages de programmation dynamiques. En particulier, ce travail se concentre sur les programmes écrits en Python, qui peuvent appeler des bibliothèques écrites en C.

En suivant le déroulement usuel de l'interprétation abstraite, nous partons d'une sémantique concrète, qui sera ensuite approximée pour créer des analyses sûres et calculables. Notre première étape consiste donc à définir formellement la sémantique collectrice, et non calculable, de Python. Cette sémantique s'appuie sur des travaux antérieurs et sur une rétro-ingénierie à partir du code source de l'interprète de référence. Nous avons pris soin de rendre la sémantique explicable en mettant, pour chaque cas, des références vers le code source correspondant dans l'interprète.

Nous avons implémenté toutes les analyses présentées dans cette thèse dans le logiciel intitulé Mopsa. Du point de vue de l'implémentation des analyses statiques, Mopsa a un double objectif. Il vise tout d'abord à réduire le coût du passage d'un domaine abstrait fonctionnant sur un langage jouet à un langage de programmation réel. Il aspire également à définir des analyses relationnelles composées de domaines abstraits faiblement couplés qui peuvent coopérer. Nous prenons un soin particulier à définir tous nos domaines abstraits de manière indépendante les uns des autres. Le noyau de Mopsa n'est pas spécialisé pour l'analyse d'un langage en particulier, ce qui rend Mopsa apte à analyser différents langages de programmation. Les analyses s'appuient sur certains domaines généraux, tels que les domaines numériques et l'itérateur de boucle.

Nous présentons une première analyse visant à détecter les erreurs liées aux types dans les programmes Python. Elle le fait en déduisant les types nominaux et structurels des objets. Les annotations de type de Python peuvent être utilisées pour supporter des librairies. Cette analyse est raffinée en une analyse de valeur qui est plus précise et infère des invariants numériques. Ces deux analyses reposent sur des abstractions du mécanisme d'allocation dynamique de mémoire et des abstractions de structures de données que nous avons adaptées au cas des langages de programmation dynamiques. Nos analyses fonctionnent sur des benchmarks réels de quelques milliers de lignes utilisés par les développeurs Python.

La plupart des programmes Python s'appuient sur des bibliothèques écrites en C, afin d'améliorer leurs performances ou de réutiliser du code existant. Nous définissons une analyse multilangage qui peut détecter les erreurs à l'exécution dans chaque partie d'un programme multilangage : dans la partie Python, dans la partie C, et à la frontière entre les langages. Avant de définir cette analyse multilangage, nous fournissons un aperçu des différents mécanismes d'interopérabilité disponibles entre Python et C, ainsi que la définition d'une sémantique collectrice pour le mécanisme d'interopérabilité fourni par l'interprète de référence. Notre analyse multilangage est capable d'analyser en quelques minutes les tests de bibliothèques populaires, issues du monde réel, faisant quelques milliers de lignes de C et de Python.

Acknowledgments & Remerciements

I am deeply grateful to Isabella Mastroeni and Anders Møller for having agreed to review this dissertation in the busy months of September and October.

I would like to thank Emmanuel Chailloux, Francesco Logozzo, Peter Müller and Alan Schmitt for agreeing to participate to the defense committee.

Merci Antoine pour toutes les discussions extrêmement enrichissantes et tes conseils avisés. Merci pour tes relectures infatigables, dans les moindres détails, avec des corrections, mais aussi des suggestions d'améliorations, ainsi que des idées de travaux futurs. Rien que pour le manuscrit, les différents brouillons cumulent autour de 800 annotations.

Merci Abdelraouf : sans toi et ni Mopsa, cette thèse aurait pris beaucoup plus de temps, et l'analyse multilangage n'aurait pas été si facile à faire ! Je me rappelle encore des après-midi passés en début de thèse à dessiner des diagrammes de séquence pour comprendre comment l'analyse de Python pourrait marcher. J'ai gardé cette approche pour illustrer l'analyse multilangage dans les figures 11.6, 11.7 et 11.8.

Merci à vous deux pour votre bienveillance constante, pour avoir toujours trouvé du temps pour discuter malgré vos emplois du temps chargés, et pour toutes les discussions captivantes autour de Mopsa.

Merci Alan et Pierre-Évariste pour vos précieux conseils lors des comités de suivi.

Merci à Daniel Hirschkoff, Simon Castellan et Jean-Marie Madiot pour m'avoir fait tomber dans la potion magique des langages de programmation en L3, et fait découvrir – entre autres – les rudiments de l'analyse de programmes.

Merci aux précédents thésards d'Antoine : Caterina, Thibault, Ghiles et Matthieu, pour les discussions intéressantes, vos conseils, et les modèles que vous fournissez. Bon courage à David, Guillaume et Francesco.

APR est un environnement à la fois accueillant et stimulant pour faire une thèse. Merci au bureau 303 (ou assimilés) pour le bout de chemin fait ensemble : Alice, Vincent, Marwan, Boubacar, Clément, Steven, Yi-Ting, Martin, Jules, Mehdi, Mat(t)hieu, Keanu et Pierre. Merci à Romain (Demangeon) pour toutes les discussions impromptues qui permettent de changer d'air, et pour m'avoir mis le pied à l'étrier de l'enseignement. Merci aussi à Maryse, Mathieu, Pascal, Frédéric, Tong, Emmanuel, Antoine (Genitrini), Philippe et Tali pour les enseignements donnés ensembles.

Denis, merci de m'avoir proposé de t'aider sur cet à-côté que semblait être le reverse-engineering des impôts durant un cours d'escrime. Je suis heureux que l'on ait pû faire évoluer cela un vrai projet de recherche et de transfert vers la DGFiP. Cela m'a conforté dans mon positionnement du côté "pratique" des méthodes formelles, et dans l'intérêt de pouvoir suivre ses propres pistes. Merci Jonathan pour ton aide précieuse lors de la rédaction de l'article CC. Merci aux agents de la DGFiP pour leur accueil et leur patience.

Merci à Romain (Dubessy) de m'avoir appris à gérer un groupe d'escrimeurs. Cela m'a beaucoup facilité la prise de parole face à des groupes de TD. Merci à Alex, et Jean-Pierre pour votre animation de l'ECE. Merci à Mathilde pour l'enseignement des groupes débutants à deux en 2019, et la reprise du flambeau cette année. Merci aux habitués du club pour ces années passées ensemble : Adrien, Arthur, Charles, Denis (de nouveau), Irène, José, Kim, Léo, Marion, Mathilde, Philippine, Soazic, Théo, Yoko, Vincent, ...

I've been glad to meet fencers from all over Europe during competitions, and make friends from Amsterdam, Mainz and Polytechnique.

Merci à Guillaume pour tes cours d'escrime à la fac, qui permettent toujours de se changer la tête dans la bonne humeur, tout en progressant quelque soit le niveau de départ. Merci aussi à Maria (Mihaescu) et Lucas pour l'opposition fournie au fil des années.

Merci à Maria (Boritchev), Titouan, Isao, Etienne, Victor, Arthur, Amandine, Simon et Jean-Yves pour les années passées à Lyon ensemble. Maria, on essaiera de moins factoriser nos soutenances dans une autre vie !

Loïc, Carole, je suis toujours aussi heureux de vous avoir rencontrés en prépa, et de vous voir quand nos emplois du temps s'alignent. Merci à Fanny, Louise, Marc, Sylvain et Victor pour tous les moments passés ensembles.

Merci à mes parents et à mes soeurs pour votre soutien continu. Je ne serais jamais arrivé jusque là sans vous.

Laura, merci pour tout.

Contents

Part I

Background

1

Introduction

1.1 Software bugs and what can be done about it

Software is ubiquitous and touching every aspect of modern life, from its use in transportation means to personal communication systems. A bug happens in a software when the latter does not behave as it is expected to. For example, it can reach an erroneous state terminating its execution. Depending on the nature of the bug and the criticality of the host system, a bug ranges from being annoying (a random crash in a smartphone app), to having a monetary cost reaching billions of dollars, to losses of life. These bugs happen a lot in everyday software. Wong et al. [156] survey 59 notable software accidents which happened from 1993 to 2015. In a 2020 report [86], the Consortium for Information & Software Quality estimated that $607 billions were spent in 2020 on finding and fixing software bugs in the US alone, and that those software failures cost $1.56 trillion. At the time of writing, the CVE (Common Vulnerabilities and Exposure) database of the NIST (National Institute of Standards and Technology) [120] lists more than 169,371 software vulnerabilities that could be exploited by attackers on publicly available software alone.

1.1.1 A first approach: testing

A first way to avoid bugs is by testing that the software works as intended on specific input cases. There are, however two difficulties to this approach. First, generating test cases is difficult and time-consuming, since developers have to compute the expected result on each input case, and this independently from the software's implementation. Second, testing every input case is impossible in most cases, where the input space is too big to be exhaustively covered.

Let us take the example of developers wanting to check their implementation of a tax computation system, which given taxpayers' information – such as salaries, relationship's status, number of children, ... – yields the income tax owed. Provided an input case describing a fiscal household, developers would have to compute manually (using the law as their guide) the income tax owed before checking that their software returns the same result. In that case, it is straightforward to understand that testing the software on every fiscal household would be a tremendous task. In addition, it would defeat the whole point of automating the tax computation software in the first place.

In some cases, some industrial actors have found testing to be too costly in their industrial processes and introduced alternative approaches to validation by testing. In the domain of critical embedded software, Airbus is an example [43]:

> Considering the steady increase of the size and complexity of this kind of software, classical validation and verification processes, based on massive testing campaigns and complementary intellectual analyses, hardly scale up within reasonable costs. Therefore, Airbus has decided to introduce formal proof techniques providing product-based assurance into its own verification processes.

1.1.2 An impossibility theorem

It would thus be nice to have techniques and methods avoiding these shortcomings, that is, having approaches that can automatically and generally prove that programs are correct (i.e., free of bugs). However, we know there is no such general approach since 1953. At that date, Henry Gordon Rice [128] proved that "all non-trivial semantic properties of programs are undecidable". In a simplified way, this means that proving program properties cannot be done by another program that will always return the correct result in finite time. This theorem can be seen as a generalization of Alan Turing's theorem on the undecidability of the halting problem [151]. Scientists in the field of formal methods, aiming at studying bugs in programs, have thus had to circumvent this impossibility theorem by picking at most two of the three following properties in their approaches:

- **Completeness.** An approach is complete if it is possible to prove any true property using that approach. In particular, these approaches ensure the absence of false positives: if a bug is detected, it necessarily exists in the analyzed program.
- **Soundness.** An approach is sound if any property it proves on a program actually holds on the program. In particular, they do not exhibit false negatives, i.e., if the approach proves that a program is free of bugs, it is.
- **Automation.** An approach is automatic if it does not require user interaction to finish its task and can do so in finite time.

We briefly survey the different static approaches that work on programs' source code without running them. Each approach has its benefits and downsides, and some are more suitable for specific cases.

1.1.3 Deductive program verification

The field of deductive verification relies on sound and complete tools that are guided by their users. These tools are useful to prove strong properties, such as the correctness of a program with respect to a specification. They usually require a lot of expertise and guidance from a user to complete proofs, although some parts can be discharged using automatic SAT/SMT solvers. These tools tend to work on a custom input programming language, with options to extract programs into more mainstream languages. Examples of deductive program verification frameworks include the Why3 platform [52] or F* [144]. In the case of Why3, the input language is called WhyML; (sub)proofs can be discharged to SMT solvers such as Z3 [41], or a proof assistant. In addition, the Frama-C platform [8] builds upon Why3 to allow deductive verification of original C code.

1.1.4 Symbolic execution

Symbolic execution approaches replace unknown variables' values with symbolic variables and propagate them during the execution. They perform explicit disjunctions in the case of conditionals or loops. The collected constraints can then be solved to determine if an erroneous state has been reached or not. It is then possible to determine a concrete input state leading to the erroneous state. These disjunctions may quickly create a combinatorial explosion.

According to a recent survey by Baldoni et al. [5], symbolic execution engines “often settle for less ambitious goals, e.g., by trading soundness for performance”.

1.1.5 Model checking

Model-checking [30] generally applies to proof on decidable fragments of languages. In these cases, approaches can be sound, complete, and automatic. These approaches require a model of a finite-state machine and a formula in a given logic and decide if the formula holds in the provided model. The decision procedure can be a custom algorithm or rely on a constraint solver. In the case of general program analysis, it is also possible to restrict programs' executions. In that case, the inference of semantic properties is decidable again. For example, Clarke et al. [31] apply bounded model-checking to analyze C programs by unwinding loops, and functions calls up to a user-provided bound. They are thus unsound (if a specific behavior happens deeper into the execution of the program). Similarly to symbolic execution, these tools have the benefit of providing counterexamples when they find a bug. Multiple model checkers are compared every year in the software verification competition [12].

1.1.6 Static analysis by abstract interpretation

The focus of this thesis is the case of sound and automatic approaches. These approaches are however incomplete, and can raise false alarms. In particular, the goal of this thesis is to develop static analyses within the framework of abstract interpretation, developed by Radhia and Patrick Cousot [33]. They have been particularly successful in the verification of absence of runtime errors in the context of critical embedded software. For example, Astrée [11, 35] has been used to prove the absence of runtime errors in control and commands software of Airbus planes [43]. More recently, Frama-C's static analysis plug-in [40, 8] has been used to analyze the code of French nuclear power plants [122]. In these cases of critical software, a subset of the C programming language is used, and it precludes complex control flow as well as dynamic memory allocation. The sound, efficient, and precise analysis of more dynamic languages and program traits remains a challenge.

1.2 The challenges of analyzing dynamic programming languages

Dynamic programming languages are a relatively new class of programming languages, aiming at making software development quicker by being more implicit. They have been steadily growing in popularity in the last two decades. Examples of such dynamic programming languages include JavaScript, which is extremely popular on the client side of web pages; Python, which is initially a scripting language and the focus of this thesis; and PHP, which is a server-side scripting language. At the time of writing, JavaScript and Python are the two most used languages on GitHub, followed by Java [62].

We present some specificities of dynamic programming languages (some already pointed out by Tratt [150]), and the consequences for their analysis.

- ▷ **Permissive, high-level syntax.** Languages such as PHP and Python provide a permissive, high-level syntax, where most operators can be overloaded, by hiding an underlying complex semantics. This flexibility also relies on multiple inheritance of objects, and complex resolution of method calls. JavaScript and PHP also perform implicit, dynamic casts instead of raising errors. In the case of JavaScript, the addition of the string `"1 hello"` and the integer 2 is interpreted as a string concatenation and returns `"1 hello2"`. On the other hand, PHP chooses to perform an integer addition by casting the string into the integer 1, and returning 3. In Python, this operation yields a `TypeError` exception. In the case

of PHP and Python, a prevailing implementation acts as the reference for the semantics, as opposed to a more accessible and readable standardization that is available in JavaScript. In the absence of a standard, establishing the languages' semantics is the first challenge before developing an analysis. [1]

- ▷ **Builtin data structures.** The high-level syntax is complemented by builtin data structures such as dynamic arrays and associative maps. These data structures may contain heterogeneous elements. These cases are less studied than the case of numeric, static arrays in the context of C programs.
- ▷ **Dynamic typing.** Variables are neither statically declared nor statically typed. They can thus point to objects having different types at execution time. To handle these cases, introspection operators can inspect the type of an object at runtime and affect the control-flow. It is also possible to alter objects' structures at runtime, e.g. by adding or removing fields to them. This makes the static analyses harder: the types of variables have to be determined too, and introspection operators have to be supported in order to precisely handle the control-flow. In the case of Python, two different type systems are used: a nominal one, corresponding to the class from which an object is instantiated, and a structural one, informally called duck typing, which is based on attributes.
- ▷ **Dynamic evaluation.** Although the frequency of its usage varies heavily from one language to another, it is possible to construct a string and evaluate it as a piece of code at runtime. If no exact representation of the evaluated string is available, it is next to impossible to analyze those dynamic evaluations. Since these evaluations change the program's state, forgetting their effects yields unsound analyses.
- ▷ **Automatic memory management.** Object allocation and deallocation are handled automatically by the language. Compared to embedded C software, dynamic memory allocations thus happen frequently. Specific dynamic memory allocation abstractions have to be used and tailored to the case of dynamic programming languages. Deallocations may happen asynchronously, at any time after objects are not referenced anymore. Precisely capturing their effects, which can be arbitrary due to the finalization functions, is thus difficult.
- ▷ **Batteries-included library.** Dynamic programming languages often ship with vast standard libraries. Programs frequently rely on multiple parts of these libraries, which thus have to be supported.
- ▷ **Complex control-flow.** High-level iterators and exceptional control-flow create more dynamic control-flow. Another case of complex control-flow concerns Python's generators, and more generally asynchronous functions. The control-flow can be more global and dynamic than in the case of C software.

1.3 Contributions & outline

The first part of the thesis provides background on static analysis by abstract interpretation. Chapter 2 explains how these analyses work in the setting of a simple imperative language. In addition to providing the base terminologies, we introduce some key concepts reused in the analyses of Python programs: (i) the use of control-flow tokens to describe non-local control-flow operators on analyses working by induction on the syntax, (ii) the use of auxiliary variables by some domains, delegating work to other domains, (iii) a modular definition of relational concretizations for each domain: concretizations are defined independently for each domain, and can be composed through specific operators. These **modular definitions of concretizations** are a novel contribution used throughout this thesis.

[1] Standardizations are usually written informally and leave ambiguities. The best approach is to rely on fully formal semantics, which requires additional work – such as the work of Bodin et al. [16] for JavaScript.

The second part of this thesis defines language-agnostic concepts and abstractions. Chapter 3 describes the Mopsa static analyzer, in which the analyses presented in this thesis are implemented. Mopsa has a deep influence on the design of these analyses. It is a prototype analyzer, which aims at simplifying the definition of analyses, and focuses on making **collaborative, relational analyses**. This chapter extends a previous publication to VSTTE 2019 [82] by providing a more in-depth comparison with related work, introducing the notion of hooks, and formalizing the composition operators in Mopsa. Chapter 4 revisits the recency abstraction [4], originally developed to handle dynamic memory allocation in the analysis of binaries or low-level C code, and subsequently used in the analysis of JavaScript [74], [124]. We provide a **modular definition of the recency abstraction and its concretization** and a notion of **variable policies** used to tailor the allocation-site sensitivity. Some policies specific to the analysis of Python are defined. Chapter 5 defines abstractions of containers such as arrays and dictionaries. These are very coarse abstractions, well known for the analysis of C or Java. These **coarse containers abstractions have been defined in the setting of dynamic programming languages**, where data-structures have variable length and can potentially be heterogeneous. They are also defined to work by **delegation over scalar domains**.

The third part of this thesis focuses on Python programs. Following the usual workflow of abstract interpretation, we start by defining the **concrete semantics of a large subset of Python** in Chapter 6. A first part of the semantics covers the core language, and another the most commonly used objects. This semantics is assumed to be a correct modeling of Python's semantics, without a formal connection to the actual Python interpreter, but we took special care to make the **semantics manually checkable, using references to the actual implementation of the reference interpreter**, called CPython. The concrete semantics has been tested by running our analysis on more than 700 conformance tests. Contrary to previous works, this interpreter-like semantics is directly amenable to static analysis by abstract interpretation, improving our confidence in our analysis. Chapter 7 describes a **type analysis for Python**, focusing on precise detection of type-related exceptions. It is based on a work published at ECOOP in 2020 [110]. It can interpret **Python's type annotations**, allowing us to leverage Python annotations written by developers in the typeshed project [152] and easily support libraries. It scales to small **real-world benchmarks** (less than 2,500 LOC per program) providing analysis times of the order of a few minutes. This analysis is refined into a **numeric value analysis** in Chapter 8. This value analysis can be **relational**, and scales using **packing heuristics**. We keep a particular interest in comparing it with the type analysis on the same benchmarks, based on our SOAP'20 publication [112]. We find that the precision gained with the value analysis does not remove any type-related error in the programs we analyze, and that the non-relational value analysis requires around 3 times more memory and time than the type analysis.

The fourth part of this thesis extends the target to Python programs calling C code. This is particularly important to analyze the whole source code of a significant number of Python programs since one-in-five of the 200 most downloaded Python libraries contains C code. This part is an extension of our SAS paper published in 2021 [113]. Since the interoperability mechanisms are quite complex, we start by providing an introductory example of Python program calling a C module using the Python/C API. This chapter ends by surveying other Python/C interoperability mechanisms. Chapter 10 defines a **semantics for the Python/C API**. This semantics builds upon the semantics of each language. Chapter 11 defines a **multilanguage value analysis of Python programs using C extension modules**. This analysis relies on our previous value analysis of Python, and the work of Ouadjaout and Miné [121] for the C analysis. It reports runtime errors that may happen in Python, in C, and at the interface. Thanks to Mopsa, the abstract state is shared between abstract domains of different languages. We only add a minimal number of transfer functions. Our analyzer can tackle tests of **real-world libraries a few thousand lines of C and Python long** (5,500 at the most) in a few minutes.

The final part of this thesis concludes and reflects on future work.

Artifact. One of the main goals of this work has been to develop and maintain Mopsa so that it can analyze Python programs. In the conferences proposing it, we have successfully submitted software artifacts alongside our articles in order to make our experimental evaluation reproducible [111, 115]. Both Mopsa and its benchmarks are publicly available [78, 79]. We enrich this document with the release of an artifact of Mopsa in its current state, supporting the analysis of programs described in this thesis [114].

Collaborations. All the research done in this thesis is joint work with Antoine Miné and Abdelraouf Ouadjaout. Mopsa was already a work-in-progress for the C analysis at the beginning of this thesis, started by Antoine Miné, Abdelraouf Ouadjaout, and Matthieu Journault. The definitions of the modular concretizations have been extensively discussed with Matthieu Journault.

2

Static Analysis by Abstract Interpretation

This chapter introduces the basics of static analysis by abstract interpretation, as well as common notations used in the rest of this thesis. In general, the goal of static analyses is to prove program properties. Here, we focus on proving safety properties (i.e., that program states are included in the set of safe states). In particular, we want to prove that our programs avoid erroneous states.

More in-depth introductions to abstract interpretation have been written by Miné [106], Rival and Yi [132]. We introduce concepts of order theory and abstract interpretation gradually, once they have been encountered in examples. In a classic approach, we start by defining `Imp` in Section 2.1, the toy language on which this chapter is based. We define the semantics of `Imp` in Section 2.2, and numerical analyses in Section 2.3. More specific and recurring concepts used in this thesis are presented in Section 2.4. Section 2.5 revisits the concretizations defined in Section 2.4, through the lens of a new framework allowing the definition of modular concretization functions.

2.1 A Toy Imperative Language, `Imp`

This chapter focuses on analyzing `Imp` programs. `Imp` is a toy imperative language, where variables are declared and statically typed. At first, the only type available is `int`, the values of which are mathematical integers. There are no functions. `Imp` programs may fail if a division by zero occurs or if the division result is not an integer.

An example `Imp` program is shown in Listing 2.1. It computes the fourth term of the $3n + 1$ sequence [89] starting at three using a while loop. This program does not fail since the division by two is always performed on even numbers.

The grammar of the `Imp` language is described in Figure 2.1. Statements can be variable declarations (for now, the only variable type is `int`, for mathematical integers), variable assignments, sequences of statements, conditional `if` statements and `while` loops. Variable declarations and assignments can be combined into a single statement. Expressions are either variables (denoted $\mathcal{V}$), integers, an interval designating a random integer in this interval, or a binary operation over two expressions. Unary operators (such as the unary negation) could be added straightforwardly to our presentation. Conditionals are comparisons between two expressions.

Listing 2.1: An `Imp` program computing the $3n + 1$ sequence

```
int u = 3;
int n = 4;

int i = 0;
while (i < n) {
  if (u % 2 == 0) { u = u / 2; }
  else { u = 3 * u + 1; }
  i = i + 1;
}
// u = 8
```

⟨*statement*⟩ ::= ⟨*type*⟩ ⟨*var*⟩ | ⟨*var*⟩ `=` ⟨*expr*⟩ | ⟨*type*⟩ ⟨*var*⟩ `=` ⟨*expr*⟩
| ⟨*statement*⟩ `;` ⟨*statement*⟩
| `if` ⟨*conditional*⟩ `{` ⟨*statement*⟩ `}` `else` `{` ⟨*statement*⟩ `}`
| `while` ⟨*conditional*⟩ `{` ⟨*statement*⟩ `}`

⟨*expr*⟩ ::= ⟨*var*⟩ $\in \mathcal{V}$ | $z \in \mathbb{Z}$ | $[z_1 \in \mathbb{Z} \cup \{-\infty\}; z_2 \in \mathbb{Z} \cup \{+\infty\}]$
| ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩

⟨*conditional*⟩ ::= ⟨*expr*⟩ ⟨*compop*⟩ ⟨*expr*⟩

⟨*binop*⟩ ::= `+` | `-` | `*` | `/` | `%`

⟨*compop*⟩ ::= `<=` | `<` | `>` | `>=` | `==` | `!=`

⟨*type*⟩ ::= `int`

Figure 2.1: Grammar of `Imp` programs

Remark 2.1 **Non-determinism in the semantics**

The interval constant $[z_1 \in \mathbb{Z} \cup \{-\infty\}; z_2 \in \mathbb{Z} \cup \{+\infty\}]$ is used to model non-determinism. It can be used to model randomness from arbitrary input, as well as imprecisions introduced to approximate the behavior of a function.

2.2 Semantics of `Imp`

We define the semantics of `Imp`, that is, a formal mathematical definition of the behavior of expressions and statements over the program state.

Definition 2.2 **Program state**

A state $\mathcal{S}$ of an `Imp` program consists in a map from the program's variables $\mathcal{V}$ to mathematical integers, i.e $\mathcal{S} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{Z}$.

Example 2.3 **Program state**

In our running example, the initial state is $\mathcal{S} = \emptyset$ (no variable is defined), and the final state is $(i \mapsto 4, n \mapsto 4, u \mapsto 16)$.

Remark 2.4 **Set and function notations**

Sets are written $\{ a; b \}$, where the separator between two elements is a semicolon. Functions can be defined as a mapping $(i \mapsto v, j \mapsto w)$, where the separator is a comma.

The semantics acts as a function describing the effect of statements on an input state, similarly to the implementation of an interpreter, and to the implementation of an analyzer.

2.2.1 Semantics of expressions

The semantics of an expression e is written $\mathbb{E}[\![e]\!] : S \to \mathcal{P}(\mathbb{Z})$. Given an input state, the expression is evaluated in the set of output values (due to non-determinism). The semantics of expressions is shown in Figure 2.2. A variable is evaluated using the input state σ. If the variable is not defined in the state, we consider it is an error and evaluate it into the empty set. A constant integer is evaluated in itself. An interval can be evaluated into any of the integers it contains. The addition of two expressions e_1 and e_2 is evaluated into the set of values $v_1 + v_2$, where v_1 (resp. v_2) ranges over the values of e_1 (resp. e_2). The semantics of subtraction and multiplication are the same. The division of two expressions e_1 and e_2 has a similar definition, but we keep only evaluations where the denominator evaluates to a non-zero value, and the division results in an integer. Similarly, the remainder of two expressions drops the cases where the modulus is zero.

$$
\begin{aligned}
&\mathbb{E}[\![v \in \mathcal{V}]\!]\sigma \stackrel{\text{def}}{=} \{ \sigma(v) \} \\
&\mathbb{E}[\![z \in \mathbb{Z}]\!]\sigma \stackrel{\text{def}}{=} \{ z \} \\
&\mathbb{E}[\![[z_1, z_2]]\!]\sigma \stackrel{\text{def}}{=} \{ z \in \mathbb{Z} \mid z_1 \leq z \leq z_2 \} \\
&\mathbb{E}[\![e_1 \dagger e_2]\!]\sigma \stackrel{\text{def}}{=} \mathbb{E}[\![e_1]\!]\sigma \dagger \mathbb{E}[\![e_2]\!]\sigma \quad \forall \dagger \in \{ +, -, *, /, \% \}
\end{aligned}
$$

$$
\begin{aligned}
&X \dagger Y \stackrel{\text{def}}{=} \{ x \dagger y \mid x \in X, y \in Y \} \; \forall \dagger \in \{ +, -, * \} \\
&X / Y \stackrel{\text{def}}{=} \{ x/y \mid x \in X, y \in Y, y \neq 0, x\%y = 0 \} \\
&X \% Y \stackrel{\text{def}}{=} \{ x\%y \mid x \in X, y \in Y, y \neq 0 \}
\end{aligned}
$$

Figure 2.2: Semantics of expressions

Example 2.5 **Semantics & errors**

Let σ be a program state where $\sigma(x) = 2$. We evaluate $x/[-3, 3]$ in this state. We have $\mathbb{E}[\![x]\!]\sigma = \{ \sigma(x) \} = \{ 2 \}$, and $\mathbb{E}[\![[-3, 3]]\!]\sigma = \{ -3; -2; -1; 0; 1; 2; 3 \}$. Thus, $\mathbb{E}[\![x/[-3, 3]]\!]\sigma = \{ -1; -2; 2; 1 \}$: the denominator value 0 is dropped to prevent a division by zero, as well as values -3 and 3 to avoid non-integer values.

Remark 2.6 **Errors**

Erroneous evaluations and states are not explicit in the semantics. In the case of a division by zero, for example, we just continue the evaluation with the non-erroneous cases. This simplifies the definition of the semantics. In the implementation of an analyzer, erroneous states will be marked to report them, but they are not evaluated further either.

In the case of languages raising exceptions to signal errors (such as Python), the execu-

tion will be continued, and the state flagged as erroneous, since these exceptions can be caught later on.

2.2.2 Semantics of statements

Due to the non-determinism of expressions, an assignment may transform one program state into multiple ones. Given a statement s, its semantics could thus be written $\mathbb{S}[\![s]\!] : S \to \mathcal{P}(\mathcal{S})$. In order to ease the composition of semantics, we define by join-morphism extension $\mathbb{S}[\![s]\!] : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$. The semantics of statements transforms a set of states into another one.

The semantics of variable declaration, assignment, and sequence is shown in Figure 2.3. A declared variable is initialized with any value ($\sigma' = \sigma[x \mapsto z]$ denotes the function σ extended so that $\sigma'(x) = z$). Given an expression e and a set of states Σ, the semantics of the assignment $\mathbb{S}[\![x = e]\!]\Sigma$ is the set of states where variable x maps to a value v, where v is a value corresponding to the evaluation of e in a program state $\sigma \in \Sigma$. The semantics of a sequence of statements consists in the composition of the semantics of statements.

$$\mathbb{S}[\![\texttt{int } x]\!]\Sigma \overset{\text{def}}{=} \{ \sigma[x \mapsto z] \mid \sigma \in \Sigma, z \in \mathbb{Z} \}$$
$$\mathbb{S}[\![x = e]\!]\Sigma \overset{\text{def}}{=} \{ \sigma[x \mapsto v] \mid \sigma \in \Sigma, v \in \mathbb{E}[\![e]\!]\sigma \}$$
$$\mathbb{S}[\![s_1; s_2]\!] \overset{\text{def}}{=} \mathbb{S}[\![s_2]\!] \circ \mathbb{S}[\![s_1]\!]$$

Figure 2.3: Semantics of basic statements

The semantics of conditionals is shown in Figure 2.4. It consists in executing the statements of the `if` branch for the states where the guard holds and executing the statements of the `else` branch in the states where the negation of the condition holds. We thus define the conditional filtering operator $\mathbb{C}[\![c]\!]$ which filters the states to keep only those where the evaluation of the comparison may hold.

$$\mathbb{S}[\![\texttt{if } (c) \{s_t\} \texttt{ else } \{s_f\}]\!] \overset{\text{def}}{=} \mathbb{S}[\![s_t]\!] \circ \mathbb{C}[\![c]\!] \cup \mathbb{S}[\![s_f]\!] \circ \mathbb{C}[\![\neg c]\!]$$
$$\mathbb{C}[\![e_1 \trianglelefteq e_2]\!]\Sigma \overset{\text{def}}{=} \{ \sigma \in \Sigma \mid \exists v_1 \in \mathbb{E}[\![e_1]\!]\sigma, \exists v_2 \in \mathbb{E}[\![e_2]\!]\sigma, v_1 \trianglelefteq v_2 \}$$
$$\trianglelefteq \in \{ <=, <, >, >=, ==, != \}$$

Figure 2.4: Semantics of conditionals

Remark 2.7 **Conditional semantics**

To model a non-deterministic execution between two statements, we can use a conditional with a non-deterministic guard:

$$\mathbb{S}[\![\texttt{if } ([0,1] == 0) \{s_t\} \texttt{ else } \{s_f\}]\!] = \mathbb{S}[\![s_t]\!] \cup \mathbb{S}[\![s_f]\!]$$

Example 2.8 **Conditional filtering**

Let $\Sigma = \mathbb{S}[\![x = [0,2]]\!]\emptyset = \{ (x \mapsto 0); (x \mapsto 1); (x \mapsto 2) \}$. We have $\mathbb{C}[\![x < [1,2]]\!]\Sigma = \{ (x \mapsto 0); (x \mapsto 1) \}$. In particular, $\sigma = x \mapsto 1$ is a kept by the filtering operator, since $2 \in \mathbb{E}[\![[1,2]]\!]\sigma$ and $\mathbb{E}[\![x]\!]\sigma < 2$.

Remark 2.9 **Negation of conditions**

The conditional filtering operator is defined only on comparisons, but we use the negation of conditions in the definition of the semantics of `if`. These negations are translated back to a condition, e.g. $\neg(e_1 < e_2) = e_1 \geq e_2$.

The semantics of while loops is shown in Figure 2.5. It consists in iterating the loop body (along with filtering using the guard) any number of times. Then, we keep the states where the negation of the guard holds (i.e., when the loop exits).

$$\mathbb{S}[\![\,\texttt{while}\ (c)\ \{s\}\,]\!]\Sigma \overset{\text{def}}{=} \mathbb{C}[\![\,\neg c\,]\!]\left(\bigcup_{n\in\mathbb{N}} (\mathbb{S}[\![\,s\,]\!] \circ \mathbb{C}[\![\,c\,]\!])^n\,\Sigma\right)$$

Figure 2.5: Semantics of while loops

Note that this definition is not computable in general due to the unbounded number of iterations. In particular, let us consider the program in Listing 2.2. It starts from a non-negative number u and computes the $3n+1$ sequence until it reaches 1, so i computes what is called the total stopping time of the $3n+1$ sequence. As of today, proving that i is always finite is still an open problem [89].

Listing 2.2: Computing the total stopping time of the $3n+1$ sequence

```
1 int u = [0, +∞];
2 int i = 0;
3
4 while (u > 1) {
5   if (u % 2 == 0) { u = u / 2; }
6   else { u = 3 * u + 1; }
7   i = i + 1;
8 }
```

2.2.3 Comparing states

As we have mentioned before, a recurrent goal of static analysis is to prove program properties, such as the absence of bugs. Program properties can be seen as the set of program states verifying that property. In the case of our running example of Listing 2.1, we may want to prove that the division at line 6 is correct, i.e., that u is always even. If Σ_t is the set of program states reaching the if body at line 6, the property amounts to proving the following inclusion: $\Sigma_t \subseteq \Sigma_e$ where $\Sigma_e = \{\,\sigma \mid \sigma(u) \in 2\mathbb{Z}\,\}$. Of course, we can also prove a stronger property, such as u is 10 or 16 (i.e, $\Sigma_b = \{\,\sigma \mid \sigma(u) \in \{\,10, 16\,\}\,\}$). Since $\Sigma_t \subseteq \Sigma_b$, and $\Sigma_b \subseteq \Sigma_e$, we have by transitivity $\Sigma_t \subseteq \Sigma_e$. We notice that the set of program states, along with the inclusion relation, form a partially ordered set, which is useful to reason over the program properties mentioned before.

Definition 2.10 **Poset**

A partially ordered set (abbreviated poset) is a set X equipped with a relation $\sqsubset \in X \times X$ which is:

- Reflexive $\forall x \in X, x \sqsubset x$
- Anti-symmetric $\forall x, y \in X, x \sqsubset y \wedge y \sqsubset x \implies x = y$
- Transitive $\forall x, y, z \in X, x \sqsubset y \wedge y \sqsubset z \implies x \sqsubset z$

Example 2.11 **Poset**

Given a set X, $(\mathcal{P}(X), \subseteq)$ is a poset. In particular, the set of program states $\mathcal{P}(\mathcal{S})$ is a poset.

Due to Rice's theorem, the current program states are too expressive to prove program properties on them automatically. As a first simplification towards the obtention of a computable semantics, we track potential values for each variable instead of keeping sets of states – i.e., we move from $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$ to $\mathcal{V} \to \mathcal{P}(\mathbb{Z})$. We can remark that $\mathcal{V} \to \mathcal{P}(\mathbb{Z})$ is still a poset, thanks to the following pointwise lifting property:

Property 2.12 **Pointwise poset lifting**

Given a poset $(X, \sqsubset)$ and a set M, we define the pointwise relation $\dot{\sqsubset}$ such that:

$$\dot{\sqsubset} \in (M \to X) \times (M \to X)$$

$$f_1 \dot{\sqsubset} f_2 \Leftrightarrow \forall m \in M, f_1(m) \sqsubset f_2(m)$$

Then, $(M \to X, \dot{\sqsubset})$ is also a poset.

Remark 2.13 **Pointwise lifting notation**

As a convention, the pointwise lifting of an operator $\sqsubseteq$ is written $\dot{\sqsubseteq}$.

This simplification from $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$ to $\mathcal{V} \to \mathcal{P}(\mathbb{Z})$ can be seen as an abstraction – since we simplify the structure of the state – formally defined as the function α below. Given a set of states Σ, $\alpha(\Sigma)$ is the function mapping for each variable v its corresponding values in Σ.

$$\alpha : \left\{ \begin{array}{ccc} \mathcal{P}(\mathcal{V} \rightharpoonup \mathbb{Z}) & \to & \mathcal{V} \to \mathcal{P}(\mathbb{Z}) \\ \Sigma & \mapsto & \lambda v.\{\, \sigma(v) \mid \sigma \in \Sigma \,\} \end{array} \right.$$

It is also possible to define a converse operation, called a concretization, defined as the function γ. γ maps a function $f \in \mathcal{V} \to \mathcal{P}(\mathbb{Z})$ to a set of states Σ it represents, where each variable v of each state $\sigma \in \Sigma$ is a value of $f(v)$.

$$\gamma : \left\{ \begin{array}{ccc} \mathcal{V} \to \mathcal{P}(\mathbb{Z}) & \to & P(\mathcal{V} \rightharpoonup \mathbb{Z}) \\ f & \mapsto & \{\, \sigma \mid \forall v \in \mathcal{V}, \sigma(v) \in f(v) \,\} \end{array} \right.$$

Example 2.14 **Abstraction of $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$**

We define

$$\Sigma = \{\, (u \mapsto 10, i \mapsto 1)\,; (u \mapsto 16, i \mapsto 3) \,\}$$

The abstraction of Σ is

$$\alpha(\Sigma) = (u \mapsto \{\, 10, 16 \,\}, i \mapsto \{\, 1, 3 \,\})$$

If we concretize the abstracted set, we get

$$\gamma(\alpha(\Sigma)) = \{\, (u \mapsto 10, i \mapsto 1)\,; (u \mapsto 16, i \mapsto 3)\,; (u \mapsto 10, i \mapsto 3)\,; (u \mapsto 16, i \mapsto 1) \,\}$$

We can notice that Σ is strictly included in $\gamma(\alpha(\Sigma))$.

Remark 2.15 **Non-relational abstraction**

The abstraction α defined here breaks the relationality between variables. For example, the property $5i \leq u$ holds on the state Σ of Example 2.14, but it is not possible to prove it in the abstract.

Remark 2.16 **False alarms**

The case where a property holds in the concrete world but cannot be proved in the abstract due to imprecisions is called a false positive or false alarm. Using more precise abstractions may rule out some false alarms.

We can see that the order relation is preserved by the transformations α and γ: given any program state $\Sigma \in \mathcal{P}(\mathcal{V} \rightharpoonup \mathbb{Z})$ and a function $f \in \mathcal{V} \to \mathcal{P}(\mathbb{Z})$, the following equivalence holds: $\Sigma \subseteq \gamma(f) \Leftrightarrow \alpha(\Sigma) \dot{\sqsubseteq} f$. This property of the α and γ functions can be generalized to define the notion of Galois connection, initially introduced by Cousot and Cousot [33] in the founding paper of abstract interpretation.

Definition 2.17 **Galois connection**

Let $(A, \sqsubseteq)$ and $(C, \subseteq)$ be two posets. The pair $(\alpha : C \to A, \gamma : A \to C)$ is a Galois connection if:

$$\forall a \in A, \forall c \in C, c \subseteq \gamma(a) \Leftrightarrow \alpha(c) \sqsubseteq a$$

In that case, we write $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$. α is called the abstraction function, and γ the concretization function.

Remark 2.18 **Notation of abstract elements**

Abstract elements and abstract operators are usually suffixed with $^\#$ to ease distinguishing them from concrete elements.

Property 2.19 **Best abstraction**

Given a Galois connection $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, $\alpha(c)$ is the best abstraction of $c \in C$ (for $\subseteq$).

Proof. Let $c \in C$, and a be a sound abstraction of c. We have $c \subseteq \gamma(a)$. The definition of Galois connection entails $\alpha(c) \sqsubseteq a$. Thus $\alpha(c)$ is the best abstraction of c. □

Remark 2.20 **Concretization-only framework**

In some cases, there is no best abstraction. For example, the polyhedra abstract domain briefly presented in Section 2.4.4 abstracts subsets of the 2-dimensional plane $\mathbb{R}^2$. In particular, there is no best abstraction of a circle by a polyhedron (given that any polyhedron can be further extended to abstract the circle more precisely). In that case, a concretization $\gamma : C \to A$ is sufficient to reason about soundness (cf., Definition 2.21 below).

Definition 2.21 **Sound abstraction**

Let A be a set, $(C, \subseteq)$ a poset, and $\gamma \in A \to C$. We say that a is a sound abstraction of c when $c \subseteq \gamma(a)$.

Remark 2.22 **Proving safety properties in the abstract**

Let $\Sigma_e^\sharp \in \mathcal{V} \to \mathcal{P}(\mathbb{Z})$ and $\Sigma_e^\sharp(u) = 2\mathbb{Z}$. Let $\Sigma_e = \gamma(\Sigma_e^\sharp) = \{ (u \mapsto 2k) \mid k \in \mathbb{Z} \}$, where Σ_e is defined in Section 2.2.3. Assuming $f^\sharp \dot{\subseteq} \Sigma_e^\sharp$, and f is a sound abstraction of Σ, we get that $\Sigma \subseteq \gamma(f^\sharp) \subseteq \gamma(\Sigma_e^\sharp) = \Sigma_e$. Thus, for program properties that are exactly representable in the abstract ($\gamma(\Sigma_e^\sharp) = \Sigma_e$), if we prove that this property holds in the abstract ($f^\sharp \dot{\subseteq} \Sigma_e^\sharp$), it holds in the concrete too.

The notion of sound and best abstractions can also be defined for operators. These definitions will be helpful to prove that the semantics is sound.

Definition 2.23 **Sound operator abstraction**

Let A be a set, $(C, \subseteq)$ a poset, and $\gamma \in A \to C$. $f^\sharp \in A \to A$ is a sound abstraction of $f \in C \to C$ if

$$\forall a \in A, f(\gamma(a)) \subseteq \gamma(f^\sharp(a))$$

Property 2.24 **Best operator abstraction**

Given $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, and $f \in C \to C$, the best abstraction of f is $\alpha \circ f \circ \gamma$.

Proof. Let $f^\sharp$ be a sound abstraction of f. Due to the soundness of $f^\sharp$, $f \circ \gamma \dot{\subseteq} \gamma \circ f^\sharp$. By definition of the Galois connection, this is equivalent to $\alpha \circ f \circ \gamma \dot{\subseteq} f^\sharp$, which proves the property. □

2.3 Inferring ranges of `Imp` variables

Our program states $\mathcal{V} \to \mathcal{P}(\mathbb{Z})$ are still too expressive to be computable. One idea to further reduce the computational cost of the analysis is to abstract sets of integers by intervals.

2.3.1 The interval domain

We define the interval poset $(\mathcal{I}, \sqsubseteq_\mathcal{I})$ in Figure 2.6. The upper (resp. lower) bound of an interval can be plus (resp. minus) infinity, and the upper bound should be greater or equal to the lower bound. Intervals are extended with $\bot_\mathcal{I}$, representing the empty interval. The order $\sqsubseteq_\mathcal{I}$ denotes interval inclusion. A schematic representation of the interval poset is shown in Figure 2.7.

$$\begin{aligned}\mathcal{I} \overset{\text{def}}{=} & \{ [l, u] \mid l \in \mathbb{Z} \cup \{ -\infty \}, u \in \mathbb{Z} \cup \{ +\infty \}, l \leq u \} \cup \{ \bot \} \\ & \forall i \in \mathcal{I}, \bot_\mathcal{I} \sqsubseteq_\mathcal{I} i \\ & \forall ([l_1, u_1], [l_2, u_2]) \in (\mathcal{I} \setminus \{ \bot_\mathcal{I} \})^2, [l_1, u_1] \sqsubseteq_\mathcal{I} [l_2, u_2] \Leftrightarrow l_2 \leq l_1 \wedge u_1 \leq u_2\end{aligned}$$

Figure 2.6: The Interval Poset

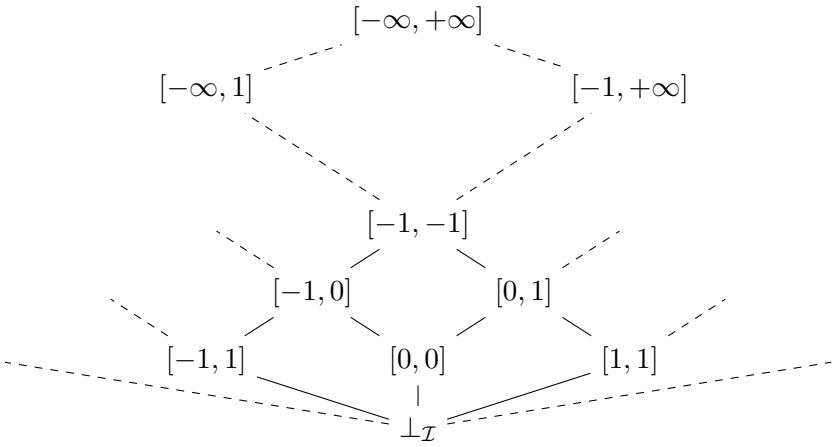

Figure 2.7: Hasse's diagram for the Interval Poset.
Plain arrows represent relationship through $\sqsubseteq_{\mathcal{I}}$.

Property 2.25 **The interval domain is a poset**

$(\mathcal{I}, \sqsubseteq_{\mathcal{I}})$ is a poset.

Proof.

- $\sqsubseteq_{\mathcal{I}}$ is reflexive: $\bot_{\mathcal{I}} \sqsubseteq_{\mathcal{I}} \bot_{\mathcal{I}}$, and $\forall [l, u] \in \mathcal{I} \setminus \{ \bot_{\mathcal{I}} \}, [l, u] \sqsubseteq_{\mathcal{I}} [l, u]$.
- $\sqsubseteq_{\mathcal{I}}$ is antisymmetric: let $i_1, i_2 \in \mathcal{I}$ such that $i_1 \sqsubseteq i_2 \wedge i_2 \sqsubseteq i_1$. We show that $i_1 = i_2$.:
  - If $i_1 = \bot_{\mathcal{I}}$, we have $i_2 \sqsubseteq_{\mathcal{I}} \bot_{\mathcal{I}}$. By definition of $\sqsubseteq_{\mathcal{I}}$, $i_2 = \bot_{\mathcal{I}}$.
  - Otherwise, without loss of generality, let $[l_1, u_1] = i_1$ and $[l_2, u_2] = i_2$. We have $l_2 \leq l_1 \wedge u_1 \leq u_2$ and $l_1 \leq l_2 \wedge u_2 \leq u_1$. Thus $i_1 = i_2$ too.
- $\sqsubseteq_{\mathcal{I}}$ is transitive: let $i_1, i_2, i_3 \in \mathcal{I}$ such that $i_1 \sqsubseteq i_2 \wedge i_2 \sqsubseteq i_3$. We show that $i_1 \sqsubseteq_{\mathcal{I}} i_3$.
  - If $i_1 = \bot_{\mathcal{I}}$, the property is straightforward.
  - If $i_2 = \bot_{\mathcal{I}}$, we have $i_1 = \bot_{\mathcal{I}}$ (since $i_1 \sqsubseteq i_2$), and the property holds.
  - Similarly, $i_3 = \bot_{\mathcal{I}}$ entails $i_2 = \bot_{\mathcal{I}}$ entails $i_1 = \bot_{\mathcal{I}}$.
  - We can thus assume that $i_1 = [l_1, u_1]; i_2 = [l_2, u_2]; i_3 = [l_3, u_3]$. By definition of $\sqsubseteq_{\mathcal{I}}$, $l_2 \leq l_1 \wedge u_1 \leq u_2$ and $l_3 \leq l_1 \wedge u_2 \leq u_3$. By transitivity of $\leq$, we have $l_3 \leq l_1$ and $u_1 \leq u_3$. Thus, $i_1 \sqsubseteq_{\mathcal{I}} i_3$

□

2.3.2 Concretization

Similarly to what we have done previously, we can define how to abstract a set of integers into an interval and concretize an interval into a set of integers.

Property 2.26 **Galois connection for intervals**

These functions define a Galois connection $(\mathcal{P}(\mathbb{Z}), \subseteq) \underset{\alpha_{\mathcal{I}}}{\overset{\gamma_{\mathcal{I}}}{\leftrightarrows}} (\mathcal{I}, \sqsubseteq_{\mathcal{I}})$.

$$\alpha_{\mathcal{I}} : \begin{cases} \mathcal{P}(\mathbb{Z}) & \to & \mathcal{I} \\ \emptyset & \mapsto & \bot \\ S & \mapsto & [\min S, \max S] \end{cases} \qquad \gamma_{\mathcal{I}} : \begin{cases} \mathcal{I} & \to & \mathcal{P}(\mathbb{Z}) \\ \bot & \mapsto & \emptyset \\ [l,u] & \mapsto & \{\, z \in \mathbb{Z} \mid l \leq z \leq u \,\} \end{cases}$$

Proof. Let $i \in \mathcal{I}$, $X \in \mathcal{P}(\mathbb{Z})$.

- If $i = \bot$, then $\gamma_{\mathcal{I}}(\bot_{\mathcal{I}}) = \emptyset$ and $c \subseteq \emptyset \Leftrightarrow \alpha(c) \sqsubseteq_{\mathcal{I}} \bot_{\mathcal{I}}$ is trivial to prove.
- If $i = [l, u]$:

$$\begin{aligned} & X \subseteq \gamma_{\mathcal{I}}([l,u]) \\ \Leftrightarrow\ & \forall x \in X, x \in \gamma_{\mathcal{I}}([l,u]) \\ \Leftrightarrow\ & \forall x \in X, l \leq x \leq u \\ \Leftrightarrow\ & l \leq \min(x) \wedge \max(x) \leq u \\ \Leftrightarrow\ & [\min(x), \max(x)] \sqsubseteq_{\mathcal{I}} [l,u] \\ \Leftrightarrow\ & \alpha(X) \sqsubseteq_{\mathcal{I}} [l,u] \end{aligned}$$

□

2.3.3 Interval transfer functions

Let $[l_x, u_x], [l_y, u_y]$ be two non-empty intervals. Thanks to Property 2.24, we can derive the best abstraction for the addition operator on sets, i.e we search for $+_{\mathcal{I}}$ such that:

$$[l_x, u_x] +_{\mathcal{I}} [l_y, u_y] = \alpha_{\mathcal{I}}(\gamma_I([l_x, u_x]) + \gamma_I([l_y, u_y]))$$

Given that:

$$\begin{aligned} & \alpha_{\mathcal{I}}(\gamma_I([l_x, u_x]) + \gamma_I([l_y, u_y])) \\ =& \alpha_{\mathcal{I}}(\{\, x + y \mid l_x \leq x \leq u_x, l_y \leq y \leq u_y \,\}) \\ =& [l_x + l_y, u_x + u_y] \end{aligned}$$

We define the interval addition such that:

$$\forall i \in \mathcal{I}, \bot_{\mathcal{I}} +_{\mathcal{I}} i = i +_{\mathcal{I}} \bot_{\mathcal{I}} = i \qquad [l_x, u_x] +_{\mathcal{I}} [l_y, u_y] = [l_x + l_y, u_x + u_y]$$

This is the best abstraction of the set addition on the intervals. We can similarly derive subtraction, multiplication, division and remainder operators on intervals.

2.3.4 Abstract semantics of expressions

For now, we have only abstracted sets of integers into intervals. In order to abstract sets of program states over several variables, our abstract domain is a non-relational, pointwise lifting of intervals. In order to avoid multiple representations of the empty set[1], we use a coalescent point-wise lifting to intervals, written $\mathcal{V} \dot{\to} \mathcal{I}$.

[1] Given $f \in \mathcal{V} \to \mathcal{I}$, if there exists $v \in \mathcal{V}$ such that $f(v) = \bot_{\mathcal{I}}$, f represents the empty set in the concrete.

Definition 2.27 **Coalescent point-wise lifting**

Let $(X, \sqsubseteq)$ be a poset with minimal element $\bot_X$. We define the coalescent point-wise lifting of X, written $\mathcal{V} \dot{\to} X$, as:

$$\mathcal{V} \dot{\to} X \stackrel{\text{def}}{=} (\mathcal{V} \to (X \setminus \bot_X)) \cup \bot$$

Let $f, g \in \mathcal{V} \dot{\to} X$. We define $f \dot{\sqsubseteq} g \stackrel{\text{def}}{=} f = \bot \vee (f \neq \bot \wedge g \neq \bot \wedge \forall v \in V, f(v) \sqsubseteq g(v))$.

$(\mathcal{V} \dot{\to} X, \dot{\sqsubseteq})$ is also a poset.

Similarly to the pointwise lifting property of posets, we can establish the same property for Galois connections. This property also holds for the coalescent point-wise lifting.

Property 2.28 **Pointwise Galois connection lifting**

Let $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, and M be a set. $\dot{\subseteq}$ and $\dot{\sqsubseteq}$ are the lifted partial order operator as defined in Property 2.12, and $\dot{\delta}(f) = \lambda m.\delta(f(m))$ for $\delta \in \{\alpha, \gamma\}$. Then $(M \to C, \dot{\subseteq}), \underset{\dot{\alpha}}{\overset{\dot{\gamma}}{\leftrightarrows}} (M \to A, \dot{\sqsubseteq})$.

Instead of deriving the best abstraction of the semantics of expressions using $\alpha_{\mathcal{I}}$, we use a different approach. We first define the abstract semantics of expressions in Figure 2.8 and show it is sound. The evaluation of a variable looks up in the abstract state (except if the state is $\bot$). The evaluation of constants returns constants. The evaluation of binary expression consists in recursively evaluating both the left-hand and right-hand sides and combining them using the binary operators defined on intervals.

$$\begin{aligned}
&\mathbb{E}^\sharp[\![\, e \,]\!] \in (\mathcal{V} \dot{\to} \mathcal{I}) \to \mathcal{I} \\
&\mathbb{E}^\sharp[\![\, v \in \mathcal{V} \,]\!]\sigma^\sharp \stackrel{\text{def}}{=} \text{if } \sigma^\sharp = \bot \text{ then } \bot_{\mathcal{I}} \text{ else } \sigma^\sharp(v) \\
&\mathbb{E}^\sharp[\![\, z \in \mathbb{Z} \,]\!]\sigma^\sharp \stackrel{\text{def}}{=} [z, z] \\
&\mathbb{E}^\sharp[\![\, [z_1, z_2] \,]\!]\sigma^\sharp \stackrel{\text{def}}{=} [z_1, z_2] \\
&\mathbb{E}^\sharp[\![\, e_1 + e_2 \,]\!] \stackrel{\text{def}}{=} \mathbb{E}^\sharp[\![\, e_1 \,]\!] \dot{+}_{\mathcal{I}} \mathbb{E}^\sharp[\![\, e_2 \,]\!] \\
&\mathbb{E}^\sharp[\![\, e_1 - e_2 \,]\!] \stackrel{\text{def}}{=} \mathbb{E}^\sharp[\![\, e_1 \,]\!] \dot{-}_{\mathcal{I}} \mathbb{E}^\sharp[\![\, e_2 \,]\!] \\
&\mathbb{E}^\sharp[\![\, e_1 * e_2 \,]\!] \stackrel{\text{def}}{=} \mathbb{E}^\sharp[\![\, e_1 \,]\!] \dot{*}_{\mathcal{I}} \mathbb{E}^\sharp[\![\, e_2 \,]\!] \\
&\mathbb{E}^\sharp[\![\, e_1 \,/\, e_2 \,]\!] \stackrel{\text{def}}{=} \mathbb{E}^\sharp[\![\, e_1 \,]\!] \dot{/}_{\mathcal{I}} \mathbb{E}^\sharp[\![\, e_2 \,]\!] \\
&\mathbb{E}^\sharp[\![\, e_1 \,\%\, e_2 \,]\!] \stackrel{\text{def}}{=} \mathbb{E}^\sharp[\![\, e_1 \,]\!] \dot{\%}_{\mathcal{I}} \mathbb{E}^\sharp[\![\, e_2 \,]\!]
\end{aligned}$$

Figure 2.8: Abstract semantics of Expressions

Property 2.29 $\mathbb{E}^\sharp[\![\, \cdot \,]\!]$ **is a sound abstraction of** $\mathbb{E}[\![\, \cdot \,]\!]$

$$\forall \sigma \in (\mathcal{V} \to \mathcal{P}(\mathbb{Z})), \forall \sigma^\sharp \in (\mathcal{V} \dot{\to} \mathcal{I}), \sigma \dot{\sqsubseteq}_{\mathcal{I}} \dot{\gamma}_{\mathcal{I}}(\sigma^\sharp) \implies \forall e, \mathbb{E}[\![\, e \,]\!]\sigma \sqsubseteq_{\mathcal{I}} \gamma_{\mathcal{I}}(\mathbb{E}^\sharp[\![\, e \,]\!]\sigma^\sharp)$$

Proof. Let $\sigma, \sigma^\sharp$ such that $\sigma \dot{\sqsubseteq}_{\mathcal{I}} \dot{\gamma}_{\mathcal{I}}(\sigma^\sharp)$. The case where $\sigma^\sharp = \bot$ is straightforward. We prove by structural induction on e that $\mathbb{E}[\![\, e \,]\!]\sigma \subseteq \gamma_{\mathcal{I}}(\mathbb{E}^\sharp[\![\, e \,]\!]\sigma^\sharp)$:

- Case $e = v \in \mathcal{V}$: $\mathbb{E}[\![v]\!]\sigma = \sigma(v) \subseteq \dot{\gamma}_{\mathcal{I}}(\sigma^\sharp)(v) = \gamma_{\mathcal{I}}(\sigma^\sharp(v)) = \gamma_{\mathcal{I}}(\mathbb{E}^\#[\![v]\!]\sigma^\sharp)$.
- Case $e = [z_1, z_2]$: $\mathbb{E}[\![[z_1, z_2]]\!]\sigma = \{ z \in \mathbb{Z} \mid z_1 \leq z \leq z_2 \} = \gamma_{\mathcal{I}}([z_1, z_2]) = \gamma_{\mathcal{I}}(\mathbb{E}^\#[\![[z_1, z_2]]\!]\sigma)$.
- Case $e = z \in \mathbb{Z}$. Subcase of $e = [z, z]$.
- Case $e = e_1 \dagger e_2$, with $\dagger \in \{ +, -, *, /, \% \}$. By induction hypothesis (I.H.), we have that $\mathbb{E}[\![e_i]\!]\sigma \subseteq \gamma_{\mathcal{I}}(\mathbb{E}^\#[\![e_i]\!]\sigma^\sharp)$, given $i \in \{ 1, 2 \}$.

$$
\begin{aligned}
\gamma_{\mathcal{I}}(\mathbb{E}^\#[\![e_1 \dagger e_2]\!]\sigma^\sharp) &= \gamma_{\mathcal{I}}(\mathbb{E}^\#[\![e_1]\!]\sigma^\sharp \dagger_{\mathcal{I}} \mathbb{E}^\#[\![e_2]\!]\sigma^\sharp) && \text{(def. of } \mathbb{E}^\#[\![e_1 \dagger e_2)]\!] \\
&\supseteq \gamma_{\mathcal{I}}(\mathbb{E}^\#[\![e_1]\!]\sigma^\sharp) \dagger \gamma_{\mathcal{I}}(\mathbb{E}^\#[\![e_2]\!]\sigma^\sharp) && \text{(soundness of } \dagger_{\mathcal{I}}) \\
&\supseteq \mathbb{E}[\![e_1]\!]\sigma \dagger \mathbb{E}[\![e_2]\!]\sigma && \text{(I.H. \& } \dagger \text{ monotonic)} \\
&= \{ v_1 \dagger v_2 \mid v_1 \in \mathbb{E}[\![e_1]\!]\sigma, v_2 \in \mathbb{E}[\![e_2]\!]\sigma \} && \text{(def. of } \dagger \text{ on } \mathcal{P}(\mathbb{Z})) \\
&= \mathbb{E}[\![e_1 \dagger e_2]\!]\sigma && \text{(def. of } \mathbb{E}[\![e_1 \dagger e_2]\!])
\end{aligned}
$$

  This inclusion still holds for the division and remainder operators, even if an abstract state may contain 0: both the abstract and the concrete operations will prune out these cases.

□

Remark 2.30 **Imprecisions with intervals**

Given $\sigma = \{ (u \mapsto 3, i \mapsto 2); (u \mapsto 5, i \mapsto 0) \}$, $\mathbb{E}[\![u * i]\!]\sigma$ yields $\{ 6; 0 \}$. In the abstract, $\sigma^\sharp = (u \mapsto [3, 5], i \mapsto [0, 2])$, so $\mathbb{E}^\#[\![u * i]\!]\sigma^\sharp$ yields $[0, 10]$. The lower bound of the interval is tight, but the upper bound is not. This shows that the interval analysis is imprecise in this case.

2.3.5 Abstract semantics of statements

The abstract semantics of a statement s has signature $\mathbb{S}^\#[\![s]\!] : (\mathcal{V} \dot{\to} \mathcal{I}) \to (\mathcal{V} \dot{\to} \mathcal{I})$. The case of variable declarations, assignments, and sequence of statements are straightforward.

2.3.6 Basic statements

$$
\begin{aligned}
&\mathbb{S}^\#[\![\texttt{int}\ x]\!]\sigma^\sharp \stackrel{\text{def}}{=} \sigma^\sharp[x \mapsto [-\infty; +\infty]] \\
&\mathbb{S}^\#[\![x = e]\!]\sigma^\sharp \stackrel{\text{def}}{=} \text{let } v = \mathbb{E}^\#[\![e]\!]\sigma \text{ in if } v = \bot_{\mathcal{I}} \text{ then } \bot \text{ else } \sigma^\sharp[x \mapsto v] \\
&\mathbb{S}^\#[\![s_1; s_2]\!] \stackrel{\text{def}}{=} \mathbb{S}^\#[\![s_2]\!] \circ \mathbb{S}^\#[\![s_1]\!]
\end{aligned}
$$

Figure 2.9: Abstract semantics of basic statements

2.3.7 Conditionals

The semantics of conditionals is more complex. The concrete operator performed the union of the set of states reached after the execution of each branch. We thus need to define an operator equivalent to the union, called least upper bound or join.

Definition 2.31 **Least upper bound**

Let $(X, \sqsubseteq)$ be a poset, and $a, b \in X$. Any $c \in X$ such that $a \sqsubseteq c$ and $b \sqsubseteq c$ is called an upper bound of a and b.

We say that c is the least upper bound of a and b, if for any c' upper bound of a and b,

$c \sqsubseteq c'$.

The least upper bound is abbreviated as lub, or denoted as join operator, and is usually written $a \sqcup b$.

Remark 2.32 **Unicity of the least upper bound**

The least upper bound may not exist but is unique if it does.

Remark 2.33 **Greatest lower bound**

We can define a converse notion of greatest lower bound, or meet, written $a \sqcap b$.

We can derive the best abstraction of the union operator, such that:

$$[a,b] \sqcup_{\mathcal{I}} [c,d] = \alpha_{\mathcal{I}}(\gamma_{\mathcal{I}}([a,b]) \cup \gamma_{\mathcal{I}}([c,d]))$$

$$\begin{aligned}
&\alpha_{\mathcal{I}}(\gamma_{\mathcal{I}}([a,b]) \cup \gamma_{\mathcal{I}}([c,d])) \\
=&\alpha_{\mathcal{I}}(\{\, x \in \mathbb{Z} \mid a \leq x \leq b \,\} \cup \{\, x \in \mathbb{Z} \mid c \leq x \leq d \,\}) \\
=&\alpha_{\mathcal{I}}(\{\, x \in \mathbb{Z} \mid a \leq x \leq b \wedge c \leq x \leq d \,\}) \\
=&[\min(a,c), \max(b,d)]
\end{aligned}$$

Thus, we define $[a,b] \sqcup_{\mathcal{I}} [c,d] \stackrel{\text{def}}{=} [\min(a,c), \max(b,d)]$. Similarly, we define $[a,b] \sqcap_{\mathcal{I}} [c,d] \stackrel{\text{def}}{=} [\max(a,c), \min(b,d)]$ (or $\bot_{\mathcal{I}}$ if $\max(a,c) > \min(b,d)$). Both definitions are trivially extended if an interval is $\bot_{\mathcal{I}}$.

Posets structured with join and meet operators are also common in quite a few areas of computer science. They form the basis for the definition of a complete lattice.

Definition 2.34 **Complete lattice**

A complete lattice, $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a poset $(X, \sqsubseteq)$ where:

- $\forall A \subseteq X, \sqcup A$ exists,
- $\forall A \subseteq X, \sqcap A$ exists.

In particular, $\bot = \sqcup\emptyset$ is the least element and $\top = \sqcup X$ is the greatest element.

Property 2.35 **Complete lattice – parts of a set**

Given a set X, $(\mathcal{P}(X), \sqsubset, \cup, \cap, \emptyset, X)$ is a complete lattice.

Property 2.36 **The interval domain is a complete lattice**

$(\mathcal{I}, \sqsubseteq_{\mathcal{I}}, \sqcup_{\mathcal{I}}, \sqcap_{\mathcal{I}}, \bot_{\mathcal{I}}, \top_{\mathcal{I}})$ is a complete lattice, where $\top_{\mathcal{I}} = [-\infty, +\infty]$.

Remark 2.37 **Pointwise lifting of complete lattices**

Similarly to the poset and Galois connection pointwise liftings, complete lattices can also be pointwise lifted.

Thus, the semantics of conditional statements is:

$$\mathbb{S}^\#[\![\,\texttt{if}\ (c)\ \{s_t\}\ \texttt{else}\ \{s_f\}\,]\!] \overset{\text{def}}{=} \mathbb{S}^\#[\![\,s_t\,]\!] \circ \mathbb{C}^\#[\![\,c\,]\!] \dot{\sqcup} \mathbb{S}^\#[\![\,s_f\,]\!] \circ \mathbb{C}^\#[\![\,\neg c\,]\!]$$

We do not define the abstract conditional filtering operator $\mathbb{C}^\#[\![\,c\,]\!]$. Precise versions rely on bottom-up and top-down traversals of the abstract syntax tree of the condition c, called the HC4 algorithm in the constraint solving community [10]. In the setting of abstract interpretation, these traversals are described by Miné [106, Section 4.6].

2.3.8 Terminating loop analyses

The current definition of the semantics of `while` loops – reminded below – is not computable due to the unbounded number of iterations.

$$\mathbb{S}[\![\,\texttt{while}\ (c)\ \{s\}\,]\!]\Sigma \overset{\text{def}}{=} \mathbb{C}[\![\,\neg c\,]\!]\left(\bigcup_{n\in\mathbb{N}} (\mathbb{S}[\![\,s\,]\!] \circ \mathbb{C}[\![\,c\,]\!])^n\,\Sigma\right)$$

We search for an alternative semantics of loops that exhibits the underlying notion of loop invariant. A typical and widely used concept that applies here is searching for a fixpoint of an operator $f : C \to C$, i.e for $x \in C$ such that $f(x) = x$. If f models the effects of the loop body, a fixpoint of f will be a loop invariant. In particular, the least fixpoint of f, written $\mathrm{lfp}\, f$ will be the strongest loop invariant (intuitively, the smallest state has no extra information and is thus the strongest property). We start by introducing concepts necessary to state Kleene's fixpoint theorem. We use Kleene's theorem to establish an equivalent definition of the semantics of `while` using a least fixpoint. We then define methods to perform computable fixpoint approximations.

Definition 2.38 **Chain**

Let $(X, \sqsubseteq)$ be a poset and $C \subseteq X$. C is a chain if it is totally ordered, i.e: $\forall c, d \in C, c \sqsubseteq d \vee d \sqsubseteq c$

Example 2.39 **Chain**

$C = \cup_{n\in\mathbb{N}}\{\,[0, n]\,\}$ is a chain of intervals, since each one is included in the next one. $D = \cup_{n\in\mathbb{N}}\{\,[2n, 2n+1]\,\}$ is not a chain, since all intervals are disjoint (thus not included into one another).

Definition 2.40 **Complete partial order**

A complete partial order (CPO), is a poset $(X, \sqsubseteq)$ such that every chain has a least upper bound.

Property 2.41 **Lattices and partial orders**

Complete lattices are complete partial orders.

Definition 2.42 **Continuous operator**

Let $(A, \sqsubseteq_A, \sqcup_A)$ and $(B, \sqsubseteq_B, \sqcup_B)$ be two complete partial orders. $f : A \to B$ is continuous, if for every chain $C \in \mathcal{P}(A)$, $f(C) = \{\, f(c) \mid c \in C \,\}$ is also a chain and $f(\sqcup_A C) = \sqcup_B\{\, f(c) \mid c \in C \,\}$

Remark 2.43 **Join-morphism and continuity**

In particular, let $f : A \to B$ be a complete join-morphism, i.e $\forall X \subset A, f(\sqcup_A X) = \sqcup_B f(X)$. Then, f is continuous.

Theorem 2.44 **Kleene's fixpoint theorem**

Let $(X, \sqsubseteq, \sqcup, \bot)$ be a complete partial order, and $f : X \to X$ a continuous operator. lfp f exists and lfp $f = \sqcup\{ f^i(\bot) \mid i \in \mathbb{N} \}$

Now, we use Kleene's fixpoint theorem to define the semantics of while programs using a least fixpoint.

Property 2.45 **Rewriting the semantics of loops into a fixpoint**

Let $\Sigma \in \mathcal{P}(\mathcal{S})$, s a statement and c a condition.

$$\bigcup_{n\in\mathbb{N}} (\mathbb{S}[\![s]\!] \circ \mathbb{C}[\![c]\!])^n \Sigma = \text{lfp } f, \text{ with } f(X) \stackrel{\text{def}}{=} \Sigma \cup \mathbb{S}[\![s]\!] \circ \mathbb{C}[\![c]\!] X$$

Proof. We can apply Kleene's fixpoint theorem since:

- $(\mathcal{P}(\mathcal{S}), \subseteq, \cup, \top, \bot)$ is a complete partial order.
- f is continuous: the concrete semantics is a complete join-morphism.

Thus, we know that

$$\text{lfp } f = \bigcup_{n\in\mathbb{N}} f^n(\emptyset)$$

By induction over the integers, we can prove that $\forall n \in \mathbb{N}, f^n(\emptyset) = \bigcup_{k\leq n} (\mathbb{S}[\![s]\!] \circ \mathbb{C}[\![c]\!])^k \Sigma$:

- Case $n = 0$, $f^0(\emptyset) = \Sigma \cup \mathbb{S}[\![s]\!] \circ \mathbb{C}[\![c]\!] \emptyset = \Sigma$
- Assuming the property holds for a given n, we have:

$$\begin{aligned} f^{n+1}(\emptyset) =& \Sigma \cup (\mathbb{S}[\![s]\!] \circ \mathbb{C}[\![c]\!])(f^n(\emptyset)) \\ =& \Sigma \cup (\mathbb{S}[\![s]\!] \circ \mathbb{C}[\![c]\!])(\bigcup_{k\leq n} (\mathbb{S}[\![s]\!] \circ \mathbb{C}[\![c]\!])^k \Sigma) \\ =& \Sigma \cup \bigcup_{k\leq n} (\mathbb{S}[\![s]\!] \circ \mathbb{C}[\![c]\!])^{k+1} \Sigma \\ =& \bigcup_{k\leq n+1} (\mathbb{S}[\![s]\!] \circ \mathbb{C}[\![c]\!])^k \Sigma \end{aligned}$$

Hence

$$\text{lfp } f = \bigcup_{n\in\mathbb{N}} f^n(\emptyset) = \bigcup_{n\in\mathbb{N}} \bigcup_{k\leq n} (\mathbb{S}[\![s]\!] \circ \mathbb{C}[\![c]\!])^k \Sigma = \bigcup_{n\in\mathbb{N}} (\mathbb{S}[\![s]\!] \circ \mathbb{C}[\![c]\!])^n \Sigma$$

□

From our property, we can thus define an alternative characterization of the semantics of

while loops.

$$\mathbb{S}[\![\,\texttt{while}\ (c)\ \{s\}\,]\!]\Sigma = \mathbb{C}[\![\,\neg c\,]\!]\,\text{lfp}\,f \text{ with } f(X) \overset{\text{def}}{=} \Sigma \cup \mathbb{S}[\![\,s\,]\!] \circ \mathbb{C}[\![\,c\,]\!]X$$

We now have to approximate the semantics of while loops in the abstract, using widening operators to make it computable.

Definition 2.46 **Widening operator**

Let $(A, \sqsubseteq)$ be a poset. $\nabla : A \times A \to A$ is a widening operator if it:

- Computes an upper bound $\forall x, y \in A, x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$
- Ensures convergence: for any sequence $(y_i)_{i\in\mathbb{N}}$ of elements of A, and given the sequence $(x_i)_{i\in\mathbb{N}}$ defined as $x_0 \overset{\text{def}}{=} y_0$, $x_{n+1} \overset{\text{def}}{=} x_n \nabla y_{n+1}$, we have: $\exists k \in \mathbb{N}, x_{k+1} = x_k$

Theorem 2.47 **Fixpoint approximation**

Let $(C, \subseteq, \cup, \cap, \bot, \top)$ be a complete lattice, and $f : C \to C$ a monotonic function. Let $(A, \sqsubseteq)$ be a poset with a minimal element $\bot$, $f^\sharp : A \to A$ an abstraction of f and ∇ a widening operator. The sequence $x_0 \overset{\text{def}}{=} \bot$, $x_{i+1} \overset{\text{def}}{=} x_i \nabla f^\sharp(x_i)$:

- converges in finite time, and
- its limit (written x_l) is a sound abstraction of lfp f, i.e, lfp $f \subseteq \gamma(x_l)$

Now we can define a widening operator on intervals. As usual, this operator is then lifted to program states.

Property 2.48 **Widening operator on intervals**

The $\nabla_{\mathcal{I}}$ operator defined below is a widening.

$$[a,b]\nabla_{\mathcal{I}}\bot_I = \bot_I\nabla_{\mathcal{I}}[a,b] = [a,b]$$
$$[a,b]\nabla_{\mathcal{I}}[c,d] = \left[\begin{cases} a & \text{if } a \le c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \ge d \\ +\infty & \text{otherwise} \end{cases}\right]$$

Thanks to Theorem 2.47, we can thus define a sound, computable, abstract semantics of intervals for loops:

$$\mathbb{S}^\#[\![\,\texttt{while}\ (c)\ \{s\}\,]\!]\sigma^\sharp = \mathbb{C}^\#[\![\,\neg c\,]\!](\lim \delta^n(\dot{\bot}_{\mathcal{I}})) \text{ with } \delta(x^\sharp) = x^\sharp \nabla (\sigma^\sharp \sqcup \mathbb{S}^\#[\![\,s\,]\!] \circ \mathbb{C}^\#[\![\,c\,]\!]x^\sharp)$$

Example 2.49 **Fixpoint computation**

We show the iterations computed when analyzing the program of Listing 2.1, starting with $u = [1, 3]$ and $n = 100$. We start from $\sigma^\sharp = (u \mapsto [1, 3], i \mapsto [0, 0])$ (the value of n is constant, so we omit it in the states).

$$\begin{aligned}
\delta^1(\dot{\bot}_{\mathcal{I}}) &= \dot{\bot}_{\mathcal{I}} \dot{\nabla}_{\mathcal{I}} (\sigma^\sharp \dot{\sqcup}^\sharp_{\mathcal{I}} \mathbb{S}^\sharp[\![c]\!] \circ \mathbb{C}^\sharp[\![c]\!] \dot{\bot}_{\mathcal{I}}) \\
&= \sigma^\sharp \dot{\sqcup}^\sharp_{\mathcal{I}} \dot{\bot}_{\mathcal{I}} = \sigma^\sharp \\
\delta^2(\dot{\bot}_{\mathcal{I}}) &= \sigma^\sharp \dot{\nabla}_{\mathcal{I}} (\sigma^\sharp \dot{\sqcup}^\sharp_{\mathcal{I}} (u \mapsto [1,10], i \mapsto [1,1]) \\
&= \sigma^\sharp \dot{\nabla}_{\mathcal{I}} (u \mapsto [1,10], i \mapsto [0,1]) \\
&= (u \mapsto [1,+\infty], i \mapsto [0,+\infty]) \\
\delta^{n+2}(\dot{\bot}_{\mathcal{I}}) &= \delta^2(\dot{\bot}_{\mathcal{I}}), \forall n \in \mathbb{N}
\end{aligned}$$

Thus, $\lim \delta^n(\dot{\bot}_{\mathcal{I}}) = \delta^2(\dot{\bot}_{\mathcal{I}})$, and the state after the while loop is

$$\mathbb{C}^\sharp[\![i < 100]\!] \delta^2(\dot{\bot}_{\mathcal{I}}) = (u \mapsto [1,+\infty], i \mapsto [100,+\infty])$$

Decreasing iteration. The result of the range analysis performed in the previous example is a bit disappointing, as we are not even able to infer that $i = 100$ at the end of the loop. Thankfully, there is an easy way to recover some precision. Using the notations of Theorem 2.47, we show that $f^\sharp(x_l)$ is a sound approximation of lfp f which is better than x_l.

- We prove that $f^\sharp(x_l)$ is a sound approximation of lfp:

$$\begin{aligned}
\text{lfp } f &= f(\text{lfp } f) && (\text{lfp } f \text{ is a fixpoint of } f) \\
&\subseteq f(\gamma(x_l)) && (\text{lfp } f \subseteq \gamma(x_l)) \\
&\subseteq \gamma(f^\sharp(x_l)) && (f^\sharp \text{ sound abstraction of } f)
\end{aligned}$$

- We notice that $f^\sharp(x_l) \sqsubseteq x_l$. Since x_l is the limit of the sequence $(x_i)_{i \in \mathbb{N}}$, we have $x_l = x_l \nabla f^\sharp(x_l)$. The widening ∇ computes an upper bound, hence $f^\sharp(x_l) \sqsubseteq x_l$.

From an implementation perspective, we usually have to compute $f^\sharp(x_l)$ to check that x_l is the limit, so we get the decreasing iteration for free.

Example 2.50 **Decreasing iteration on the previous example**

Applying a decreasing iteration on $\delta^2(\dot{\bot}_{\mathcal{I}})$ yields:

$$\begin{aligned}
&\sigma^\sharp \dot{\sqcup}^\sharp_{\mathcal{I}} \mathbb{S}^\sharp[\![s]\!] \circ \mathbb{C}^\sharp[\![c]\!] \delta^2(\dot{\bot}_{\mathcal{I}}) \\
=&\sigma^\sharp \dot{\sqcup}^\sharp_{\mathcal{I}} \mathbb{S}^\sharp[\![s]\!]((u \mapsto [1,+\infty], i \mapsto [0,99])) \\
=&\sigma^\sharp \dot{\sqcup}^\sharp_{\mathcal{I}} (u \mapsto [1,+\infty], i \mapsto [100,100]) \\
=&(u \mapsto [1,+\infty], i \mapsto [0,100])
\end{aligned}$$

The state obtained after the loop is thus

$$\mathbb{C}^\sharp[\![i < 100]\!] (u \mapsto [1,+\infty], i \mapsto [0,100]) = (u \mapsto [1,+\infty], i \mapsto [100,100])$$

2.3.9 Improving the analysis with congruences

If we go back to our running example program, the interval domain cannot prove that the division by two will always result in an integer. In an analyzer, this would cause spurious false alarms: we know the division will be correct, but our tool cannot prove it. We introduce a new abstract domain tracking integer congruences [65] to tackle this issue. Congruences are defined as $\mathcal{C} \stackrel{\text{def}}{=} \{ a\mathbb{Z} + b \mid a \in \mathbb{N}, 0 \leq b < a \} \cup \{ \bot \}$. The abstract value $a\mathbb{Z} + b$ represents the set of integers congruent to b modulo a (defined in the concretization Figure 2.10).

$$\gamma_{\mathcal{C}} : \begin{cases} \mathcal{C} & \to & \mathcal{P}(\mathbb{Z}) \\ \bot & \mapsto & \emptyset \\ a\mathbb{Z}+b & \mapsto & \{\, ak+b \mid k \in \mathbb{Z} \,\} \end{cases}$$

Figure 2.10: Concretization of congruences

We do not delve into the details of the definition of the abstract operators ($+^\sharp, -^\sharp, *^\sharp, /^\sharp, \%^\sharp, \sqcup^\sharp, \sqcap^\sharp$). An abstraction function $\alpha_{\mathcal{C}}$ exists and defines a Galois connection with $\gamma_{\mathcal{C}}$.

Using this domain, we are able to prove that starting from $\sigma^\sharp = u \mapsto 1\mathbb{Z}+0$, and assuming `u % 2 == 0`, we have $\sigma^\sharp = u \mapsto 2\mathbb{Z}+0$. We can thus infer that the division by two afterward will be correct.

2.3.9.1 Deriving the semantics

The congruences abstract values can be lifted to a non-relational domain $\mathcal{V} \to \mathcal{C}$ similarly to the lifting performed for intervals. Instead of performing a lifting specific to the congruences, we define a unified framework lifting non-relational domains. This framework applies straightforwardly to the case of congruences. We define a notion of value abstract domain (Definition 2.51), to which intervals and congruences belong. The value abstract domain is lifted into an abstract semantics in Property 2.52.

Definition 2.51 **Value abstract domain**

A value abstract domain consists in:

- a poset $(\mathcal{D}^\sharp, \sqsubseteq^\sharp)$,
- a smallest element $\bot$ and a largest element $\top$
- sound abstractions of:
  - constants and intervals (written $c^\sharp$, $[l,u]^\sharp$)
  - binary operators, written $+^\sharp, -^\sharp, *^\sharp, /^\sharp, \%^\sharp$,
  - set union and intersection $\sqcup^\sharp, \sqcap^\sharp$
- a widening operator ∇
- a concretization operator $\gamma : \mathcal{D}^\sharp \to \mathcal{P}(\mathbb{Z})$

Given any value abstract domain, we can derive a non-relational abstract semantics for `Imp`, similarly to what was presented for intervals.

Property 2.52 **Non-relational abstract semantics**

Let $(\mathcal{D}^\sharp, \sqsubseteq^\sharp)$ be a value abstract domain. The program state is $\mathcal{V} \dot{\to} \mathcal{D}^\sharp$, and the lattice operators are pointwise liftings of the value abstract domain. We define:

$$\mathbb{E}^\sharp[\![\, v \,]\!]\sigma^\sharp \stackrel{\text{def}}{=} \sigma^\sharp(v)$$

$$\mathbb{E}^\sharp[\![\, c \,]\!]\sigma^\sharp \stackrel{\text{def}}{=} c^\sharp$$

$$\mathbb{E}^\sharp[\![\, [z_1, z_2] \,]\!]\sigma^\sharp \stackrel{\text{def}}{=} [z_1, z_2]^\sharp$$

$$\mathbb{E}^\sharp[\![\, e_1 \dagger e_2 \,]\!]\sigma^\sharp \stackrel{\text{def}}{=} \mathbb{E}^\sharp[\![\, e_1 \,]\!]\sigma^\sharp \dagger^\sharp \mathbb{E}^\sharp[\![\, e_2 \,]\!]\sigma^\sharp, \forall \dagger \in \{\, +, -, *, /, \% \,\}$$

$\mathbb{S}^\#[\![\,\texttt{int}\ x;\,]\!]\sigma^\sharp \stackrel{\text{def}}{=} \sigma^\sharp[x \mapsto \top]$

$\mathbb{S}^\#[\![\,x = e\,]\!]\sigma^\sharp \stackrel{\text{def}}{=}$ let $v = \mathbb{E}^\#[\![\,e\,]\!]\sigma^\sharp$ in if $v = \bot$ then $\bot$ else $\sigma^\sharp[x \mapsto \mathbb{E}^\#[\![\,e\,]\!]\sigma^\sharp]$

$\mathbb{S}^\#[\![\,\texttt{if}\ (c)\ \{s_t\}\ \texttt{else}\ \{s_f\}\,]\!] \stackrel{\text{def}}{=} \mathbb{S}^\#[\![\,s_t\,]\!] \circ \mathbb{C}^\#[\![\,c\,]\!] \dot{\sqcup}^\sharp \mathbb{S}^\#[\![\,s_f\,]\!] \circ \mathbb{C}^\#[\![\,\neg c\,]\!]$

$\mathbb{S}^\#[\![\,\texttt{while}\ (c)\ \{s\}\,]\!]\sigma^\sharp \stackrel{\text{def}}{=} \mathbb{C}^\#[\![\,\neg c\,]\!](\lim \delta^n(\bot))$ with $i(x^\sharp) \stackrel{\text{def}}{=} x^\sharp \nabla(\sigma^\sharp \sqcup^\sharp \mathbb{S}^\#[\![\,s\,]\!] \circ \mathbb{C}^\#[\![\,c\,]\!]x^\sharp)$

This semantics is a sound abstraction of the concrete semantics.

2.3.9.2 Cooperation between congruences and intervals

Let us assume that our analysis inferred that u has abstract values $[3, 5]$ and $2\mathbb{Z}$. In order to deduce that $u = 4$, we introduce a reduction operator $\rho_{\mathcal{I}\times\mathcal{C}}$ between intervals and congruences in Figure 2.11.

$$\rho_{\mathcal{I}\times\mathcal{C}}([a, b], c\mathbb{Z} + d) = \begin{cases} \bot_{\mathcal{I}}, \bot_{\mathcal{C}} & \text{if a' > b'} \\ [a', a'], 0\mathbb{Z} + a' & \text{if a' = b'} \\ [a', b'], c\mathbb{Z} + d & \text{if a' < b'} \end{cases}$$

$$\text{with } a' = \min\{\, x \geq a \mid x = kd + c, k \in \mathbb{Z}\,\}$$
$$b' = \max\{\, x \leq b \mid x = kd + c, k \in \mathbb{Z}\,\}$$

Figure 2.11: Reduction operator for intervals and congruences

The reduction operator computes updated bounds a' and b' of the interval, that are congruent to d modulo c. In the case mentionned before, the second case is used: $\rho_{\mathcal{I}\times\mathcal{C}}([3, 5], 2\mathbb{Z}+0) = ([4, 4], 0\mathbb{Z} + 2)$. The reduction operator shares information between domains in order to refine each domain's state.

Definition 2.53 **Reduction operator**

Let $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$ be two value abstract domains (from Definition 2.51). $\rho : \mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp \to \mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp$ is a reduction operator if, given $(y_1, y_2) = \rho(x_1, x_2)$:

- the reduction is sound: $\gamma_{1\times2}(y_1, y_2) = \gamma_{1\times2}(x_1, x_2)$ with $\gamma_{1\times2}(v_1, v_2) = \gamma_1(v_1) \cap \gamma_2(v_2)$
- the reduction reduces each element: $\gamma_1(y_1) \sqsubseteq_1^\sharp \gamma_1(x_1)$, and $\gamma_2(y_2) \sqsubseteq_2^\sharp \gamma_2(x_2)$

Remark 2.54 **Optimal reduction**

If both domains have a Galois connection, the following reduction ρ is optimal:

$$\rho(v_1, v_2) = (\alpha_1(\gamma_1(v_1) \cap \gamma_2(v_2)), \alpha_2(\gamma_1(v_1) \cap \gamma_2(v_2)))$$

Property 2.55 **Reduction between intervals and congruences**

$\rho_{\mathcal{I}\times\mathcal{C}}$ is a reduction operator.

Definition 2.56 **Reduced product of abstract values**

Given two value abstract domains $\mathcal{D}_1^\sharp$, $\mathcal{D}_2^\sharp$ and a reduction operator ρ on them, we can define the reduced product of these abstract values. Intuitively, ρ is applied on every element of

$\mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp$, except on the widening.

- $(\mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp, \sqsubseteq_{1\times2}^\sharp)$, where $(v_1, v_2) \sqsubseteq_{1\times2} (v_1', v_2') \stackrel{\text{def}}{=} v_1 \sqsubseteq_1 v_1' \wedge v_2 \sqsubseteq_2 v_2'$,
- a smallest element $(\bot_1, \bot_2)$ and a greatest element $(\top_1, \top_2)$
- sound abstractions of
  - intervals: $[l, u]_{1\times2}^\sharp \stackrel{\text{def}}{=} [l, u]_1^\sharp, [l, u]_2^\sharp$
  - binary operators such that for $\dagger \in \{+, -, *, /, \%\}$,

$$(v_1, v_2) \dagger_{1\times2}^\sharp (v_1', v_2) \stackrel{\text{def}}{=} \rho(v_1 \dagger_1^\sharp v_1', v_2 \dagger_2^\sharp v_2')$$

  - set union and intersection such that for $\square \in \{\sqcup, \sqcap\}$

$$(v_1, v_2) \square_{1\times2}^\sharp (v_1', v_2') \stackrel{\text{def}}{=} \rho(v_1 \square_1^\sharp v_1', v_2 \square_2^\sharp v_2')$$

- a widening operator $\nabla_{1\times2}$ applying pointwise widening without reduction.
- a concretization operator $\gamma_{1\times2} : \mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp \to \mathcal{P}(\mathbb{Z})$, $\gamma_{1\times2}(v_1, v_2) \stackrel{\text{def}}{=} \gamma_1(v_1) \cap \gamma_2(v_2)$.

Property 2.57 **Reduced product of abstract values**

The reduced product defined above is a value abstract domain.

Remark 2.58 **Reduction operator and widening**

The reduction operator is not applied during widening. Applying it would forego the guaranteed termination of the widening [35, Section 7.4].

Remark 2.59 **Reduced product of abstract domains**

As long as two abstract domains $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$ abstract the same concrete state C, it is possible to generalize the notion of reduction and reduced product [106, Section 6.2], from the setting of abstract values used here.

2.4 Extending `Imp` and its analyses

This section presents more advanced concepts and analyses. We start by extending `Imp` with immutable strings. We provide a first abstraction of strings tracking their length. We show how to perform an analysis of non-local control-flow statements, such as the `break` operator. We improve the precision of the analysis by introducing relational abstract domains. We provide a second abstraction of strings tracking an abstraction of their content, and revisit products of abstract domains.

2.4.1 Extending `Imp` with strings

We add strings to our toy `Imp` language. Similarly to the C language, a string is a sequence of characters, each represented as integers. The set of characters is $\mathcal{A} \stackrel{\text{def}}{=} [0, 127]$, and follows the ASCII encoding (for example, 'a' is 97). We write $\mathcal{A}^n$ the set of words of length n, and $\mathcal{A}^* = \cup_{n\in\mathbb{N}}\mathcal{A}^n$ is the set of words. Given characters $c, c' \in \mathcal{A}$, we write $c \cdot c' \in \mathcal{A}^2$ the concatenation of characters. Given a word w, we write $w_0, \cdots, w_{n-1}$ its characters, and $|w|$ its length. We add a type `str` for strings, the expression $expr[expr]$ to access a character, and the statement $expr[expr] = expr$ to update a string at a given index.

The values are extended to be integers or strings of any length, $\mathbb{Z} \cup \mathcal{A}^*$. The semantics has thus for signature:

$$\begin{aligned}
&\textsf{Value} = \mathbb{Z} \cup \mathcal{A}^* \\
&\mathbb{E}[\![\, e \,]\!] : \mathcal{P}(\mathcal{V} \rightharpoonup \textsf{Value}) \to \textsf{Value} \\
&\mathbb{S}[\![\, s \,]\!] : \mathcal{P}(\mathcal{V} \to \textsf{Value}) \to \mathcal{P}(\mathcal{V} \to \textsf{Value})
\end{aligned}$$

The updated semantics (defined on a single state, and lifted by the usual join-morphism) is shown in Figure 2.12. Index accesses are evaluated into the integer value (between 0 and 127) corresponding to the accessed character. A string can also be updated one element at a time through an integer value. We assume that expressions have been typed before the analysis and that the type information is available.

$$\begin{aligned}
&\mathbb{E}[\![\, s[i] \,]\!]\sigma \stackrel{\text{def}}{=} \{\, s_i \mid s \in \mathbb{E}[\![\, s \,]\!]\sigma \wedge s \in \mathcal{A}^* \wedge 0 \leq i \leq n-1 \wedge s = s_0 \cdot \ldots \cdot s_{n-1} \,\} \\
&\mathbb{S}[\![\, \texttt{str}\ \ s; \,]\!]\sigma \stackrel{\text{def}}{=} \{\, \sigma[s \mapsto c] \mid c \in \mathcal{A}^* \,\} \\
&\mathbb{S}[\![\, s = c_0 \cdots c_{n-1} \,]\!]\sigma \stackrel{\text{def}}{=} \sigma[s \mapsto c_0 \cdots c_{n-1}] \\
&\mathbb{S}[\![\, s = t \,]\!]\sigma \stackrel{\text{def}}{=} \sigma[s \mapsto \sigma(t)] \\
&\mathbb{S}[\![\, s[i] = t \,]\!]\sigma \stackrel{\text{def}}{=} \sigma[s \mapsto \tilde{s}], \tilde{s} = s_0 \cdots s_{i-1} \cdot t \cdot s_{i+1} \cdots s_{n-1}
\end{aligned}$$

Figure 2.12: Updated semantics with strings

The introduction of strings comes with new errors we are interested in detecting: out-of-bound accesses, and updating a string with a non-ASCII value (i.e., `s[i] = t` where t evaluates to a value that is not between 0 and 127).

Example 2.60 **Program on strings**

An example program is provided in Listing 2.3. After line 3, the program state is (the ASCII code for 'd' is 100):

$$(s \mapsto \text{"}abcd\text{"}, t \mapsto \text{"}abcd\text{"}, l \mapsto 101)$$

Note that strings are copied during assignments. At the end of the program, s is the string "abce", and t is still "abcd".

Listing 2.3: `Imp` program with strings

```
1 str s = "abcd";
2 str t = s;
3 int l = s[3] + 1;
4 s[3] = l;
```

2.4.2 Using ghost variables to track string length

One way to abstract the content of strings is to keep only their length. This can be done by adding numerical variables in the abstract states representing the length of each string. We will keep the length of a string s into an auxiliary variable $\underline{\text{len}}(s)$. The string operators can then be rewritten as numerical operations, that can be delegated to the numeric domain. In the area of deductive verification, the variable $\underline{\text{len}}(s)$ variable would be called a ghost variable, as it is used to prove a program property but does not appear in the original program. This

naming is also used by Chevalier and Feret [28] in the setting of static analysis. Contrary to the value abstract domains presented earlier, this abstract domain is stateless[2] and will delegate the operations to the underlying numerical domains. The benefits of this approach will be discussed in the presentation of relational abstract domains, and is one of the cornerstones of the design of abstract domains in Mopsa.

We assume we have a numerical abstract domain $\mathcal{D}^\sharp_{\text{num}}$, handling numeric expressions and assignments through the abstract operators $\mathbb{E}^\#_{\text{num}}[\![\cdot]\!], \mathbb{S}^\#_{\text{num}}[\![\cdot]\!], \mathbb{C}^\#_{\text{num}}[\![\cdot]\!]$.

The semantics of this domain extends the numerical one with the transfer functions of Figure 2.13. The case of index access checks that the index has valid bounds. If that is the case, we know the return value is an ASCII number, i.e., a number between 0 and 127. Upon a variable declaration, we initialize the auxiliary length variable to be non-negative. Variable assignments are rewritten into a matching length assignment. The analysis of a character update checks that the bound is valid and that the character is a valid ASCII code.

$$
\begin{aligned}
&\mathbb{E}^\#_{\text{len}}[\![\, e \,]\!] : \mathcal{D}^\sharp_{\text{num}} \to \mathcal{I} \\
&\mathbb{E}^\#_{\text{len}}[\![\, s[i] \,]\!] \overset{\text{def}}{=} \mathbb{E}^\#_{\text{num}}[\![\, [0, 127] \,]\!] \circ \mathbb{C}^\#_{\text{num}}[\![\, 0 \leq i \wedge i < \underline{\text{len}}(s) \,]\!] \\
&\mathbb{S}^\#_{\text{len}}[\![\, s \,]\!] : \mathcal{D}^\sharp_{\text{num}} \to \mathcal{D}^\sharp_{\text{num}} \\
&\mathbb{S}^\#_{\text{len}}[\![\, \texttt{str } s; \,]\!] \overset{\text{def}}{=} \mathbb{S}^\#_{\text{num}}[\![\, \underline{\text{len}}(s) = [0, +\infty] \,]\!] \\
&\mathbb{S}^\#_{\text{len}}[\![\, s \in \mathcal{V}_{\text{str}} = c_0 \cdots c_{n-1} \in \mathcal{A}^* \,]\!] \overset{\text{def}}{=} \mathbb{S}^\#_{\text{num}}[\![\, \underline{\text{len}}(s) = n \,]\!] \\
&\mathbb{S}^\#_{\text{len}}[\![\, s \in \mathcal{V}_{\text{str}} = s' \in \mathcal{V}_{\text{str}} \,]\!] \overset{\text{def}}{=} \mathbb{S}^\#_{\text{num}}[\![\, \underline{\text{len}}(s) = \underline{\text{len}}(s') \,]\!] \\
&\mathbb{S}^\#_{\text{len}}[\![\, s[i] = e \,]\!] \overset{\text{def}}{=} \mathbb{C}^\#_{\text{num}}[\![\, 0 \leq e \leq 127 \,]\!] \circ \mathbb{C}^\#_{\text{num}}[\![\, 0 \leq i \wedge i < \underline{\text{len}}(s) \,]\!]
\end{aligned}
$$

Figure 2.13: String length abstract domain

Example 2.61 **Program analysis using the string length domain**

Going back to the example of Listing 2.3, we get at the end of line 3:

$$\sigma^\sharp = (\underline{\text{len}}(s) \mapsto [4, 4], \underline{\text{len}}(t) \mapsto [4, 4], l \mapsto [0, 128])$$

Thus, our analysis infers that the assignment `s[3] = l` at line 4 can be incoherent when $l = 128$ (128 is not in the range of valid ASCII codes). This is a false alarm, since the program is correct.

Remark 2.62 **Convention: format of auxiliary variables**

In the rest of this thesis, auxiliary variables such as $\underline{\text{len}}(s)$ are always underlined.

We assume that our numerical abstract domain $\mathcal{D}^\sharp_{\text{num}}$ has a concretization function $\gamma_{\text{num}} : \mathcal{D}^\sharp_{\text{num}} \to \mathcal{P}(\mathcal{V} \to \mathbb{Z})$. We define the concretization of the string length domain below.

Definition 2.63 **String length concretization**

The string length concretization is defined below. It consists in the set of environments s, such that there exists a concrete numerical state $\sigma \in \gamma_{\text{num}}(\sigma^\sharp)$, and where:

[2] We can say its state is `unit`, and the sole inhabitant of `unit` is written `()`. The lattice operators on this domain are straightforward to define.

- all non-length variables are kept the same between s and σ,
- for all length variable $\underline{\mathsf{len}}(v)$, v is concretized as any string of length $\sigma(\underline{\mathsf{len}}(v))$.

$$\begin{aligned}\gamma_{\mathsf{length}}(\sigma^\sharp \in \mathcal{D}^\sharp_{\mathsf{num}}) = \{\, s \mid \sigma \in \gamma_{\mathsf{num}}(\sigma^\sharp),\\ v \in \mathsf{dom}\sigma \setminus \{\, \underline{\mathsf{len}}(\cdot) \,\} \Leftrightarrow s(v) = \sigma(v);\\ \underline{\mathsf{len}}(v) \in \mathsf{dom}\sigma \Leftrightarrow s(v) \in \mathcal{A}^{\sigma(\underline{\mathsf{len}}(v))} \,\}\end{aligned}$$

Example 2.64 **String length concretization**

Keeping the abstract state $\sigma^\sharp$ of Example 2.61, its concretization numerical concretization is

$$\gamma_{\mathsf{num}}(\sigma^\sharp) = \{\, (\underline{\mathsf{len}}(s) \mapsto 4, \underline{\mathsf{len}}(t) \mapsto 4, l \mapsto v_l) \mid 0 \leq v_l \leq 128 \,\}$$

We thus obtain

$$\gamma_{\mathsf{length}}(\sigma^\sharp) = \left\{\, (s \mapsto c_0 \cdots c_3, t \mapsto c_4 \cdots c_7, l \mapsto v_l) \;\middle|\; 0 \leq v_l \leq 128, (c_0, \ldots, c_7) \in [0, 127]^8 \,\right\}$$

In particular $(s \mapsto \text{"}abce\text{"}, t \mapsto \text{"}abcd\text{"}, l \mapsto 101) \in \Sigma$.

2.4.3 Breaking out of a loop

Our semantics (both concrete and abstract) are currently defined by induction over the syntax. It was easy to define since the control flow of the program matched the program's structure. However, there are cases where the control-flow is not linear: `break` statements stop the execution of loops, `return` statements exit functions, raised exceptions interrupt the normal execution until they are caught.

We focus here on adding a `break` statement to `Imp`. To that end, the states are now stored as continuations tagged by a control-flow token, following the approach used by Cousot et al. [35, footnote 4, page 6] in Astrée. The set of control-flow tokens is $\mathcal{F} = \{\, \mathit{cur}, \mathit{brk} \,\}$. *cur* denotes the normal control-flow, and *brk* is used to tag states that reached a `break` statement in the currently analyzed loop body.

We give in Figure 2.14 the updated semantics of `Imp` with these control-flow tokens. Expressions are not evaluated in a state tagged by *brk*, and the usual semantics applies for normal states. The semantics of `break` transforms normal states reaching it into states tagged by the *brk* token and keeps the other states already tagged with a *brk* token. The semantics of `while` loops returns the union of:

- input states tagged *brk* before the loop. They must correspond to `break` statements in the body of the loop immediately surrounding the current one. They are still tagged with *brk* in the output so that they can reach the end of the outer loop.
- states of the loop invariant Σ_{lfp} that exited the loop through a `break` statement are transformed back into normal states using the *cur* token (since the loop is exited),
- states of the loop invariant Σ_{lfp} denoting a normal execution are filtered by the negation of the loop's guard.

The semantics of other statements is lifted from the previous semantics: statements are executed in the normal control flow, and skipped in states tagged by the *brk* token.

$$\mathbb{E}_{\mathsf{brk}}[\![\cdot]\!] : \mathcal{F} \times (\mathcal{V} \rightharpoonup \mathsf{Value}) \to \mathcal{P}(\mathsf{Value})$$
$$\mathbb{E}_{\mathsf{brk}}[\![e]\!](cur, \sigma) \stackrel{\text{def}}{=} \mathbb{E}[\![e]\!]\sigma$$
$$\mathbb{S}_{\mathsf{brk}}[\![\cdot]\!] : \mathcal{P}(\mathcal{F} \times (\mathcal{V} \rightharpoonup \mathsf{Value})) \to \mathcal{P}(\mathcal{F} \times (\mathcal{V} \rightharpoonup \mathsf{Value}))$$
$$\mathbb{S}_{\mathsf{brk}}[\![\texttt{break}]\!]\Sigma \stackrel{\text{def}}{=} \{\,(brk, \sigma) \mid (cur, \sigma) \in \Sigma\,\} \cup \{\,(brk, \sigma) \in \Sigma\,\}$$
$$\mathbb{S}_{\mathsf{brk}}[\![\texttt{while}\ (c)\ \{s\}]\!]\Sigma \stackrel{\text{def}}{=}$$
$$\quad \text{let } \Sigma_{\mathsf{in}} = \{\,(cur, \sigma) \in \Sigma\,\} \text{ and } \Sigma_{\mathsf{lfp}} = \mathsf{lfp}\, f \text{ in}$$
$$\quad \{\,(brk, \sigma) \in \Sigma\,\} \cup \{\,(cur, \sigma) \mid (brk, \sigma) \in \Sigma_{\mathsf{lfp}}\,\} \cup \mathbb{C}[\![\neg c]\!]\{\,(cur, \sigma) \in \Sigma_{\mathsf{lfp}}\,\}$$
$$\quad \text{with } f(X) \stackrel{\text{def}}{=} \Sigma_{\mathsf{in}} \cup \mathbb{S}[\![s]\!] \circ \mathbb{C}[\![c]\!]X$$
$$\mathbb{S}_{\mathsf{brk}}[\![s]\!]\Sigma \stackrel{\text{def}}{=} \{\,(cur, \sigma') \mid \sigma' \in \mathbb{S}[\![s]\!]\sigma, (cur, \sigma) \in \Sigma\,\} \cup \{\,(brk, \sigma) \in \Sigma\,\}$$

Figure 2.14: Semantics of `Imp` with control-flow tokens

Listing 2.4: `Imp` program with `break`

```
1 str s = "abcd";
2 int c = 99;  ord('c') = 99
3 int i = 0;
4 while (i < len(s)) {
5   if(s[i] == c) break;
6   i += 1;
7 }
```

Example 2.65 **Concrete semantics of a loop with a `break`**

We illustrate the semantics on the example of Listing 2.4. This program searches for the first position of the character 'c' (whose value is stored in `c`), in the chain `s`. At the end of line 3, there is only one state $\sigma = cur, (s \mapsto \text{"}abcd\text{"}, c \mapsto 99, i \mapsto 0)$. We can compute $\mathsf{lfp}\, f$ through successive loop unrollings:

$$(\mathbb{S}_{\mathsf{brk}}[\![s]\!] \circ \mathbb{C}_{\mathsf{brk}}[\![c]\!])\{\sigma\} = \{\, cur \mapsto (s \mapsto \text{"}abcd\text{"}, c \mapsto 99, i \mapsto 1)\,\}$$
$$(\mathbb{S}_{\mathsf{brk}}[\![s]\!] \circ \mathbb{C}_{\mathsf{brk}}[\![c]\!])^2\{\sigma\} = \{\, cur \mapsto (s \mapsto \text{"}abcd\text{"}, c \mapsto 99, i \mapsto 2)\,\}$$
$$(\mathbb{S}_{\mathsf{brk}}[\![s]\!] \circ \mathbb{C}_{\mathsf{brk}}[\![c]\!])^3\{\sigma\} = \{\, brk \mapsto (s \mapsto \text{"}abcd\text{"}, c \mapsto 99, i \mapsto 2)\,\}$$
$$\forall n \geq 3, (\mathbb{S}_{\mathsf{brk}}[\![s]\!] \circ \mathbb{C}_{\mathsf{brk}}[\![c]\!])^n\{\sigma\} = (\mathbb{S}_{\mathsf{brk}}[\![s]\!] \circ \mathbb{C}_{\mathsf{brk}}[\![c]\!])^3\{\sigma\}$$

$$\Sigma_{\mathsf{lfp}} = \{\, cur \mapsto (s \mapsto \text{"}abcd\text{"}, c \mapsto 99, i \mapsto v_i) \mid 0 \leq v_i \leq 2\,\}$$
$$\qquad \cup \{\, brk \mapsto (s \mapsto \text{"}abcd\text{"}, c \mapsto 99, i \mapsto 2)\,\}$$

The state after the loop is $(cur \mapsto (s \mapsto \text{"}abcd\text{"}, c \mapsto 99, i \mapsto 2))$.

The abstract semantics is lifted similarly to the concrete, we only provide the signatures.

$$\mathsf{Value}^\sharp = \mathcal{I}$$
$$\mathbb{E}^\#_{\mathsf{brk}}[\![\cdot]\!] : (\mathcal{F} \rightharpoonup \mathcal{D}^\sharp_{\mathsf{num}}) \to \mathsf{Value}^\sharp$$
$$\mathbb{S}^\#_{\mathsf{brk}}[\![\cdot]\!] : (\mathcal{F} \rightharpoonup \mathcal{D}^\sharp_{\mathsf{num}}) \to (\mathcal{F} \rightharpoonup \mathcal{D}^\sharp_{\mathsf{num}})$$

Example 2.66 **Abstract semantics of a loop with a `break`**

We show how the fixpoint is reached in the abstract, assuming that the numerical domain used is intervals (i.e., $\mathcal{D}^\sharp_{\text{num}} = \mathcal{V} \rightharpoonup \mathcal{I}$). At the end of line 3, the abstract state is $\sigma^\sharp = \mathit{cur} \mapsto (\underline{\text{len}}(s) \mapsto [4,4], c \mapsto [99,99], i \mapsto [0,0])$.

$$
\begin{aligned}
f^0(\dot\bot_{\mathcal{I}}) &= \dot\bot_{\mathcal{I}} \\
f^1(\dot\bot_{\mathcal{I}}) &= \sigma^\sharp \\
f^2(\dot\bot_{\mathcal{I}}) &= f(\sigma^\sharp) = \sigma^\sharp \dot\nabla_{\mathcal{I}} (\sigma^\sharp \dot\sqcup^\sharp_I (\mathbb{S}^\sharp_{\text{brk}}[\![\, s \,]\!] \circ \mathbb{C}^\sharp_{\text{brk}}[\![\, c \,]\!]) \sigma^\sharp) \\
&= \sigma^\sharp \dot\nabla_I \{\, \mathit{brk} \mapsto (i \in [0,0])\,, \mathit{cur} \mapsto (i \mapsto [0,1]) \,\} \\
&= \{\, \mathit{cur} \mapsto (i \mapsto [0,0]) \,\} \dot\nabla_{\mathcal{I}} \{\, \mathit{brk} \mapsto (i \in [0,0])\,, \mathit{cur} \mapsto (i \mapsto [0,1]) \,\} \\
&= \{\, \mathit{brk} \mapsto (i \in [0,0])\,, \mathit{cur} \mapsto (i \mapsto [0,+\infty]) \,\} \\
f^3(\dot\bot_{\mathcal{I}}) &= f(f^2(\dot\bot_{\mathcal{I}})) = f^2(\dot\bot_{\mathcal{I}}) \dot\nabla_{\mathcal{I}} (\sigma^\sharp \dot\sqcup^\sharp_I (\mathbb{S}^\sharp_{\text{brk}}[\![\, s \,]\!] \circ \mathbb{C}^\sharp_{\text{brk}}[\![\, c \,]\!]) f^2(\dot\bot_{\mathcal{I}})) \\
&= f^2(\dot\bot_{\mathcal{I}}) \dot\nabla_{\mathcal{I}} (\sigma^\sharp \dot\sqcup^\sharp_I \{\, \mathit{brk} \mapsto (i \mapsto [0,3])\,, \mathit{cur} \mapsto (i \mapsto [0,4]) \,\}) \\
&= \{\, \mathit{brk} \mapsto (i \in [0,+\infty])\,, \mathit{cur} \mapsto (i \mapsto [0,+\infty]) \,\} \\
\forall n \geq 4, f^n(\dot\bot_{\mathcal{I}}) &= f^3(\dot\bot_{\mathcal{I}})
\end{aligned}
$$

With a decreasing iteration, we can obtain that the following abstract state is a fixpoint:

$$(\mathit{brk} \mapsto (i \in [0,3])\,, \mathit{cur} \mapsto (i \mapsto [0,4]))$$

From the transfer function of while, the result after the loop is

$$\mathbb{C}^\sharp_{\text{brk}}[\![\, \neg c \,]\!](\mathit{cur} \mapsto i \mapsto [0,4]) \dot\sqcup^\sharp_I (\mathit{cur} \mapsto i \mapsto [0,3]) = \mathit{cur} \mapsto i \mapsto [0,4]$$

Remark 2.67 **Widening and flow tokens**

In Example 2.66, the widening is applied pointwise to each token. It is also possible to define the widening only on the normal token *cur*, and the *brk*-labeled state will naturally stabilize afterwards.

Remark 2.68 **CFG analysis**

A typical approach of static analysis is to propagate abstract states along the control-flow graph (CFG) edges, using an iteration strategy (such as the one presented by Bourdoncle [18]). In a naïve implementation, these analyses store one abstract state per CFG node, which is memory-intensive. The approach we presented uses flow tokens, storing only abstract states that are necessary for further computations. However, the iteration strategy follows the program linearly, while CFG-based approaches can employ different iteration strategies.

2.4.4 Relational invariants

Let us assume that we want to analyze the example of string search (Listing 2.4), but on an arbitrary string s. This may be due to imprecision of the analysis or needed to infer generic contracts over functions.

With a non-relational domain such as intervals, we will only be able to prove that $\underline{\text{len}}(s) \geq 0 \wedge i \geq 0$. In particular, it is not possible to prove that the index accesses $s[i]$ in the loop are valid, since a non-relational analysis cannot infer that $i < \underline{\text{len}}(s)$ when $\underline{\text{len}}(s)$ is not constant. In order to remedy this problem, relational abstract domains have been introduced. Their goal

is to express relationships between variables. The most popular relational abstract domains are the polyhedra domain and the octagon domain respectively introduced by Cousot and Halbwachs [34] and Miné [108]. A comparison of the different numerical domains is provided in Figure 2.15 (the costs are meant with respect to the number of variables).

Domain	Constraint Shape	Memory Cost	Cost per Operation
Intervals	$V_i \in [l_i, u_i]$	linear	linear
Octagons	$\pm V_i \pm V_j \leq c_{ij}$	quadratic	cubic
Polyhedra	$\sum_i \alpha_i V_i \leq \beta_i$	exponential	exponential

Figure 2.15: Comparison of the numerical abstract domains

Example 2.69 **Graphic representations of numerical abstract domains**

We show how the interval, octagon and polyhedra abstract domains represent the constraint $0 \leq 2i \leq \underline{\text{len}}(s)$ in Figure 2.16. The interval domain is not relational, it can only represent $(i \mapsto [0, +\infty], \underline{\text{len}}(s) \mapsto [0, +\infty])$. Octagons cannot represent non-unitary constraints between variables, so the corresponding abstract element is overapproximated $0 \leq i \leq \underline{\text{len}}(s)$. On the other hand, polyhedra can exactly represent the constraints.

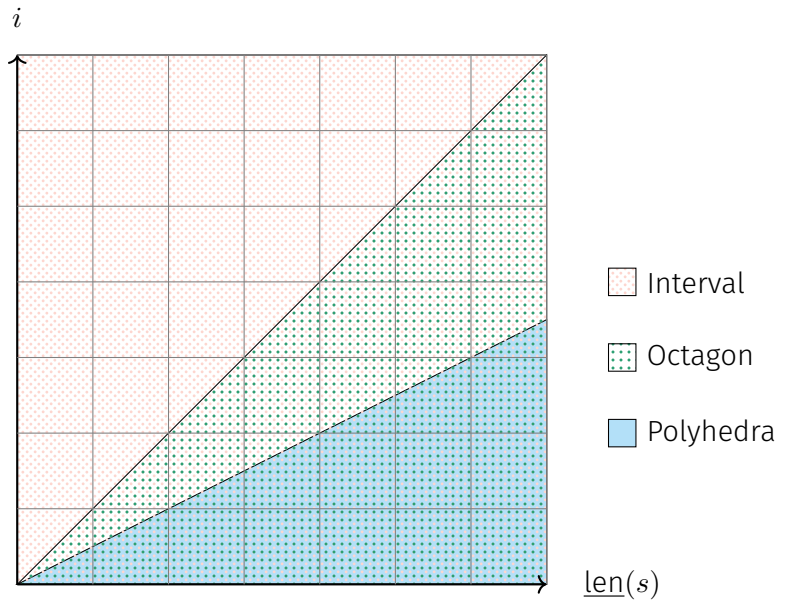

Figure 2.16: Interval, Octagon and Polyhedra Abstract Domains Representing $0 \leq 2i \leq \underline{\text{len}}(s)$

With either polyhedra or octagons as our underlying numerical domain, we would then be able to prove that the state reachable after the loop in the string search program is

$$\mathit{cur} \mapsto (i < \underline{\text{len}}(l)) \sqcup^\sharp \mathit{cur} \mapsto (i = \underline{\text{len}}(l))$$

The first case corresponds to the loop not finding the character, and the second case to an early break statement.

Remark 2.70 **Drawbacks of relational domains**

Relational abstract domains are powerful and improve the analysis's precision, just as we have seen. However, they also come with additional constraints:

- Computational cost: operations on relational abstract domains are more costly in terms of CPU and memory usage.
- Domain sharing: to have the best precision, all numerical variables should be shared in the same relational abstract domain. In a non-relational setting, we could have defined the string length domain of Section 2.4.2 with its own interval domain, different from the one handling integer variables of the program. Since we want to keep the best precision, we designed the string length domain as a domain delegating to an underlying numerical domain. This approach is more complex, requiring specific definitions of the concretization functions (shown in Section 2.5), and careful handling of reduced product (presented in the next chapter, in Section 3.3.4). It is, however, more precise since we can infer relationships between the ghost length variables and integer variables, such as $0 \leq i \leq \underline{\text{len}}(l)$ in the case of the program of Listing 2.4.

2.4.5 Summarization of string content

The string length domain of Section 2.4.2 is useful to prove correct index accesses, but it is imprecise to track characters' values. In the basic program of Listing 2.5, the string length domain is unable to prove that the assignment `s[3] = s[3] - 1` is valid, since the imprecise analysis inferred that `s[3] - 1` can be `-1`, which is an invalid ASCII code.

Listing 2.5: `Imp` program with strings

```
1 str s = "bcde";
2 s[3] = s[3] - 1;
```

We thus introduce a new abstract domain for strings. For each string variable s, the ghost variable $\underline{\text{ord}}(s)$ represents the ASCII codes contained by the string. Similarly to the string length domain presented previously, this domain is stateless and delegates its numerical operations to an underlying abstract domain $\mathcal{D}^\sharp_{\text{num}}$. The only difference is that the expression evaluation now returns an expression $e' \in \mathcal{E}$ to be relational. Instead of an interval (or another abstract value), we can thus translate a string access into its contents' variable and infer relations between the contents and other program variables.

$$\mathbb{E}^{\#}_{\text{str}}[\![\, e \,]\!] : \mathcal{D}^\sharp_{\text{num}} \to \mathcal{E}$$
$$\mathbb{S}^{\#}_{\text{str}}[\![\, s \,]\!] : \mathcal{D}^\sharp_{\text{num}} \to \mathcal{D}^\sharp_{\text{num}}$$

There is however a further difficulty to handle that appears, as the auxiliary variable $\underline{\text{ord}}(s)$ represents multiple concrete elements (i.e., the characters at all positions of the string). We assume we have an unknown string s, whose characters are between two integers (or other numerical variables) a and b, i.e. $a \leq \underline{\text{ord}}(s) \leq b$. We consider two simple programs:

▷ `x = s[0]; y = s[1];` If we perform a drop-in replacement of $s[i]$ by $\underline{\text{ord}}(s)$, our abstract state contains the constraint $x = y = \underline{\text{ord}}(s)$, which is unsound, since $s[0]$ and $s[1]$ can be different in the concrete.

▷ `s[3] = s[3] - 1;` The drop-in replacement does not work either: $a - 1 \leq \underline{\text{ord}}(s) \leq b - 1$ is unsound too, since another character of s may have value b.

Intuitively, we need to perform temporary copies of the variable $\underline{\text{ord}}(s)$ for each access, to distinguish between the concrete location actually referenced by the access and the other locations also summarized by $\underline{\text{ord}}(s)$. These copies will not be equal to $\underline{\text{ord}}(s)$ since they may represent other locations, but they are bound by the same constraints as $\underline{\text{ord}}(s)$. We introduce two new operators to handle these cases. We provide their concrete definition. We refer to the work of Gopan et al. [63], Siegel and Simon [135] for computable definitions in the abstract.

Definition 2.71 **Expand operator**

The `expand` operator copies variable v into v', meaning that variables v and v' have the same possible values. In the abstract, this means that constraints on v are duplicated to be constraints on v' too.

$$\mathbb{S}[\![\, \texttt{expand}(v, v') \,]\!]\Sigma = \{\, \sigma[v' \mapsto z] \mid \sigma \in \Sigma, \sigma(v) = z \,\}$$

Example 2.72 **Handling `x = s[0]; y = s[1];`**

In the first case, we could have expanded $\underline{\text{ord}}(s)$ into x and y, to get $a \leq v \leq b, \forall v \in \{\, x, y, \underline{\text{ord}}(s) \,\}$.

Definition 2.73 **Fold operator**

The `fold` operator performs the dual operation to `expand`. It removes v' from the environment, and its value is added as a potential value for v.

$$\mathbb{S}[\![\, \texttt{fold}(v, v') \,]\!]\Sigma = \left\{ \sigma' \;\middle|\; \exists \sigma \in \Sigma, \sigma'(v) \in \{\, \sigma(v), \sigma(v') \,\} \wedge \right. \\ \left. x \in \text{dom}\sigma \setminus \{\, v, v' \,\} \Leftrightarrow \sigma(x) = \sigma'(x) \right\}$$

Example 2.74 **Handling `s[3] = s[3] - 1;`**

In the second case, we could introduce a fresh variable t, such that `t = s[3] - 1`. Our domain is then $a \leq \underline{\text{ord}}(s) \leq b, a - 1 \leq t \leq b - 1$. We then `fold` t into $\underline{\text{ord}}(s)$, to get: $a - 1 \leq \underline{\text{ord}}(s) \leq b$.

Remark 2.75 **Weak variables**

Variables that may represent multiple concrete elements are called weak variables. As we have seen, these weak variables have to be handled with special operators to be sound.

We suffix variables with a "`weak`" subscript to indicate weak variables (for example, $\underline{\text{ord}}(s)_{\texttt{weak}}$). We introduce new assignment operators handling these variables to simplify the upcoming definitions.

In the case of an assignment where the expression e contains weak variables, we expand all those weak variables using the eponymous function (these temporarily expanded variables will be removed at the end of the assignment). If the left-hand side is weak too, we introduce a fresh variable, which is then folded back into x.

$$\mathbb{S}^{\#}_{\text{num}}[\![\, x = e \,]\!] = \mathbb{S}^{\#}_{\text{num}}[\![\, x = \texttt{expanded}(e) \,]\!]$$
$$\mathbb{S}^{\#}_{\text{num}}[\![\, x_{\texttt{weak}} = e \,]\!] = \mathbb{S}^{\#}_{\text{num}}[\![\, \texttt{fold}(x, \texttt{fresh}) \,]\!] \circ \mathbb{S}^{\#}_{\text{num}}[\![\, \texttt{fresh} = \texttt{expanded}(e) \,]\!]$$

We can now define the semantics of the string summarization domain, in Figure 2.17. Index accesses are evaluated into their corresponding ghost variable, which is weak. Note that each access is seen as potentially invalid in this domain since it does not track length.[3] A string declaration initializes its corresponding ghost variable to the range of ASCII codes (in that case, a strong update is performed, so there is no need to tag the variable as weak). During

[3] We will combine both string domains in Section 2.4.6.

an assignment of a constant string, we perform a strong update for the first character and weak updates for the others. A string copy is seen as enforcing equality between two ghost content variables (however, the ghost variable on the right-hand side has to be expanded, so it is tagged as weak). To update a character, we check that the expression represents an ASCII code and perform a weak update on the ghost variable.

$$\mathbb{E}^\sharp_{\mathsf{str}}[\![\, s[i] \,]\!] \overset{\text{def}}{=} \mathbb{E}^\sharp_{\mathsf{num}}[\![\, \underline{\mathsf{ord}}(s)_{\mathtt{weak}} \,]\!]$$
$$\mathbb{S}^\sharp_{\mathsf{str}}[\![\, \mathtt{str}\ \ s; \,]\!] \overset{\text{def}}{=} \mathbb{S}^\sharp_{\mathsf{num}}[\![\, \underline{\mathsf{ord}}(s) = [0, 127] \,]\!]$$
$$\mathbb{S}^\sharp_{\mathsf{str}}[\![\, s \in \mathcal{V}_{\mathsf{str}} = c_0 \cdots c_{n-1} \in \mathcal{A}^* \,]\!] \overset{\text{def}}{=}$$
$$\mathbb{S}^\sharp_{\mathsf{num}}[\![\, \underline{\mathsf{ord}}(s)_{\mathtt{weak}} = c_{n-1} \,]\!] \circ \ldots \circ \mathbb{S}^\sharp_{\mathsf{num}}[\![\, \underline{\mathsf{ord}}(s)_{\mathtt{weak}} = c_1 \,]\!] \circ \mathbb{S}^\sharp_{\mathsf{num}}[\![\, \underline{\mathsf{ord}}(s) = c_0 \,]\!]$$
$$\mathbb{S}^\sharp_{\mathsf{str}}[\![\, s \in \mathcal{V}_{\mathsf{str}} = s' \in \mathcal{V}_{\mathsf{str}} \,]\!] \overset{\text{def}}{=} \mathbb{S}^\sharp_{\mathsf{num}}[\![\, \underline{\mathsf{ord}}(s) = \underline{\mathsf{ord}}(s')_{\mathtt{weak}} \,]\!]$$
$$\mathbb{S}^\sharp_{\mathsf{str}}[\![\, s[i] = e \,]\!] \overset{\text{def}}{=} \mathbb{S}^\sharp_{\mathsf{num}}[\![\, \underline{\mathsf{ord}}(s)_{\mathtt{weak}} = e \,]\!] \circ \mathbb{C}^\sharp_{\mathsf{num}}[\![\, 0 \leq e \leq 127 \,]\!]$$

Figure 2.17: String summary abstract domain

Example 2.76 **Weak update**

In the concrete, we know that $s =$ "$bcdd$" at the end of the program of Listing 2.5. In the abstract, after the assignment of s, we have $98 \leq \underline{\mathsf{ord}}(s) \leq 101$. Due to the weak update, we can only infer that $97 \leq \underline{\mathsf{ord}}(s) \leq 101$, which represents the set of strings containing only the letters a, b, c, d.

The concretization of this domain is more involved. We start with an example. Let us assume that the abstract state is $\sigma^\sharp = 97 \leq \underline{\mathsf{ord}}(s) < i \leq 100$. Its concretization consists of six different states:

$$\gamma_{\mathsf{num}}(\sigma^\sharp) = \begin{pmatrix} i \\ \underline{\mathsf{ord}}(s) \end{pmatrix} \in \left\{ \begin{pmatrix} 98 \\ 97 \end{pmatrix}, \begin{pmatrix} 99 \\ 97 \end{pmatrix}, \begin{pmatrix} 99 \\ 98 \end{pmatrix}, \begin{pmatrix} 100 \\ 97 \end{pmatrix}, \begin{pmatrix} 100 \\ 98 \end{pmatrix} \begin{pmatrix} 100 \\ 99 \end{pmatrix} \right\}$$

Since $\underline{\mathsf{ord}}(s)$ summarizes multiple concrete elements, we intuitively want to draw different values from different states for each character of the string s. However, these draws should be *consistent*, to enforce the relational constraints and be sound. For example, assume we pick $(i, \underline{\mathsf{ord}}(s))$ among $(99, 97)$ and $(100, 99)$. The case $i = 99, s =$ "ac" should not be in the concrete state since $s[1] \not< i$.

In other words, the concretization performs an *expansion* of the auxiliary variable $\underline{\mathsf{ord}}(s)$ into each concrete character.

Definition 2.77 **String summarization concretization**

Given $\Sigma \in \mathcal{P}(\mathcal{V} \to \mathbb{Z})$ and $v \in \mathcal{V}$, $\Sigma(v)$ is the set of images of v: $\Sigma(v) = \{\, \sigma(v) \mid v \in \mathcal{V} \,\}$. We write $|X|$ for the cardinal of the set X.

The concretization creates concrete states s from a set of concrete numerical states $\Sigma \subseteq \gamma_{\mathsf{num}}(\sigma^\sharp)$. This set Σ should be consistent on all variables v not handled by the string summarization domain (i.e., $|\Sigma(v)| = 1$, except the auxiliary variables $\underline{\mathsf{ord}}(\cdot)$). In that case, the concrete state for these variables is $s(v) = \Sigma(v)$. For auxiliary variables $\underline{\mathsf{ord}}(v)$, $s(v)$ will be a string of any length, where all characters are picked in the set $\Sigma(\underline{\mathsf{ord}}(v))$.

$$\begin{aligned}\gamma_{str}(\sigma^\sharp) = \{\, s \mid \Sigma \subseteq \gamma_{\mathsf{num}}(\sigma^\sharp),\\ v \in \mathsf{dom}\Sigma \setminus \{\, \underline{\mathsf{ord}}(\cdot) \,\} \Leftrightarrow (|\Sigma(v)| = 1 \wedge s(v) = \Sigma(v));\\ \underline{\mathsf{ord}}(v) \in \mathsf{dom}\Sigma \Leftrightarrow \exists n \in \mathbb{N}, \exists (c_0, \ldots, c_{n-1}) \in \mathcal{A}^n,\\ (s(v) = c_0 \cdots c_{n-1} \wedge \forall i \in [0, n-1], c_i \in \Sigma(\underline{\mathsf{ord}}(v)))\,\}\end{aligned}$$

Example 2.78 **String summarization concretization**

We go back to the example where $\sigma^\sharp = 97 \leq \underline{\mathsf{ord}}(s) < i \leq 100$.

The coherent states of $\gamma_{\mathsf{num}}(\sigma^\sharp)$ should have the same value for i. We consider only the three coherent states Σ below (which maximize size), since subsets of Σ would yield concrete states already covered by concrete states related to Σ. The first line are the values of i, and the second of $\underline{\mathsf{ord}}(s)$.

$$\Sigma \in \left\{ \left\{ \begin{pmatrix}98\\97\end{pmatrix} \right\}, \left\{ \begin{pmatrix}99\\97\end{pmatrix}; \begin{pmatrix}99\\98\end{pmatrix} \right\}, \left\{ \begin{pmatrix}100\\97\end{pmatrix}; \begin{pmatrix}100\\98\end{pmatrix}; \begin{pmatrix}100\\99\end{pmatrix} \right\} \right\}$$

Given each coherent state above, we respectively get the following concrete states (where we use regular expressions to denote string sets):

- $\{\, (i \mapsto 98, s \mapsto v_s) \mid v_s \in a^* \,\}$,
- $\{\, (i \mapsto 98, s \mapsto v_s) \mid v_s \in (a|b)^* \,\}$,
- $\{\, (i \mapsto 100, s \mapsto v_s) \mid v_s \in (a|b|c)^* \,\}$

Remark 2.79 **Consistency conditions**

The consistency conditions are only here to ensure that relations in the abstract are kept in the concrete states. In a non-relational setting, there are no relations to preserve and thus no need to ensure consistency.

2.4.6 Combining string length and summarization

Let us assume we run a modified version of the string search program (Listing 2.4), where we search for a character $c > 100$ in the string `s`, consisting in a random string containing only the characters "a" to "d". We would like the analysis to infer that the search will be unsuccessful and that $i = \underline{\mathsf{len}}(s)$. This is possible by combining the string length and string summarization domains in a reduced product on top of a relational abstract domain. We will need to introduce reduction rules for the evaluation of `s[i]`. Intuitively, the length domain will be able to prove that the accesses are valid, and the summarization one will give a more precise result based on the contents of the string.

There is, however, an issue related to the concretizations. Let us assume that our program works over a string s. If both abstract domains are used and share the underlying numerical abstract domain, this latter domain will contain variables $\underline{\mathsf{len}}(s)$ and $\underline{\mathsf{ord}}(s)$. The concretization of the string length domain will provide a concrete state over s and $\underline{\mathsf{ord}}(s)$ (since it preserves variables it does not handle). The concretization of the string summarization domain will provide a concrete state over s and $\underline{\mathsf{len}}(s)$. In their current form, the concretizations cannot be combined. It would be tempting the drop the auxiliary variables and intersect the resulting concrete state, but this approach introduces imprecisions, as shown in the example below.

Example 2.80 **Imprecise concretization**

Let us consider the following numerical abstract state:

$$\sigma^\sharp = 0 \leq \underline{\mathsf{ord}}(s) - 97 < \underline{\mathsf{len}}(s) \wedge 1 \leq \underline{\mathsf{len}}(s) \leq 2$$

The numerical concretization is

$$\gamma_{\mathsf{num}}(\sigma^\sharp) = \begin{pmatrix} \underline{\mathsf{len}}(s) \\ \underline{\mathsf{ord}}(s) \end{pmatrix} \in \left\{ \begin{pmatrix} 1 \\ 97 \end{pmatrix}, \begin{pmatrix} 2 \\ 97 \end{pmatrix}, \begin{pmatrix} 2 \\ 98 \end{pmatrix} \right\}$$

The string length concretization yields

$$\gamma_{\mathsf{len}}(\sigma^\sharp) = \begin{pmatrix} s \\ \underline{\mathsf{ord}}(s) \end{pmatrix} \in \left\{ \begin{pmatrix} c_0 \\ 97 \end{pmatrix}, \begin{pmatrix} c_0c_1 \\ 97 \end{pmatrix}, \begin{pmatrix} c_0c_1 \\ 98 \end{pmatrix} \;\middle|\; (c_0, c_1) \in [0, 127]^2 \right\}$$

The string summarization domain returns

$$\gamma_{\mathsf{str}}(\sigma^\sharp) = \begin{pmatrix} \underline{\mathsf{len}}(s) \\ s \end{pmatrix} \in \left\{ \begin{pmatrix} 1 \\ w_1 \end{pmatrix}, \begin{pmatrix} 2 \\ w_2 \end{pmatrix}, \;\middle|\; w_1 \in a^*, w_2 \in (a|b)^* \right\}$$

In particular, we consider the string $s =$ "b". We have $\underline{\mathsf{ord}}(s) = 98$, $\underline{\mathsf{len}}(s) = 1$, thus $\underline{\mathsf{ord}}(s) - 97 \not< \underline{\mathsf{len}}(s)$. However, we have:

$$\begin{pmatrix} \text{"}b\text{"} \\ 97 \end{pmatrix} \in \gamma_{\mathsf{len}}(\sigma^\sharp) \qquad \wedge \qquad \begin{pmatrix} 2 \\ \text{"}b\text{"} \end{pmatrix} \in \gamma_{\mathsf{str}}(\sigma^\sharp)$$

Thus, the string concretizations cannot be combined.

It would always be possible to define a concretization for the product of the string abstract domains, which would take into account both kind of constraints from the different auxiliary variables. This definition would however be complicated and not reuse the previous concretizations of each abstract domain. In the current approach, the concretizations are not modular, and cannot be easily combined. This does not fit with our approach based on small, specialized domains, delegating operations to other domains, used to improve precision of relational analyses (cf. Remark 2.70).

2.5 Defining modular concretization functions

The objective of this section is to present a new approach, allowing to define concretization functions modularly, while keeping them precise even when relational abstract domains are used.

2.5.1 Generic approach

Given an abstract domain $\mathcal{D}^\sharp$, its concretization $\gamma_{\mathcal{D}}(\sigma^\sharp_{\mathcal{D}} \in \mathcal{D}^\sharp)$ is a relation between a set of concrete input states and a set of concrete output states.

$$\gamma_{\mathcal{D}}(\sigma^\sharp_{\mathcal{D}} \in \mathcal{D}^\sharp) \in \mathcal{P}\left(\mathcal{P}(\mathcal{D}_{\mathsf{in}}) \times \mathcal{P}(\mathcal{D}_{\mathsf{out}})\right)$$

In the examples of the string length domains, the concrete input state is purely numeric, $\mathcal{D}_{\mathsf{in}} = \mathcal{V} \to \mathbb{Z}$, and the concrete output state also references strings, i.e. $\mathcal{D}_{\mathsf{out}} = \mathcal{V} \to (\mathbb{Z} \cup \mathcal{A}^*)$.

Definition 2.81 **Relation projection operator**

Let $R \in \mathcal{P}(\mathcal{P}(\mathcal{D}_{\text{in}}) \times \mathcal{P}(\mathcal{D}_{\text{out}}))$. We define relation projection operator, written $\downarrow$, as

$$\downarrow R = \{\, o \mid (i, o) \in R \,\}$$

Definition 2.82 **Application of a concretization to another**

Let us assume that we have two domains A and B, with $\mathcal{D}^A_{\text{in}} = \mathcal{D}^B_{\text{out}}$.

$$\gamma_A(\sigma^\sharp_A) \in \mathcal{P}\left(\mathcal{P}(\mathcal{D}^A_{\text{in}}) \times \mathcal{P}(\mathcal{D}^A_{\text{out}})\right)$$
$$\gamma_B(\sigma^\sharp_B) \in \mathcal{P}\left(\mathcal{P}(\mathcal{D}^B_{\text{in}}) \times \mathcal{P}(\mathcal{D}^B_{\text{out}})\right)$$

Intuitively, the application of concretization B to A consists in taking the image of A's concretization, starting from concrete elements in the output concrete state of B. The formal definition is defined below.

$$\gamma_{\substack{A \\ | \\ B}}\begin{pmatrix}\sigma' \\ | \\ \sigma\end{pmatrix} = \{\, (\Sigma, \Sigma') \in \gamma_A(\sigma') \mid \Sigma \subseteq \downarrow \circ \gamma_B(\sigma) \,\}$$

In particular, this application is a subset of the relations defined by γ_A.

The composition of concretization, which yields a relation between $\mathcal{D}^B_{\text{in}}$ and $\mathcal{D}^A_{\text{out}}$, will be covered in the next chapter.

Remark 2.83 **Concretizations of leaf domains**

Leaf domains, such as numerical domains, have concretizations that do not represent relations. In order to unify signatures, we can see those concretizations as relations between the unit value `()` (or the empty set) and the actual concretized state. We introduce an artificial lifting operator, $\uparrow_{\texttt{unit}}$, creating thse dummy relations.

$$\uparrow_{\texttt{unit}}: \begin{cases} \mathcal{P}(\mathcal{D}) & \to & \mathcal{P}(\mathcal{P}(\emptyset) \times \mathcal{P}(\mathcal{D})) \\ \Sigma & \mapsto & \{\, (\emptyset, \{\, \sigma \,\}) \mid \sigma \in \Sigma \,\} \end{cases}$$

2.5.2 String summary domain

Definition 2.84 **Modular string summary concretization**

The modular string summary concretization is a relation between sets of purely numerical concrete states with a single concrete state over integers and strings. It reuses the constraints defined in the original concretization (Definition 2.77), that is:

- Σ should be consistent on variables not handled by the domain. On these variables, the state is the same in the input and output.
- For auxiliary variables $\underline{\text{ord}}(v)$, $\sigma(v)$ will be a string of any length, where all characters are picked in the set $\Sigma(\underline{\text{ord}}(v))$.

$$\begin{aligned}\gamma_{str}(()) = \{\, \Sigma, \{\, \sigma \,\} \mid\ & \Sigma \in \mathcal{P}(\mathcal{V} \to \mathbb{Z}), \sigma \in \mathcal{V} \to \mathsf{Value}, \\ & v \in \text{dom}\Sigma \setminus \{\, \underline{\text{ord}}(\cdot) \,\} \Leftrightarrow (|\Sigma(v)| = 1 \land \sigma(v) = \Sigma(v)); \\ & \underline{\text{ord}}(v) \in \text{dom}\Sigma \Leftrightarrow \exists n \in \mathbb{N}, \exists (c_0, \ldots, c_{n-1}) \in \mathcal{A}^n, \\ & \qquad (\sigma(v) = c_0 \cdots c_{n-1} \land \forall i \in [0, n-1], c_i \in \Sigma(\underline{\text{ord}}(v))) \,\}\end{aligned}$$

Example 2.85 **String summarization concretization**

We go back to the example where $\sigma^\sharp = 97 \leq \underline{\mathsf{ord}}(s) < i \leq 100$. We search for the concretization of this abstract state using the string summary domain, i.e., we compute the application as defined in Definition 2.82.

$$\gamma_{\substack{\mathsf{str}\\|\\\mathsf{num}}}\begin{pmatrix}()\\|\\\sigma^\sharp\end{pmatrix} = \{\,(\Sigma, \{\,\sigma\,\}) \in \gamma_{\mathsf{str}}(())\mid \Sigma \subseteq \gamma_{\mathsf{num}}(\sigma^\sharp)\,\}$$

In particular, the following elements are part of the application:

$$\left\{\begin{pmatrix}98\\97\end{pmatrix}\right\}, \left\{\begin{pmatrix}98\\c\end{pmatrix}\;\middle|\; w \in a^*\right\};$$
$$\left\{\begin{pmatrix}99\\97\end{pmatrix}; \begin{pmatrix}99\\98\end{pmatrix}\right\}, \left\{\begin{pmatrix}99\\w\end{pmatrix}\;\middle|\; w \in (a|b)^*\right\};$$
$$\left\{\begin{pmatrix}100\\97\end{pmatrix}; \begin{pmatrix}100\\98\end{pmatrix}; \begin{pmatrix}100\\99\end{pmatrix}\right\}, \left\{\begin{pmatrix}100\\w\end{pmatrix}\;\middle|\; w \in (a|b|c)^*\right\}$$

2.5.3 String length domain

Contrary to the string summarization domain, the string length one does not require combining multiple concrete input states to create output states. We can thus define a simplified concretization over pairs of concrete states, which will be lifted afterward.

Definition 2.86 **Simplified string length concretization**

The concretization transforms an input concrete state, which is purely numerical, into an output concrete state, where integers and strings are used. Bindings of variables that are not defined by the domain are kept as-is. Given an auxiliary variable $\underline{\mathsf{len}}(v)$, the output concrete state binds v to any string having length $\rho(\underline{\mathsf{len}}(v))$.

$$\begin{aligned}\gamma_{\mathsf{len}}(())= \{\,(\rho,\rho') \mid\ & \rho \in \mathcal{V} \rightharpoonup \mathbb{Z}, \rho' \in \mathcal{V} \to \mathsf{Value}\\ & v \in \mathsf{dom}\rho \setminus \{\,\underline{\mathsf{len}}(\cdot)\,\} \Leftrightarrow \rho'(v) = \rho(v);\\ & \underline{\mathsf{len}}(v) \in \mathsf{dom}\rho \Leftrightarrow \rho'(v) \in \mathcal{A}^{\rho(\underline{\mathsf{len}}(v))}\,\}\end{aligned}$$

Definition 2.87 **One-to-many lifting operator**

$\uparrow$ transforms a set of pairs into a pair of sets respecting membership of the initial pairs:

$$\uparrow: \left\{\begin{array}{ccl}\mathcal{P}(A\times B) & \to & \mathcal{P}(\mathcal{P}(A)\times\mathcal{P}(B))\\ X & \mapsto & \{\,\Sigma_a, \Sigma_b \mid \forall\sigma_a \in \Sigma_a, (\sigma_a,\sigma_b) \in X \Leftrightarrow \sigma_b \in \Sigma_b\,\}\end{array}\right.$$

For example, if $X = \{\,(1,'a');(2,'b')\,\}$, we have

$$\uparrow X = \{\,(\{\,1\,\},\{\,'a'\,\});(\{\,2\,\},\{\,'b'\,\});(\{\,1;2\,\},\{\,'a';'b'\,\})\,\}$$

Definition 2.88 **Full, relational string length concretization**

The relational string length concretization, written $\gamma_{\dot{\mathsf{len}}}$, is the lifting of the simplified con-

cretization defined before.

$$\gamma_{\substack{\dot{\text{len}} \\ | \\ \text{num}}} = \uparrow \circ \gamma_{\text{len}}$$

Example 2.89 **String length concretization**

Keeping the abstract state $\sigma^\sharp$ of Example 2.61, its concretization is

$$\gamma_{\text{num}}(\sigma^\sharp) = \{\, (\underline{\text{len}}(s) \mapsto 4, \underline{\text{len}}(t) \mapsto 4, l \mapsto v_l) \mid 0 \leq v_l \leq 128 \,\}$$

The application of this numerical state on the string length concretization yields:

$$\gamma_{\substack{\dot{\text{len}} \\ | \\ \text{num}}} \begin{pmatrix} () \\ | \\ \sigma^\sharp \end{pmatrix} = \uparrow \Big(\Big\{ (\underline{\text{len}}(s) \mapsto 4, \underline{\text{len}}(t) \mapsto 4, l \mapsto v_l), \\ (s \mapsto c_0 \cdots c_3, t \mapsto c_4 \cdots c_7, l \mapsto v_l) \;\Big|\; 0 \leq v_l \leq 128, (c_0, \cdots, c_7) \in [0, 127]^8 \Big\} \Big)$$

2.5.4 Combining both concretizations

When both string domains are used in a product, we can now easily define the concretization of the latter as the intersection of the relations defined by the concretization of the string domains. We show how the abstract state that was not precisely concretized in the usual framework (in Example 2.80) is concretized in our case.

Example 2.90 **Concretization of the product of string domains**

Let us illustrate the concretizations, starting from the following abstract state:

$$\sigma^\sharp = 0 \leq \underline{\text{ord}}(s) - 97 < \underline{\text{len}}(s) \wedge 1 \leq \underline{\text{len}}(s) \leq 2$$

$$\gamma_{\text{num}}(\sigma^\sharp) = \begin{pmatrix} \underline{\text{len}}(s) \\ \underline{\text{ord}}(s) \end{pmatrix} \in \left\{ \begin{pmatrix} 1 \\ 97 \end{pmatrix}, \begin{pmatrix} 2 \\ 97 \end{pmatrix}, \begin{pmatrix} 2 \\ 98 \end{pmatrix} \right\}$$

$$\gamma_{\substack{\dot{\text{len}} \\ | \\ \text{num}}} \begin{pmatrix} () \\ | \\ \sigma^\sharp \end{pmatrix} = \uparrow \left(\left\{ \begin{pmatrix} 1 \\ 97 \end{pmatrix}, \begin{pmatrix} c \\ 97 \end{pmatrix} \;\middle|\; c \in \mathcal{A} \right\} \cup \left\{ \begin{pmatrix} 2 \\ 97 \end{pmatrix}, \begin{pmatrix} c \\ 97 \end{pmatrix} \;\middle|\; c \in \mathcal{A}^2 \right\} \right. \\ \left. \cup \left\{ \begin{pmatrix} 2 \\ 98 \end{pmatrix}, \begin{pmatrix} c \\ 98 \end{pmatrix} \;\middle|\; c \in \mathcal{A}^2 \right\} \right)$$

$$\gamma_{\substack{\text{str} \\ | \\ \text{num}}} \begin{pmatrix} () \\ | \\ \sigma^\sharp \end{pmatrix} = \left\{ \left\{ \begin{pmatrix} 1 \\ 97 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 1 \\ c \end{pmatrix} \right\} \;\middle|\; c \in a^* \right\} \cup \left\{ \left\{ \begin{pmatrix} 2 \\ 97 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 2 \\ c \end{pmatrix} \right\} \;\middle|\; c \in a^* \right\} \\ \cup \left\{ \left\{ \begin{pmatrix} 2 \\ 98 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 2 \\ c \end{pmatrix} \right\} \;\middle|\; c \in b^* \right\} \cup \left\{ \left\{ \begin{pmatrix} 2 \\ 97 \end{pmatrix}, \begin{pmatrix} 2 \\ 98 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 2 \\ c \end{pmatrix} \right\} \;\middle|\; c \in (a|b)^* \right\}$$

The second coordinate of the states concretized by the string domains have different mapping domains: the length one has $(s, \underline{\text{ord}}(s))$, and the summarization one has $(\underline{\text{len}}(s), s)$. We define $\cap_s$ the intersection keeping only common variables.

We can now intersect the applications of the concretizations:

$$\gamma_{\substack{\dot{\text{len}} \wedge \text{str} \\ | \\ \text{num}}} = \gamma_{\substack{\dot{\text{len}} \\ | \\ \text{num}}} \cap_s \gamma_{\substack{\text{str} \\ | \\ \text{num}}}$$

Contrary to Example 2.80, the intersection is now guided by the input concretized state, which will allow us to be precise. For the input $\left\{ \begin{pmatrix} 1 \\ 97 \end{pmatrix} \right\}$, the intersection of the second coordinates yields:

$$\left\{ \begin{pmatrix} s \\ \underline{\mathsf{ord}}(s) \end{pmatrix} \mapsto \begin{pmatrix} c \\ 97 \end{pmatrix} \;\middle|\; c \in \mathcal{A} \right\} \cap_s \left\{ \begin{pmatrix} \underline{\mathsf{len}}(s) \\ s \end{pmatrix} \mapsto \begin{pmatrix} 1 \\ a^n \end{pmatrix} \;\middle|\; n \in \mathbb{N} \right\} = \{ s \mapsto a \}$$

Given $\left\{ \begin{pmatrix} 2 \\ 97 \end{pmatrix}, \begin{pmatrix} 2 \\ 98 \end{pmatrix} \right\}$ as input, the second coordinate is:

$$\begin{aligned}
&\left\{ \begin{pmatrix} s \\ \underline{\mathsf{ord}}(s) \end{pmatrix} \in \left\{ \begin{pmatrix} c \\ 97 \end{pmatrix} \;\middle|\; c \in \mathcal{A}^2 \right\}, \left\{ \begin{pmatrix} c \\ 97 \end{pmatrix} \;\middle|\; c \in \mathcal{A}^2 \right\} \right\} \cap_s \left\{ \begin{pmatrix} \underline{\mathsf{len}}(s) \\ s \end{pmatrix} \mapsto \begin{pmatrix} 2 \\ c \end{pmatrix} \;\middle|\; c \in (a|b)^* \right\} \\
&= \{ s \mapsto c \mid c \in \mathcal{A}^2 \wedge c \in (a|b)* \} \\
&= \{ s \mapsto c \mid c \in \{ aa, ab, ba, bb \} \}
\end{aligned}$$

Hence, we find that the concretized state of $\sigma^\sharp$ is a, aa, ab, ba, bb.

2.6 Conclusion

We have defined a concrete, but uncomputable collecting semantics for the `Imp` language. In order to define computable analyses, we have used abstractions, such as the one from the set of integers to intervals, or the summarization of strings content. We also have defined a loop approximation method that terminates and relies on a specific widening operator. We have shown how ghost variables can be used to delegate work to underlying domains, which is especially helpful to improve the precision of the analysis when using relational numerical domains. We have developed a new framework to define modular concretizations, in line with the modular definitions of the abstract domains we used. This delegation-based approach is one of the core concepts used in Mopsa, the static analyzer used in this thesis and presented in the next chapter.

Part II

Base Abstractions

3

Mopsa

As we have seen in the previous chapter, combining abstract domains is key to improving the precision of a relational analyzer. We have also studied domains delegating part of their operations to an underlying domain (such as the string length domain of Section 2.4.2 relying on a numerical domain). These design decisions also ease their implementation.

This chapter presents the Mopsa static analyzer, in which the analyses described in this thesis will be implemented. Mopsa stands for Modular Open Platform for Static Analysis. It is written in OCaml. One of the goals of Mopsa is to let developers define abstract domains modularly (i.e., as independently from each other as possible), while allowing them to cooperate and communicate. Some of these abstract domains may be relational, and we want to allow relational communication between them. Thus, expressions are evaluated into other expressions rather than abstract values. In addition, multiple domains can be composed on top of an underlying domain, resulting in combinations representable as directed acyclic graphs, while other static analyzers rely on trees of abstract domains. For example, the string length and string summarization domain used in Example 2.90 share an underlying relational numeric domain, allowing to express relationships between the lengths and contents of strings. A schematic representation of this example is provided in Figure 3.2. Another goal of Mopsa is to support the analysis of multiple languages (it currently targets C and Python). Using this approach, we can define our analyses in layers, which also eases implementation and maintenance. Some of the innermost layers are shared between the languages, thus factoring the codebase and simplifying support for new languages. Analyses are defined using configuration files, describing how abstract domains are combined. When invoked, Mopsa parses the configuration file, dynamically instanciates the abstract domains and constructs the abstract state from the configuration. Each analyzed expression (or statement) flows from the top domain of the configuration to the bottom until one domain handles it.

We start by showing the design choices of some state-of-the-art static analyzers, and compare to our design in Mopsa. We then describe the abstract syntax tree (AST) on which Mopsa operates, the domains' signature, the notion of hooks, acting as observers of the analysis. We finish by defining modular concretizations of abstract domains from a theoretical standpoint. Sections 3.2 and 3.3 are close to a previously published description of Mopsa [82].

3.1 Related work

We start with a brief overview of related static analyzers. We take a look at Infer, TAJS, Frama-C's Eva, and Astrée. Analyzers related to Python or performing multilanguage analyses will be

described in Sections 7.7 and 11.7 respectively. Except for Infer, the analyzers described here focus on a single language.

3.1.1 Infer

Infer [48, 45] is a static analyzer developed by Facebook, which aims at helping developers find bugs in their commits during continuous integration. Thus, its main objective is to be fast and have a low false-positive rate. Soundness is not often sought in the analyses defined.

Infer can analyze Java and C/C++/Objective-C programs. Programs written in these supported languages are statically translated into a reduced intermediate representation called SIL, using four instructions[1]. Infer performs an on-demand bottom-up analysis using procedure summaries, and relies on a CFG representation to iterate over nodes. Different analyses can be defined using the provided framework. Some analyses may reuse others' results, but they cannot mutually cooperate as a reduced product does.

Each analysis can choose its iteration strategy (forward or backward), initial starting abstract value, as well as the representations of function summaries. It is thus difficult to provide a big picture representation of what Infer's analyses generally do. We thus study three different analyses below, all of which work by forward iterations. Given an abstract domain and its transfer function over SIL statements, Infer can lift an intraprocedural analysis into an interprocedural one. It is also possible to define interprocedural analyses and custom summaries directly.

- ▷ **InferBO** is used to detect out-of-bound accesses in arrays[2]. It relies on a symbolic interval domain. Function summaries consist in necessary conditions that need to be satisfied to avoid out-of-bound accesses[3].
- ▷ **RacerD** detects data races in programs. It does not rely on numerical domains. Example summaries are shown [13, page 4].
- ▷ **Pulse** performs an underapproximating analysis aiming at detecting memory errors [126]. Summaries[4] consist of a pair of pre/post symbolic memory states combined with a numerical interval domain. The pre state describes a refinment of the entry state that leads to the current program point being analyzed. The post state describes the state at the current program point.

Mopsa also targets multiple languages. Infer operates on a small intermediate language. A benefit of this approach is to reduce the number of constructions that need to be analyzed. Supporting a new language is only a matter of adding a static translation from the source language to the intermediate representation. This approach has several downsides: the translation into the intermediate language may reduce the precision of the analyses, and the intermediate language may not be appropriate for every programming language. For example, Infer does not support any dynamic programming language such as Python or JavaScript and its intermediate representation may need changes to precisely handle features such as dynamic typing or asynchronous computations. Mopsa currently supports only fully context-sensitive approaches, compared to the bottom-up function analysis of Infer. It relies on relational numerical abstract domains to be precise. Its goal is to combine domains rather than have specialized independent checkers.

[1] `https://github.com/facebook/infer/blob/v1.1.0/infer/src/IR/Sil.mli#L40`

[2] `https://research.fb.com/inferbo-infer-based-buffer-overrun-analyzer/`

[3] `https://github.com/facebook/infer/blob/v1.1.0/infer/src/bufferoverrun/bufferOverrunDomain.ml#L1910`

[4] `https://github.com/facebook/infer/blob/v1.1.0/infer/src/pulse/PulseAbductiveDomain.mli#L58`

3.1.2 TAJS

TAJS [74] is a sound static analyzer for JavaScript programs, aiming at detecting type-related errors. It performs a context-sensitive, CFG-based analysis. Similarly to Infer, TAJS relies on a simplified intermediate representation with around 20 instructions [74, page 7]. TAJS relies on a constants domain for numerical values.

Jensen et al. [75] improve the interprocedural analysis by making opaque the fields of objects passed to functions. If specific fields are required during the analysis of the function, they can be recovered. If an object's field is opaque in a function, but its value is changed outside of the function, there is no need to analyze the function again, reducing the cost of the interprocedural analysis.

A prevalent usecase of JavaScript is to interact with HTML pages, possibly to react to user input. A specific modeling of these interactions has been proposed by Jensen et al. [76].

JavaScript programs often use the `eval` function to interpret the passed string as a JavaScript program and evaluate it. Jensen et al. [77] propose an approach to remove calls to `eval` where the passed string is semantically constant, through a constant propagation.

Stein et al. [142] added backward evaluation mechanisms in TAJS to improve the precision at critical points and gain back relational information during unknown property accesses. Another approach for improving precision is the notion of value partitioning introduced by Nielsen and Møller [119], where an abstract value domain is locally partitioned given a , such as an imprecise read of a property.

Kristensen and Møller [87] define a notion of reasonably most-general client for the analysis of TypeScript libraries. They then leverage TAJS to analyze the libraries using this client and detect errors in the type annotations of the libraries.

Contrary to TAJS, Mopsa aims at performing relational numerical analyses, and supports the analysis of multiple languages. It would be interesting to extend the interprocedural optimizations of Jensen et al. [75] to a language-agnostic approach in Mopsa. Although Mopsa targets the analysis of Python programs, it does not support the `eval` statement for now, which seems less used than in JavaScript. Mopsa does not support the standalone analysis of libraries for now.

3.1.3 Frama-C

Frama-C provides a platform for C program analysis and verification. It features different plug-ins, allowing to perform deductive verification, runtime verification, or static analysis. The first static analysis plug-in of Frama-C was Value, a non-relational monolithic abstract domain. It has been upgraded by Bühler [22] [15] to Eva, which provides a more modular architecture adding new abstract domains (including an equality domain). Eva also provides better cooperation and communication mechanisms between the different supported abstractions. Eva performs sound, whole-program, context-sensitive analyses. Recursive functions are not supported.

Eva performs forward iterations over the CFG of the program, where a modified C intermediate representation is used (loops are transformed into infinite `while` with `break` statements, functions have a single `return` statement, the ternary conditional operator is explicitly rewritten – cf. [22, Section 4.1.2]).

Given a set of abstract domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$ defining an analysis, an expression e is evaluated into value abstractions $v_1, \ldots, v_n$ for all those domains. Value abstractions abstract a set of values and have a lattice structure. Domains communicate through these abstract values, and reduce the evaluation of e to $v_1 \sqcap^\sharp \ldots \sqcap^\sharp v_n$.

In its current form, Eva relies on a main abstract domain called Cvalue (the previous monolithic abstract domain of its predecessor plug-in). Additional domains can be activated, such

as a symbolic equalities domain. Bindings to Apron's [73] relational abstract domains are available but only work on program variables declared as integers.

Frama-C is more stable and mature than Mopsa. It has been successfully used in industrial settings, for example to analyze the code of French nuclear power plants [122]. Mopsa aims at creating more collaborative, relational analyses than what can be currently expressed in Frama-C.

3.1.4 Astrée

Astrée is a static analyzer of C programs specialized in the certification of critical embedded software. It performs a sound, whole-program, context-sensitive analysis. It does not handle recursive functions, but supports dynamic memory allocation. Astrée has been recently upgraded to provide more modular abstractions through the work of Chevalier [27].

Astrée's analysis proceeds by induction on the syntax. The structure of Astrée's domain is described in [105, page 61], [27, page 168], and the upgraded version [27, page 298]. A topmost domain handles trace partitioning. An environment domain follows; it handles non-local control-flow. A struct domain rewrites C variables and dereferences into cell accesses [107]. The base-offset domain handles pointers by delegating its operations to the shared numerical domain. This approach relying on the cell abstraction and a base-offset domain is reused in the analysis of C programs in Mopsa [121]. A combination of relational and non-relational numerical domains is used at the bottom of the analyzer.

A notable improvement of the new version is the use of ghost variables to define program properties that can be shared in a (relational) numerical domain to improve precision. These ghost variables are used to represent pointers offsets or slices of variables whose values are bitwise manipulated. A whole theoretical framework has been developed by Chevalier and Feret [28] to ensure that analyses using ghost variables are sound and terminate. In all cases, ghost variables build upon variables, be it program variables or ghost variables themselves. The key dispenser, translator, and driver domains [27, page 298] act as centralized domains handling these ghost variables. Mopsa also relies on auxiliary variables to define some program properties in underlying, potentially relational, numerical domains.

Contrary to Astrée, each domain is responsible for its auxiliary variables in Mopsa. In Mopsa, and in particular in the analysis of Python programs, auxiliary variables are also used on abstract addresses.

Astrée is much more mature than Mopsa. It has been successfully applied for the verification of safety-critical software, such as the control command of Airbus planes [43], and has been commercialized by AbsInt since 2009. In addition, some versions of Astrée support backward analyses [131], allowing to refine an input state from which runtime errors were detected, and rule out some false alarms. Astrée also supports the analysis of multi-threaded programs to detect for example data races [109, 105]. The memory model used is compatible with PSO. Astrée has been recently extended by Chevalier [27] to support the analysis of snippets of assembly language embedded in C code.

Mopsa explores a different design space, with the goal of being more modular and extensible than Astrée. It is also not specialized to a specific input language, and has a different architecture.

3.1.5 Framework of Keidel et al.

Keidel et al. [84] defined an approach to perform soundness proofs locally, on each abstract domain. Their approach make soundness proofs easier and lighter. Keidel and Erdweg [83] define modular abstract domains that can be combined by transformers, where the soundness proofs of their previous work apply. They however make the assumption that abstract domains are non-relational maps from variable to abstract values. In constrat, one of the

main goals of Mopsa is to perform relational analyses. We have already seen in the previous chapter that defining analyses and concretizations in a relational setting is significantly more complicated. For example, the consistency criterion on concrete input states in the case of the string summarization concretization (Definition 2.77) was required to handle the case of relational domains. In the case of non-relational domains, this property is always ensured.

3.2 Abstract syntax tree (AST)

Similarly to Astrée, Mopsa performs an analysis by induction on the syntax, rather than by iterating over a control-flow graph (the approaches were compared in Remark 2.68). Mopsa uses a single AST, which is extended when needed to support new languages. This way, it keeps the high-level structure of parsed programs. A downside of this approach is the high number of AST nodes an analysis needs to support. This is mitigated by dynamic translations from language-specific AST nodes to language-agnostic ones (e.g., Python and C loops shown in Section 3.2.4), which will be analyzed by so-called "universal" abstract domains. For now, Mopsa supports the following languages :

- a large subset of Python 3, using a dedicated parser originally developed by Fromherz et al. [54],
- most of C, using Clang's parser,
- a contract annotation language close to ACSL [7] used to model library functions for C programs [121],
- universal, a simple language close to the one described in Chapter 2.

We illustrate the inner workings of our approach on the case of loop statements. We start by defining elementary expressions and statements used in our examples (including support for basic while loops of `Imp`). We explain how a domain would compute an approximate fixpoint for these loops. We extend our approach to the cases of Python and C loops.

3.2.1 Elementary expressions and statements

The syntactic elements of Mopsa include variables, constants, types, binary operators, expressions and statements. All these types are defined as records consisting of multiple fields. The main field of interest is the kind, which is an extensible variant type, meaning that any OCaml module can further extend it. Additional fields include the program ranges for expressions and statements, and the type for variables and expressions.

Definition 3.1 **The `var` type**

Variables are defined in Mopsa using the code shown in Listing 3.1. The type of variables, `var`, is defined using four different fields:

▷ **vname,** which is the unique name of the variable;
▷ **vkind,** the kind of the variable, which should be an inhabitant of `var_kind`;
▷ **vtyp,** the type of the variable;
▷ **vmode,** the mode of the variable (weak or strong).

We show a base variant of the `var_kind` type. This variant is used for original variables present in the analyzed program. `V_uniq` represents variables identified by a unique integer identifier, and having a string identifier (coming from the source code and used for printing).

Listing 3.1: Declaration of base variables

```
type var_kind = ..

type var = {
  vname : string;
  vkind : var_kind;
  vtyp  : typ;
  vmode : mode;
}

type var_kind +=
  (** Variables identified by their unique id *)
  | V_uniq of string (** Original name *) *
              int    (** Unique ID *)
```

Definition 3.2 **The `expr` type**

Expressions are defined in Listing 3.2. They rely on a kind (an extensible variant), a type and a range of program locations from which the expression comes. The variant type `expr_kind` can be initially defined with the following variants :

- ▷ **`E_var`,** for variables.
- ▷ **`E_constant`,** denoting constants (which is also an extensible type, not detailed here).
- ▷ **`E_not`,** for the boolean negation of an expression.
- ▷ **`E_binop`,** describing a binary operation (`operator` is also extensible) over two expressions.

Listing 3.2: Declaration of universal expressions

```
type expr_kind = ..

type expr = {
  ekind: expr_kind;
  etyp: typ;
  erange: Location.range;
}

type expr_kind +=
  | E_var of var * mode option
  | E_constant of constant
  | E_not of expr
  | E_binop of operator * expr * expr
```

Definition 3.3 **The `stmt` type**

The declaration of the `stmt` type is shown in Listing 3.3. Statements are defined by a kind and a range of program locations. The first four variants defined (`S_add`, `S_remove`, `S_expand`, `S_fold`) are used to modify the domain (of variables, or addresses), on which an abstract domain is defined. `S_assume` is used to filter the abstract state so that the provided condition is satisfied. The last three variants are more usual. `S_assign` expresses an assignment between two expressions. `S_block` defines a sequence of statements. `S_while` defines a while loop.

Listing 3.3: Declaration of universal statements

```
type stmt_kind = ..

type stmt = {
```

```
  skind : stmt_kind;
  srange : Location.range;
}

type stmt_kind +=
  | S_add of expr
  | S_remove of expr
  | S_expand of expr * expr list
  | S_fold of expr * expr list

  | S_assume of expr

  | S_assign of expr * expr
  | S_block of stmt list
  | S_while of expr * stmt
```

Remark 3.4 **Availability of newly added variants**

When a kind is extended with a new variant, the variant becomes available everywhere. Particularly, it can be used as subnodes of instructions defined previously.

For example, we can extend expression with `E_subscript` to denote index accesses in strings (as defined in the previous chapter).

```
type expr_kind += E_subscript of expr * expr
```

Any expression can now use `E_subscript`.

Remark 3.5 **Ghost variables**

The variant `V_var_attr` of `var_kind` is shown in Listing 3.4. It is used to define ghost variables on top of other variables. For example, one way to define the ghost variable $\underline{\text{len}}(s)$ of Section 2.4.2 is through `V_var_attr (s, "len")`.

Listing 3.4: Variant for auxiliary variables built on top of other variables

```
type var_kind +=
  | V_var_attr of var    (** Attach variable *)
                * string (** Attribute *)
```

Remark 3.6 **Domains handling ghost variables**

Contrary to Astrée where ghost variables are handled centrally by one domain, each domain is responsible for its ghost variables. For example, if the variable s is deleted through the statement `S_remove s`, it is up to the string length domain to translate this statement into the deletion of $\underline{\text{len}}(s)$ (i.e., by executing `S_remove (V_var_attr (s, "len"))`). We show the detailed implementation of this case in Remark 3.21. Another difference with Astrée is that ghost variables can also be built upon abstract addresses, which are introduced by dynamic memory allocation abstractions, such as the recency abstraction introduced in Chapter 4.

3.2.2 A domain handling while loops

We show excerpts of the domain computing loop fixpoints in Listing 3.5. The next section (3.3) will explain in detail the signature of abstract domains. `exec` defines the local transfer function over statements for this domain. It takes three arguments: the statement to analyze, a manager referencing the global transfer functions, and the analysis' state.

The transfer function of `S_while` calls the `lfp` function. Then, it calls recursively the whole analysis (using `man.exec`, described later in Section 3.3.2.2) to filter the reached fixpoint by the negation of the loop's condition.

`lfp` performs an accelerated fixpoint computation. It starts by computing the effects of the loop's condition and body on the input state, by calling recursively the whole analysis. `mk_block` and `mk_assume` are helper functions creating statements whose `stmt_kind` is respectively `S_block` and `S_assume`. It the result is stable, we return the previously computed state[5]. Otherwise, `lfp` calls itself recursively, on a widened state.

We can intuitively notice that the analysis progresses: recursive calls are performed on strict sub-nodes of the `S_while` statement. This argument would also be used to prove termination of the analysis.

Listing 3.5: Excerpts of the domain computing fixpoint for the `S_while` statement

```
let rec lfp man cond body flow_init flow =
  let flow' = man.exec (mk_block [mk_assume cond; body]) flow in
  if man.lattice.subset (man.lattice.join flow_init flow') flow then flow'
  else lfp man cond body flow_init (man.lattice.widen flow flow')

let exec stmt man flow = match stmt_kind stmt with
  | S_while (cond, body) ->
    let lfp_flow = lfp man cond body flow flow in
    Some (man.exec (mk_assume (mk_not cond)) lfp_flow)
  | _ -> None
```

3.2.3 Extending the AST with Python and C loops

Remark 3.7 **Convention: color codes of languages**

In the rest of this thesis, we use the following color code convention: Universal constructors are written in blue, Python ones in green, and C ones in orange. The previous variants of `expr_kind` and `stmt_kind` were all part of Universal.

The Python frontend adds two new loop nodes to the statement kind. The additional statement of each loop variant corresponds to the `else` clause of the loops. The detailed semantics of Python loops will be shown in Section 6.2.7.

```
type stmt_kind += S_py_for of expr * expr * stmt * stmt
                | S_py_while of expr * stmt * stmt
```

The C frontend extends statements with:

```
type stmt_kind += S_c_for of stmt * expr option * expr option * stmt
                | S_c_do_while of stmt * expr
```

Standard `while` loops of C are identical to Universal, and thus use the variant of the Universal language. We add the case of `for` and `do-while` loops, instead of translating them directly into `while` loops.

Remark 3.8 **Multiple languages in the AST**

All languages are mixed in the AST. We can thus perform partial rewriting mixing different languages (as shown in Section 3.2.4). It will also be convenient to analyze multilanguage programs in Chapter 11.

[5] Since we return `flow'`, we obtain a free decreasing iteration, as we stated at the end of Section 2.3.8.

3.2.4 Dynamically rewriting Python and C loops

During the analysis, AST nodes can be rewritten into other nodes. This dynamic rewriting is used to perform translations from language-specific constructs to simpler constructs. The case of loops is schematically shown in Figure 3.1. The domain `C.Loops` rewrites C `for` loops into a statement containing a Universal `while` loop. These Universal `while` loops are handled by the domain `Universal.Loops` which computes a fixpoint using widening. Similarly, the domain `Python.Loops` rewrites Python `for` loops into a statement containing a Universal loop, which is then handled by the Universal loop domain. The Python rewriting shown here is the most general one (cf. Section 6.2.7) where an iterator is explicitly constructed and used.

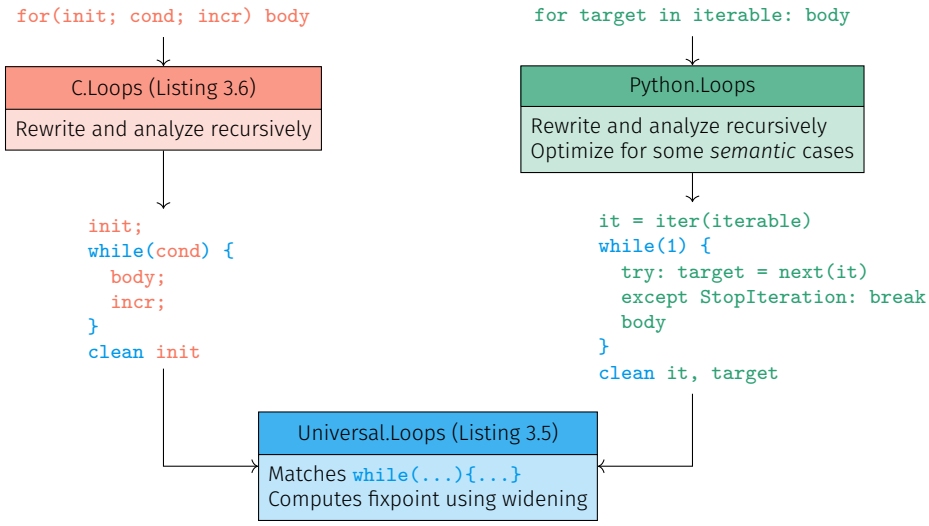

Figure 3.1: Example of dynamic rewriting

The implementation of the transfer function of the C loop iterator as done in Mopsa is shown in Listing 3.6. In both cases, the statements are rewritten into a block of statements containing a universal `S_while` loop. This loop will be handled by the universal loop domain shown in Listing 3.5. In these cases, we can argue that the analysis progresses because the recursive calls will handle a more elementary kind of loop, in a simplified language (`S_while`).

Listing 3.6: Transfer function of C loop iterator

```
let exec stmt man flow = match stmt_kind stmt with
  | S_c_for(init, cond, incr, body) ->
    let new_stmt =
      mk_block [init;
                mk_while(cond,
                         mk_block [body; incr]);
                mk_clean init] in
    Some (man.exec new_stmt flow)
  | S_c_do_while(body, cond) ->
    let new_stmt =
      mk_block [body;
                mk_while(cond, body)] in
    Some (man.exec new_stmt flow)
  | _ -> None
```

Remark 3.9 **Iterators are stateless domains**

In Mopsa, iterators are a simplified case of abstract domains with no local abstract state. They can be put anywhere in the combination of domains defining an analysis. They have access to the abstract state, which can be used to perform semantically-guided rewritings.

Example 3.10 **Semantically-optimized rewriting**

Let us consider the case of Python `for` loops. If the `iterable` is a `range` object, we would like to perform a more natural rewriting with explicit increments on `target`, that can ease the search for precise loop invariants.

However, Python is dynamically typed, so there is no systematic syntactic information we can use to optimize the rewriting in the case of `range`. With a syntactic approach, we could optimize the case with explicit `range` such as `for x in range(10): ...`, but not the case of the loop `for x in f(): ...`, where `f` returns a `range` object.

In our case, the Python loop iterator starts by evaluating `iterable`. At the end of the evaluation, we obtain semantic information on the `iterable`'s type. The iterator can then decide which rewriting to perform, and would work on both cases mentioned before.

3.3 Domains

We start by explaining how Mopsa transforms configuration files into analyses. This construction motivates the need for a unified domain signature, detailed in Section 3.3.2. We study two specific cases: non-relational domains in Section 3.3.3, and the combination of domains into a reduced product in Section 3.3.4. We explain how domains can broadcast specific requests for information to other domains in Section 3.3.5.

3.3.1 Defining analyses by combining domains

Users define analyses in Mopsa through a configuration file describing which domains are used and how they are combined. A configuration is a directed acyclic graph (DAG), where nodes are either domains or domain combinators. Each analyzed expression (or statement) flows from the top domain of the configuration to the bottom until one domain treats it (i.e., it returns a value that is not `None`). A configuration for the analysis of `Imp` programs of Chapter 2 is shown in Figure 3.2. `U.program` is the frontend handling the import from the parser. `U.intraproc` defines the execution of conditionals and sequence of statements. `U.loops` computes an abstract fixpoint for loops. Domains `U.str_len` and `U.str_sum` represent the domains presented in Sections 2.4.2 and 2.4.5 respectively. They are put in a reduced product (as described in Section 2.4.6) and share an underlying numerical domain `U.numeric` handling assignments of numeric variables and conditional filtering over numeric expressions. More involved configurations will be shown in Figures 7.10 and 8.6 for the Python analyses and in Figure 11.9 for the multilanguage analysis.

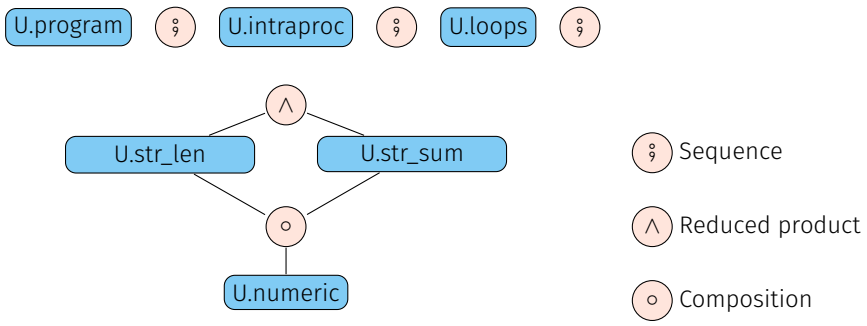

Figure 3.2: Analysis configuration in Mopsa

Domains have a unified signature simplifying their combination. Combinators build new domains, combining the types and transfer functions of input domains. Specifically, the whole analysis behaves as a domain.

The sequence combinator tries to execute the transfer functions of the left-hand side domain. If this domain does not handle the passed statement or expression, the transfer function of domain on the right-hand side of the sequence will be called. In the case of a reduced product, transfer functions on both sides are called before reductions are performed. The compose combinator is an extended case of the sequence combinator[6].

3.3.2 Domain signature

Mopsa's abstract domains abide by the domain signature shown in Listing 3.7. We successively explain each part of the signature in this section.

Listing 3.7: General domain signature

```
(* Section 3.3.2.2 *)
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;

  ask : ('a,'r) query -> 'a flow -> 'r;
  print_expr : 'a flow -> (printer -> expr -> unit);
  get_effects : teffect -> teffect;
  set_effects : teffect -> teffect -> teffect;
}

(* Section 3.3.2.4 *)
type 'a post = ('a, unit) cases
type 'a eval = ('a, expr) cases

module type DOMAIN =
sig
  (* Section 3.3.2.1 *)
  type t
  val id : t id
  val name : string
  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

  (* Section 3.3.2.5 *)
  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

  (* Section 3.3.2.6 *)
  val merge: t -> t * effect -> t * effect -> t
  val ask  : ('a,'r) query -> ('a, t) man -> 'a flow -> 'r option
  val print_state : printer -> t -> unit
  val print_expr  : ('a,t) man -> 'a flow -> printer -> expr -> unit
end
```

[6]The lattice operations are not performed pointwise on both domains, but by performing the operation on the upper domain – potentially modifying the lower domain – and then on the lower domain. This is useful when the upper domain needs to unify states in the lower one, see Remark 3.34.

3.3.2.1 Domain type and lattice operations (lines 24-33)

A domain operates on a type `t`, which is private and opaque to others. Lattice elements and operators of the domain are defined classically. The widening also takes as parameter a context (polymorphic in the whole abstraction), from which it can extract information such as thresholds for the widening (cf. Section 3.4). Each domain has its own identifier and a name.

Remark 3.11 **Domain type and polymorphism**

Since the abstract state is created dynamically by combining domains according to the configuration, the type of the global abstract state is not precisely known during the definition of each domain. The type of the whole combination of domains is thus generalized with the type `'a`.

3.3.2.2 The need for a manager (lines 2-15)

As we have mentioned previously (Section 3.3.1), a configuration is a directed acyclic graph, which defines how domains are combined. Thanks to the unified signature, the whole analysis is also a domain, and internally, a DAG having the same structure as the configuration. During the analysis, statements and expressions traverse the DAG, starting from the root, until one domain handles them.

We explain below how the manager is used to interroperate between a local domain and the overall domain. Since the manager connects the global state with the local domain, its type is `('a, t) man`.

▷ **From the global domain to a local domain (lines 3-4).** Since the combination of domains is defined at runtime, we abstract the type of the whole abstract state into a parametric type `'a` (Remark 3.11). However, we need to know how to decompose a global abstract state of type `'a` to access and update the local state of the local domain, of type `t`. This is the first usecase of the manager: `man.get` and `man.set` act as getters and setters of the local state, which is enclosed in the whole abstract state.

▷ **From a local domain to the global domain (lines 6-14).** During the execution of a local transfer function, handling a statement (or an expression), the local abstract domain may need to recursively call the whole analysis on a new statement or expression, starting from the root of DAG. This is done by calling the manager with the `man.exec` and `man.eval` functions. During their evaluation, these functions will traverse the DAG in search for a domain handling the provided statement or expression. For example, the C loop iterator Listing 3.6 calls the whole analysis over the rewritten loop to continue the analysis.
The manager can also be used to perform global lattice operations. This is used by the domain performing fixpoint computation for while loops (Listing 3.5, function `lfp`), to check if a fixpoint has been reached, or call the widening operator over the whole state otherwise.

3.3.2.3 Flow, wrapper of the global abstract state

The global abstract state of the analysis is an inhabitant of the type `'a flow`, rather than the expected `'a`. The `flow` type adds additional information. Its main usecase is to lift the abstract states to a mapping where keys are control-flow tokens. We explained in Section 2.4.3 how these control-flow tokens are used to handle non-local control-flow operators (such as `break`) when the analysis proceeds by induction on the syntax. The type of tokens can also be extended by any domain.

Example 3.12 **Use of `Flow` to handle non-local control-flow operators**

Listing 3.8 shows the transfer function of the `break` statement, as implemented by the universal iterator of loops. We store in `cur` the abstract state tagged by token `T_cur` (written *cur* in the concrete). Thus, `cur` has type `'a`. We return the updated global state, where `cur` is tagged with the `T_break` token, and where `T_cur` token has been removed.

Listing 3.8: Transfer function of `S_break` (cf. Figure 2.14 in the concrete)

```
type token += T_break

let rec exec stmt man flow =
  match stmt_kind stmt with
  | S_while(cond, body) ->
    (* Listing 3.5 *)

  | S_break ->
    let cur = Flow.get T_cur man.lattice flow in
    let flow' = Flow.add T_break cur man.lattice flow |>
                Flow.remove T_cur
    in
    Some (Post.return flow')

  | _ -> None
```

Example 3.13 **Utility functions to get and set a local state**

We show the source code of utility functions used to access and define a local state in Listing 3.9. These functions combine utilities defined by `Flow` to access the state corresponding to the provided token, and those provided by the manager to access the local state of domain (of type `t`), from the global state (of type `'a`).

Listing 3.9: Utility functions to get and set a local state

```
let get_env (tk:token) (man:('a,'t) man) (flow:'a flow) : 't =
  let tkstate : 'a = Flow.get tk man.lattice flow in
  man.get tkstate

let set_env (tk:token) (env:'t) (man:('a,'t) man) (flow:'a flow) : 'a flow =
  let tkstate : 'a = Flow.get tk man.lattice flow in
  let tkstate = man.set env tkstate in
  Flow.set tk tkstate man.lattice flow
```

The `'a flow` type also stores:

- a flow-insensitive context `'a ctx`,
- a report of the alarms already raised and the checks performed by the analysis,
- a report of the assumptions made by the analysis. For example, let us assume we analyze a C program where a function `f` is called. We assume also that this function is not defined in the source code, and only its prototype is available. To continue the analysis, Mopsa assumes that `f` has no side effects. We collect this assumption in the state to make it explicit and traceable (we provide an example later, in Listing 3.13).

These reports are then displayed to the user at the end of Mopsa's execution.

Remark 3.14 **Benefits of explicit checks and assumptions**

We believe that the reports provided are a step forward from the soundiness approach proposed by Livshits et al. [98]. We move from a declaration of the theoretical limits of

an approach, to an analyzer (here, Mopsa) explicitly collecting which properties have been verified, and which assumptions have been made.

3.3.2.4 Cases, postconditions and evaluations (lines 18-19)

Elements of `('a, 'b) cases` represent a disjunctive normal form of elements. These elements consist in a product of an element of type `'b case` in the context of a global abstract state `'a flow`. A `'b case` can be a result of type `'b` (along with additional optional information), or the special case empty denoting an error.

The interface of these cases is shown in Listing 3.10. We can create a singleton case using `Cases.return`. `Cases.empty` is the special case denoting an error. It is possible to join two or an arbitrary number of cases using `Cases.join` or `Cases.join_list`. The monadic operator `cases >>= f` executes the transfer function `f : 'b -> 'a flow -> ('a, 'c) cases` for each case in `cases : ('a, 'b) cases`. It returns a new case disjunction, of type `('a, 'c) cases`.

Listing 3.10: Cases interface

```
1 type ('a, 'b) cases
2
3 val return : 'b -> 'a flow -> ('a, 'b) cases
4 val empty : 'a flow -> ('a, 'b) cases
5
6 val join : ('a, 'b) cases -> ('a, 'b) cases -> ('a, 'b) cases
7 val join_list : ('a, 'b) cases list -> ('a, 'b) cases
8
9 val (>>=) : ('a, 'b) cases -> ('b -> 'a flow -> ('a, 'c) cases) -> ('a, 'c) cases
```

The type `'a post` is an alias for `('a, unit) cases`, representing a disjunction of abstract states. `'a eval` is an alias for `('a, expr) cases`. It represents a disjunction of expressions, each defined in a different abstract state. We show the benefits of these disjunctions in Example 3.17.

3.3.2.5 Transfer functions on expressions and statements (lines 36-38)

`init` is called to initialize the domain. It takes as argument a program, the analysis' manager, a global abstract state and returns a global abstract state, where the local domain state has been properly initialized.

Given a manager and a global abstract state, statements (resp. expressions) are executed (resp. evaluated) into postconditions (resp. evaluations).

Evaluations. Given an expression, a manager, and a global abstract state, `eval` returns a `'a eval option`. If the expression is handled by the domain, the returned value consists in a disjunction of expressions (each in a given state). Otherwise, the domain returns `None`. This means that abstract domains below the current one will be called until one supports the expression. An important feature of Mopsa is that expressions are evaluated into expressions themselves, in order to infer potentially relational invariants (and additionally, support domains performing rewriting in a unified framework). We illustrate the benefits of this approach in the examples below.

Postconditions. Given a statement, a manager and a global abstract state, `exec` returns a `'a post option`. If the statement is not supported by the given abstract domain, it will return `None`. For example, the domain `Python.Loops` does not support universal loop statements `S_while`, and will return `None`. However, the domain `Universal.Loops` will return a result. If the statement is supported by the abstract domain, a case disjunction of the final global abstract state is returned.

Example 3.15 **Assignment in a relational domain**

We consider the transfer function of an assignment in a relational domain. We show the implementation of this transfer function in Listing 3.11.

Assuming we have a string `s` in our toy language from the previous chapter, we consider the statement `x = s[i]`. Numerical domains are unable to handle that kind of assignments by themselves, since `s` is a string. However, we know that the numerical domain should handle the assignment, since `s[i]` has a numerical type (following the semantics chosen in Section 2.4.1). Thus, the numerical domain recursively calls the whole analysis to evaluate `s[i]`, using the manager (line 3). We will show later the transfer functions of this expression for both the string summarization and the string length domains, in Examples 3.17 and 3.19. For now, we just assume that `s[i]` is evaluated into an expression that the numerical domain can handle. Then, we can perform the assignment on our numerical domain. We start by retrieving the local state for the normal execution of the `flow` using `get_env` (whose definition was shown in Listing 3.9). In the case of a relational domain, the execution of the assignment is delegated to the Apron library [73]. We can then return the state, updated using `set_env`.

Listing 3.11: Transfer function of the assignment in a relational domain

```
1 let exec stmt man flow = match stmt_kind stmt with
2   | S_assign(E_var x, e) when is_numeric e ->
3     man.eval e flow >>= fun ee flow ->
4     let cur = get_env T_cur man flow in
5     let cur = Apron.assign (apron_var x) (apron_expr ee) cur in
6     Some (Cases.return () (set_env T_cur cur man flow))
7   | _ -> None
```

Remark 3.16 **Monadic operator >>=**

The cases monadic operator `cases >>= f` applies the function `f` on each case in `cases`. It is used to define the analysis on a single case at a time (for example, in Listing 3.11), and hides the case disjunctions from the implementation of the analysis' code.

Example 3.17 **Expression evaluation performing a simple rewriting**

The domain of smashed strings will evaluate `s[i]` into an auxiliary variable $\underline{\text{ord}}(s)$ (Listing 3.12). It creates the auxiliary variable by calling `mk_ord_string`.

The numerical domain will then be able to handle the assignment `x =` $\underline{\text{ord}}(s)$, since $\underline{\text{ord}}(s)$ is a purely numerical variable.

Listing 3.12: Transfer function of index access for the string summarization domain

```
1 let eval exp man flow = match ekind e with
2   | E_subscript(E_var s, i) ->
3     Some (Cases.return (mk_ord_string s) flow)
4
5   | _ -> None
```

Remark 3.18 **Modular definitions of abstract domains**

The definitions of abstract domains are modular: they do not make assumptions on the other domains. For example, the numerical domain is not specified in the transfer function of string summarization domain in Example 3.17. Changing it into another or a combination

of numerical domains does not require changes in Mopsa's code. Conversely, switching from the smashing abstraction to the string length domain does not affect the numerical domain, since it handles numerical expressions.

Example 3.19 **Expression evaluation with case disjunction**

Upon an index access of a string, the string length domain will check that the access is correct, and raises an alarm otherwise. We showed the concrete semantics in Figure 2.13, and the implementation is described in Listing 3.12.

`assume` is a global utility function that filters the global analysis' state by a condition (resp. its negation), and applies the function defined by the named argument `fthen` (resp. `felse`) on it. The filtering is done by delegation, calling `man.exec`.

In the case of the transfer function, the state is filtered according to the condition $0 \leq i < \underline{\text{len}}(s)$. The auxiliary variable $\underline{\text{len}}(s)$ is created by `mk_len_string s`. `assume` performs global evaluations and executions. In this case, the filtering will be performed by the numerical domain. If the condition is satisfied, we can keep in the state that this specific index access has been proved valid, and we return the interval $[0, 127]$ (following the semantics of Figure 2.13). If the condition is not respected, we add a new alarm in the state, stating the that index access is invalid, and do not return an expression, to interrupt the evaluation of this path.

Listing 3.13: Transfer function of index access for the string length domain

```
(* Global utility assume *)
let assume cond man flow ~fthen ~felse =
  man.eval cond flow >>$ fun econd flow ->
    man.exec (mk_assume econd) flow >>$ fun () then_flow ->
    man.exec (mk_assume (mk_not econd)) flow >>$ fun () else_flow ->
    Flow.join man.lattice (fthen then_flow) (felse else_flow)

(* Transfer function of the string length domain *)
let eval exp man flow = match ekind exp with
  | E_subscript(E_var s, i) ->
    Some (
      assume (mk_and (mk_le mk_zero i) (mk_lt i (mk_len_string s))) man flow
      ~fthen:(fun flow ->
        let flow = safe_subscript_access_check exp flow in
        Cases.return (mk_interval 0 127) flow)
      ~felse:(fun flow -> Cases.empty (invalid_subscript_access_alarm exp flow))
        )

  | _ -> None
```

Remark 3.20 **Avoiding combinatorial explosions**

Postconditions are joined into a single case after each global statement execution, in order to avoid a combinatorial explosion.

Remark 3.21 **Housekeeping auxiliary variables**

As we mentioned before, domains are responsible for their auxiliary variables. This means that when a variable is removed, auxiliary variables depending on this variable have to be removed too. We show how this is performed in the case of the string length domain in

Listing 3.14. If a string variable s is removed, this domain will rewrite the statement into a removal of the variable $\underline{\text{len}}(s)$.

Listing 3.14: String length transfer functions

```
let exec stmt man flow = match stmt_kind stmt with
  | S_remove {ekind = E_var v} when is_str v.vtyp ->
    Some (man.exec (mk_remove_var (mk_len_string e)) flow)
  (* similar cases for S_add, S_expand, S_fold *)
```

3.3.2.6 Utilities (lines 41-44)

The `merge` function is used to reconciliate diverging states in a reduced product (cf. Section 3.3.4). `ask` allows broadcasting queries to other domains (described in Section 3.3.5). `print_state` displays the abstract state of the domain. `print_expr` is used to display the part of the local abstract state representing the provided expression.[7] It can delegate calls to the manager using `man.print_expr`. For example, let us assume that `print_expr` is called to display a string variable s. As mentioned in Remark 3.6, each domain is responsible for the auxiliary variables it introduces. In this case, this means that the `print_expr` of the string length domain will delegate the printing to the numerical domain for the ghost variable $\underline{\text{len}}(s)$.

Remark 3.22 **Stateless abstract domains**

Stateless abstract domains such as iterators performing dynamic rewriting (Section 3.2.4) can be defined as modules with a simplified signature, where lattice operations do not have to be defined. They are then lifted automatically to the domain module type by Mopsa.

3.3.3 The simplified case of non-relational domains

Mopsa provides a simplified interface for non-relational domains, similar to the value abstract domains defined in Definition 2.51. They can then be lifted to the standard domain signature using a special combinator, similar to the semantics defined in Property 2.52.

Value abstract domains can be combined through a `union` combinator, representing a disjoint union between those abstract values. The most common usecase is the union of a value abstraction for integers and a value abstraction for floating-point numbers. These domains can then be lifted using a non-relational combinator having a definition similar to the one given in Property 2.52.

3.3.4 Reduced products and their pitfalls

Products are straightforwardly created in Mopsa using the corresponding combinator in the configuration. They can be extended to the case of reduced products through reduction rules over the states or the evaluations described below. We then highlight a pitfall of shared underlying domains in the case of a reduced product.

A classical example of reduced product is the reduction between the interval and the congruence abstract domains, defined in Section 2.3.9.2. We show the code used to define the reduction in Listing 3.15. Both the interval and the congruence abstract domains are defined through their abstract values, using the simplified case described in the previous section. `reduce` is defined only on the abstract values and lifted similarly (Definition 2.53). It is provided with a different manager that can access the internal states of the subdomains used in the product. `reduce` starts by fetching the congruence and the interval representing the value v through the manager. The reduction between those congruences and intervals is performed

[7] This is used in Mopsa's interactive mode, to show only the part of the abstact state related to a variable.

by `meet_cgr_itv` function, which implements the reduction operator described in Figure 2.11. The updated congruence and interval are put back in the abstract state.

Listing 3.15: Reduction between the interval and congruence domains

```
let reduce man v =
  let c = man.get Congruence.id v
  and i = man.get Interval.id v in

  let c', i' = meet_cgr_itv c i in

  man.set Congruence.id c' (man.set Interval.id i' v)
```

It is also possible to define reductions between evaluations of full domains. An example is the reduction performed after the evaluation of index accesses `s[i]` with both the string length and string summarization domains described in Section 2.4.6. The implementation of this reduction rule is shown in Listing 3.16. It traverses the results of the evaluation by the sub-domains, and keeps the auxiliary variable $\underline{\text{ord}}(s)$ created by the string summarization domain, and drops the evaluations of $[0, 127]$, created by the string length domain which is imprecise.

Listing 3.16: Reduction rule for index access between the string domains

```
let reduce exp _ _ _ results flow =
  let rec aux acc flow = function
  | [] -> Eval.singleton acc flow
  | hd::tl ->
    match ekind acc, ekind hd with
    | E_var _, E_constant (C_int_interval _) -> aux acc flow tl
    | E_constant (C_int_interval _), E_var _ -> aux hd flow tl
    | _ -> aux acc flow tl
  in
  match results with
  | [] -> Some (Eval.empty flow)
  | hd::tl -> aux hd flow tl
```

Effects on shared underlying domains. The reductions defined previously combine different expressions from domains of the product into a single one to improve the precision. A specific feature of Mopsa is that domains in a reduced product can share an underlying domain. During the execution of the analysis, domains in a reduced product may have different effects on this underlying domain, resulting in incoherent abstract states. In particular, if the resulting abstract states obtained by the product are directly intersected, the result will be unsound. It is however possible to recover back soundness by applying a generic `merge` function.

For example, we consider the following program: `s = "abc"; s = s + "def"`, written in the `Imp` language of the previous chapter. $+$ denotes the string concatenation operator. We first store a string in `s`, and update its value through a concatenation afterward. Let us assume that we have a product of the length and summarization domains, sharing an underlying interval domain (thus abiding by the configuration shown in Figure 3.2). We show the evaluation along the different domains and operators in Figure 3.3. The execution of `s = s + "def"` is transformed by the length domain into the execution of $\underline{\text{len}}(s) = \underline{\text{len}}(s) + 3$. This is handled by the numerical domain, which yields state $\sigma_1^\sharp$. Conversely, the execution of the initial statement is transformed by the string summarization domain into $\underline{\text{ord}}(s) \overset{\text{weak}}{=} \underline{\text{ord}}(s) + [100, 102]$. Once this is handled by the numerical domain, the resulting abstract state is $\sigma_2^\sharp$. We notice that the length variable is updated in $\sigma_1^\sharp$, but not in $\sigma_2^\sharp$, and conversely for the smashed variable. Additionally, none of these abstract states are sound after the assignment. $\sigma_1^\sharp$ and $\sigma_2^\sharp$ have to be merged into a sound post-condition. This is done by forgetting the *effects* of one domain over the other. In our case, $\sigma_1^\sharp$ modified $\underline{\text{len}}(s)$ and $\sigma_2^\sharp$ modified $\underline{\text{ord}}(s)$. Forgetting the effects

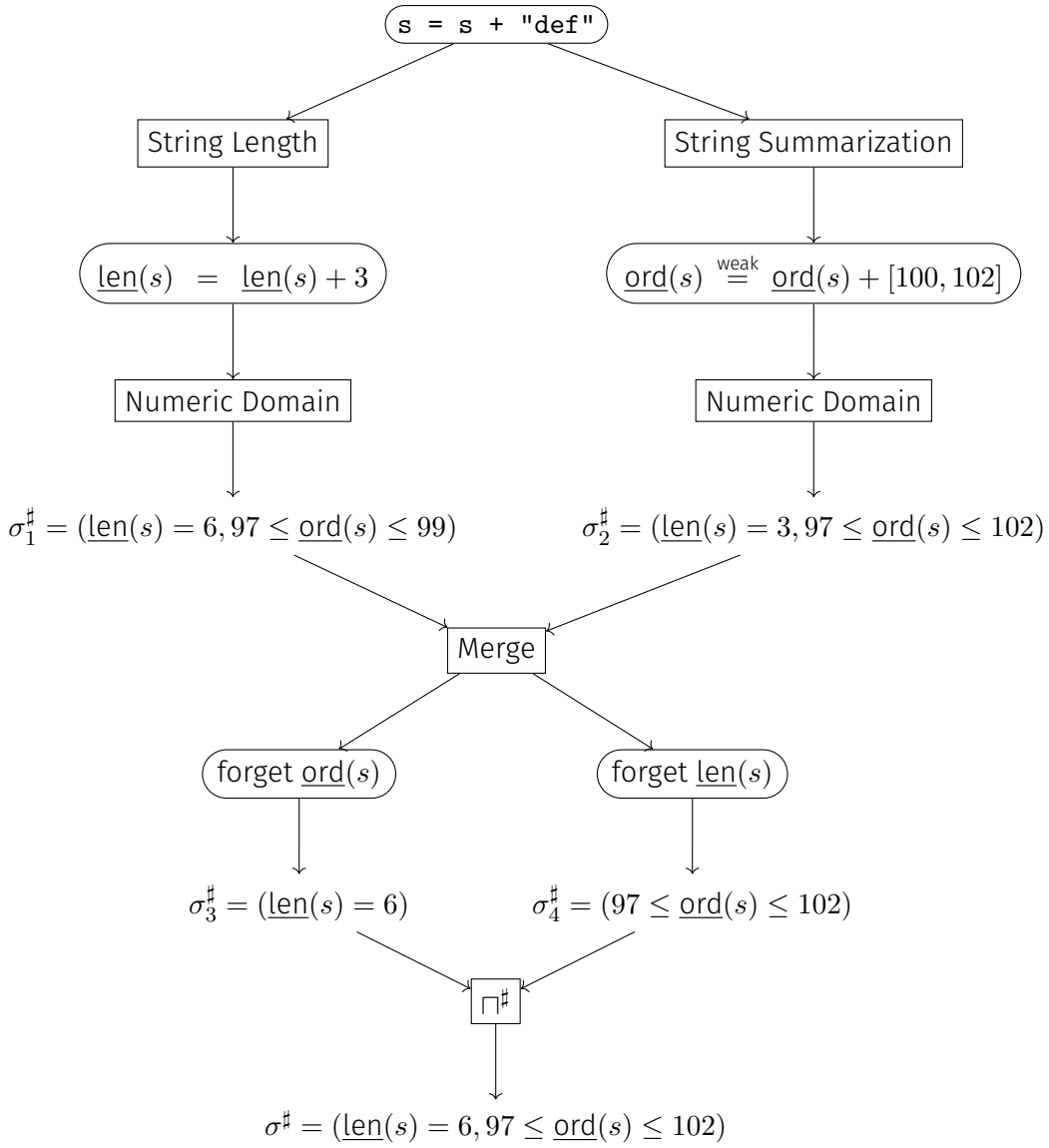

Figure 3.3: Example of shared evaluation in a reduced product

of the other domain yields $\sigma_3^\sharp$ and $\sigma_4^\sharp$ respectively. A sound postcondition $\sigma^\sharp$ is obtained by computing the meet of two states.

Remark 3.23 **Sequentialization of statements**

In this case, it would have been possible to sequentialize the operations to obtain a sound result without a call to `merge`. This is however not always possible, especially when relational domains are used.

Remark 3.24 **Effects**

Effects are collected automatically for each domain of the reduced product. For each domain, effects consist in a list of statements. A generic `merge` function is provided by the framework.

Remark 3.25 **Comparison with the approach of Chevalier [27] used in Astrée**

Chevalier mentions the same issue [27, page 330]:

> If modifying the abstract state of the left-hand side impacts the value of the right-hand side, strange interference may happen.

The solution he chooses is to add temporary variables to avoid that kind of situation. We believe that ensuring this condition when auxiliary variables are dynamically created (which is not the case in Chevalier's framework) is however difficult. In addition, introducing temporary variables increases the computational cost of relational numerical domains.

3.3.5 Communication between domains

Domains have a private type that is not available to other domains, and communicate in the analysis through statements or expressions. In some cases, inter-communication between domains through a public interface is necessary. We use a mechanism of queries similar to Astrée's input channels [35]. Queries are defined through an extensible GADT type, in order to make public the types of the queries arguments and their results. In essence, the interval query, asking for an interval representation of a provided expression, is defined as:

```
type _ query += Q_interval : expr -> (int option * int option) query
```

This query can be used to display some ranges of variables in alarm messages, or to compute the interval of a pointer offset in the analysis of C programs.

Queries are seldom used. It is usually better to communicate between abstractions through the evaluation of expressions or execution of statements. This avoids exposing abstract values, which improves precision when relational domains are used. It also avoids making assumptions about the domains used. We could replace the call to `assume` at line 12 of Listing 3.13 by queries asking for interval representations of the expression i and the variable $\underline{\text{len}}(s)$. This approach would have two downsides. First, relying on interval representations breaks relationality. If i is a variable, and the relational domain knows that $0 \leq i < \underline{\text{len}}(s) \leq 100$, the interval representations would yield $[0, 100]$ for both i and $\underline{\text{len}}(s)$. We would thus raise a false alarm. The second downside is that the string length domain would have to make explicit manipulations over intervals and handle the disjunctions, which creates lower-level code. Using our approach, these filterings were delegated to the numerical domain, whose purpose is to work on numerical abstractions.

A domain defines to which queries it answers through its `ask` transfer function. Lattice operators (join, meet) are also defined to handle the combination of replies from different domains.

Remark 3.26 **Comparison with Frama-C**

Frama-C domains cooperate through abstract values (currently intervals, to the best of our knowledge). Provided that Mopsa's query mechanism defines the same abstract values, Frama-C's domain cooperation could be simulated by these specific queries in Mopsa.

3.4 Hooks

Hooks are designed to observe the analysis. Their signature is provided in Listing 3.17. Hooks can be called at the beginning of the analysis, before and after any evaluation (resp. execution) of an expression (resp. statement), and at the end of the analysis. In particular, hooks see all subevaluations and subexecutions that may be created by domains calling the analysis recursively. Hooks can update the flow-insensitive context of the analysis to store information,

but they cannot modify the rest of the abstract state. They can thus interact with the analysis (such as the threshold hook presented below), but their execution cannot effect the analysis' soundness, contrary to abstract domains.

Listing 3.17: Hook signature

```
module type HOOK =
sig
  val name : string
  val init : 'a ctx -> 'a ctx
  val on_before_exec : stmt -> ('a,'a) man -> 'a flow -> 'a ctx option
  val on_after_exec  : stmt -> ('a,'a) man -> 'a flow -> 'a post -> 'a ctx option
  val on_before_eval : expr -> ('a,'a) man -> 'a flow -> 'a ctx option
  val on_after_eval  : expr -> ('a,'a) man -> 'a flow -> 'a eval -> 'a ctx option
  val on_finish : ('a,'a) man -> 'a flow -> unit
end
```

There are five use-cases for hooks in the current implementation of Mopsa.

▷ **Logs** display the execution and evaluation steps, and optionally the abstract state. This is useful for debugging purposes.

▷ **Coverage** hooks are provided for Python and C. They collect which statements have been analyzed and provide a global metric for the analysis' results. When a statement is analyzed, the hooks can also check if the input state is always bottom or not, to know if the statement is dead code or not. This family of hooks is quite handy to detect soundness issues (e.g., when a statement should have been analyzed but has not been).

▷ **Soundness sanity checking** hooks detect if the output state after an assignment is set to $\bot$ without alarms set, while the input state was not $\bot$. In that case, we have probably uncovered a soundness issue.

▷ **Profiling** hooks provide a high-level view of which part of the input program took time to analyze. Traditional profiling tools such as `perf` or OCaml's `memtrace` are too low-level and global to provide that kind of information. Due to function inlining and nested loops, the analysis time of programs is not proportional to their size. Our profiling hooks thus measure how much time is spent analyzing each function and each loop of a program.

▷ **Threshold** hook collects constants thresholds to be used by the widening with thresholds. The idea of widening with thresholds is to try increasing thresholds for unstable bounds rather than directly going to $\pm\infty$. This hook searches for comparisons between variables and constants happening within the analysis of loops. When a comparison is encountered, the constant is put as a threshold for the corresponding variable in the global context. This context is then read by the widening operators of the numerical domains, which can use the provided thresholds. Performing this search dynamically during the analysis, rather than a syntactical inspection of the input program has numerous benefits. The hook can find thresholds for ghost variables, and expressions may evaluate into constants during the analysis although they do not appear syntactically.

3.5 Formalization

In the previous chapter, the definition of domains explictly called other domains (e.g., the string length transfer function $\mathbb{E}^{\#}_{\text{len}}[\![\cdot]\!]$ explicitly calls the numerical domain $\mathbb{E}^{\#}_{\text{num}}[\![\cdot]\!]$ in Figure 2.13). In Mopsa, the `eval` and `exec` transfer functions of an abstract domain may call the global `man.eval` and `exec` transfer functions through the manager. To keep the analysis modular, a domain cannot explicitly call a specific domain, and we do not make hypotheses about which domains are present and which domain(s) can handle a (possibly transformed) expression or statement.

This section defines the notion of composable abstract domain, and how these domains can be composed or put in a reduced product. A limitation of our formalization is that soundness conditions are currently not defined locally, for each composable abstract domain. We write $\mathcal{E}$ the set of expressions and $\mathcal{S}$ the set of statements.

Definition 3.27 **Composable abstract domain**

A composable abstract domain consists in:

- An abstract poset $(\mathcal{D}^\sharp, \sqsubseteq)$,
- A smallest element $\bot$ and a largest element $\top$,
- Sound abstractions of set union and intersection $\sqcup^\sharp, \sqcap^\sharp$,
- A widening operator ∇,
- Partial expression and statement transfer functions, operating on the global abstraction state $\Sigma^\sharp$. The global abstraction state $\Sigma^\sharp$ corresponds to inhabitants of the type `'a` in the domain signature of Listing 3.7. The star is used to denote a list (i.e., a case disjunction), potentially empty (meaning that this domain does not handle this case).

$$\mathbb{E}^\#_{\mathcal{D}^\sharp}[\![\, expr \in \mathcal{E} \,]\!] : \Sigma^\sharp \to (\Sigma^\sharp \times \mathcal{E})^* \qquad \mathbb{S}^\#_{\mathcal{D}^\sharp}[\![\, stmt \in \mathcal{S} \,]\!] : \Sigma^\sharp \to \Sigma^{\sharp *}$$

- Concrete input and output states of the abstract domain, written $\mathcal{D}^{in}$ and $\mathcal{D}^{out}$
- A concretization operator $\gamma \in \mathcal{D}^\sharp \to \mathcal{P}(\mathcal{P}(\mathcal{D}^{in}) \times \mathcal{P}(\mathcal{D}^{out}))$. Concretizations are expressed modularly, as relations between concrete states, in line with what has been presented in Section 2.4.

Example 3.28 **String length as a composable abstract domain**

Let us consider the string summarization domain of Section 2.4.5. The poset and lattice operators are trivial since the domain is stateless. The transfer functions have been described in Figure 2.17. The concrete input state is a numerical state $\mathcal{V} \to \mathbb{Z}$, and the output state can also define string values $\mathcal{V} \to (\mathbb{Z} \cup \mathcal{A}^*)$. The concretization was defined in Definition 2.84.

Definition 3.29 **Sequence of abstract domains**

Let $(\mathcal{D}^\sharp_i, \sqsubseteq_i, \bot_i, \top_i, \sqcup^\sharp_i, \sqcap^\sharp_i, \nabla_i, \mathbb{E}^\#_i[\![\,\cdot\,]\!], \mathbb{S}^\#_i[\![\,\cdot\,]\!], \gamma_i), i \in \{1, 2\}$ be two composable abstract domains.

We define the sequence of these two domains, written $\begin{smallmatrix}\mathcal{D}^\sharp_2\\ |\\ \mathcal{D}^\sharp_1\end{smallmatrix}$ as:

- A poset $(\mathcal{D}^\sharp_1 \times \mathcal{D}^\sharp_2, \sqsubseteq_{1\times 2})$, with

$$(v_1, v_2) \sqsubseteq^\sharp_{1\times 2} (v'_1, v'_2) \stackrel{\text{def}}{=} v_1 \sqsubseteq^\sharp_1 v'_1 \wedge v_2 \sqsubseteq^\sharp_2 v'_2$$

- Similar lattice and widening operators lifted to the product.
- The transfer functions first try using domain 2. If domain 2 is unable to handle the expression or statement, domain 1 is used.

$$\mathbb{E}^\#_{1\times 2}[\![\, e \,]\!] = \text{let } r = \mathbb{E}^\#_2[\![\, e \,]\!] \text{ in if } r \neq \emptyset \text{ then } r \text{ else } \mathbb{E}^\#_1[\![\, e \,]\!]$$
$$\mathbb{S}^\#_{1\times 2}[\![\, s \,]\!] = \text{let } r = \mathbb{S}^\#_2[\![\, e \,]\!] \text{ in if } r \neq \emptyset \text{ then } r \text{ else } \mathbb{S}^\#_1[\![\, e \,]\!]$$

- A concretization operator (we assume that $\mathcal{D}^{out}_1 = \mathcal{D}^{in}_2$), corresponding to Defini-

tion 2.82.

$$\gamma_{1\times 2} : \left\{ \begin{array}{rcl} \mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp & \to & \mathcal{P}(\mathcal{P}(\mathcal{D}_1^{in}) \times \mathcal{P}(\mathcal{D}_2^{out})) \\ (\sigma_1, \sigma_2) & \mapsto & \{\, (R_1^i, R_2^o) \mid (R_1^i, R_1^o) \in \gamma_1(\sigma_1) \wedge (R_2^i, R_2^o) \in \gamma_2(\sigma_2) \wedge R_2^i \subseteq R_1^o \,\} \end{array} \right.$$

Property 3.30 **A sequence of abstract domains is composable**

$\begin{matrix} \mathcal{D}_2^\sharp \\ | \\ \mathcal{D}_1^\sharp \end{matrix}$ is also a composable abstract domain.

Definition 3.31 **Product of abstract domains**

Given two composable abstract domains, $(\mathcal{D}_i^\sharp, \sqsubseteq_i, \bot_i, \top_i, \sqcup_i^\sharp, \sqcap_i^\sharp, \nabla_i, \mathbb{E}_i^\#[\![\cdot]\!], \mathbb{S}_i^\#[\![\cdot]\!], \gamma_i)$, with $i \in \{1, 2\}$, and two functions `reduce_eval`, `reduce_exec`, defining reductions between evaluations and executions, we define the product $\mathcal{D}_{1\wedge 2}^\sharp$ as:

- A poset $(\mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp, \sqsubseteq_{1\times 2})$,
- Similar lattice operators lifted to the product,
- The transfer functions run the transfer functions of domain 1 and 2 in parallel. They are then unified using the `merge` function applying effects, and reduction rules can then be applied. We left implicit the process of gathering effects from domains and passing it to the merge function (in Mopsa, this is done automatically by logging assignments in expressions passed as argument).

$$\begin{array}{l} \mathbb{E}_{1\wedge 2}^\#[\![\, e \,]\!]\sigma = \\ \quad \text{let } v_1, \sigma_1 = \mathbb{E}_1^\#[\![\, e \,]\!] \text{ and } v_2, \sigma_2 = \mathbb{E}_2^\#[\![\, e \,]\!] \text{ in} \\ \quad \text{let } \sigma_1 = \text{merge}(\sigma_1) \text{ in} \\ \quad \text{let } \sigma_2 = \text{merge}(\sigma_2) \text{ in} \\ \quad \text{reduce\_eval}((v_1, \sigma_1), (v_2, \sigma_2)) \\ \mathbb{S}_{1\wedge 2}^\#[\![\, s \,]\!] = \\ \quad \text{let } \sigma_1 = \mathbb{E}_1^\#[\![\, e \,]\!] \text{ and } \sigma_2 = \mathbb{E}_2^\#[\![\, e \,]\!] \text{ in} \\ \quad \text{let } \sigma_1 = \text{merge}(\sigma_1) \text{ in} \\ \quad \text{let } \sigma_2 = \text{merge}(\sigma_2) \text{ in} \\ \quad \text{reduce\_exec}(\sigma_1, \sigma_2) \end{array}$$

- A concretization operator (we assume that γ_1 and γ_2 have the same codomain).

$$\gamma_{1\wedge 2} : \left\{ \begin{array}{rcl} \mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp & \to & \mathcal{P}(\mathcal{P}(\mathcal{D}^{in}) \times \mathcal{P}(\mathcal{D}^{out})) \\ (\sigma_1, \sigma_2) & \mapsto & \{\, (R, R_1 \cap R_2) \mid (R, R_1) \in \gamma_1(\sigma_1) \wedge (R, R_2) \in \gamma_2(\sigma_2) \,\} \end{array} \right.$$

Property 3.32 **A product of abstract domains is composable**

$\mathcal{D}_{1\wedge 2}^\sharp$ is also a composable abstract domain.

Remark 3.33 **Toplevel abstract domain**

The toplevel abstract domain consists of a whole combination of domains. It is also a composable abstract domain, thanks to Properties 3.30 and 3.32. From the toplevel abstract

domain, we derive global evaluation and execution operators (corresponding to `man.eval` and `man.exec`). Contrary to the partial functions of the abstract domain, these operators are total: if some expression or statement is not supported, the operators fail and raise an exception.

$$\mathbb{E}^\sharp[\![\cdot]\!] : \Sigma^\sharp \to (\Sigma^\sharp \times \mathcal{E})^+ \qquad \mathbb{S}^\sharp[\![\cdot]\!] : \Sigma^\sharp \to \Sigma^{\sharp+}$$

Given the concretization $\gamma : \Sigma^\sharp \to \mathcal{P}(\mathcal{P}(\emptyset) \times \mathcal{P}(\Sigma))$ (leaf domains take the empty set as input, cf. Remark 2.83), the toplevel concretization consists in the image of the whole concretization: $\Gamma(\sigma^\sharp) = \downarrow \circ \gamma(\sigma^\sharp)$ (using the operator defined in Definition 2.81).

Remark 3.34 **Stack variation**

In Mopsa, the notion of abstract domain can be further generalized to allow lattice operators to perform unification on domains below them.

Some domains using auxiliary variables in underlying domains may need to perform unifications on these domains when performing lattice operations. This kind of domain is called a stack domain. In that case, these operators have a signature similar to the one below

```
val join  : ('a,t) man -> ('a,'s) stack_man -> 'a ctx -> t * 's -> t * 's -> t * 's * 's
```

The `join` operator takes:

- a global manager as argument,
- a simplified manager working on the underlying abstraction, of type `'s`,
- the global context of the abstraction
- the two abstract states on which the join is performed, including the underlying abstractions.

It returns the joined abstract state as well as the unified underlying states.

In order to ease definitions, we do not consider this case in the formalization. These domains are currently not used in the analysis of Python programs.

3.6 Conclusion

We have shown that Mopsa aims at performing precise relational analyses through loose combinations of abstract domains that can be easily changed by the user. Moreover, Mopsa supports multiple languages, and some abstract domains are shared among the languages in order to reduce implementation costs. We have formalized the notion of composable abstract domain. In addition to the works related to Python presented in this thesis, Mopsa has been used by:

- Journault et al. [81] to implement a domain for C strings,
- Ouadjaout and Miné [121] to define a stub language modeling libraries such as the C standard library.
- Delmas and Miné [42], Delmas et al. [44] to infer other program properties such as semantic equivalence of patched programs or endian portability of programs.

There are still open questions related to Mopsa's design:

- The implementation of backwards analyses, where abstract domains are defined modularly and the analysis works by induction on the syntax. Backwards analyses are supported by TAJS and some versions of Astrée.

- The design of efficient, yet precise, modular analyses of functions. Mopsa currently implements a context-sensitive, top-down interprocedural analysis. In its non-relational version, Frama-C relies on a mechanism performing function summarization on-the-fly, to improve performances [158]. Infer performs efficient interprocedural analyses in a bottom-up fashion, but without soundness guarantees.
- We do not know yet how to state and prove the soundness properties locally, one composable abstract domain at a time.

4

Abstracting Dynamic Memory Allocation

This chapter presents the recency abstraction, which we use to handle dynamic memory allocation in our analyses. We present the recency abstraction on an extended version of `Imp` in Section 4.1. Following what we have established in the previous chapters, the abstract semantics and the concretization of the recency abstraction are defined modularly, using formulations compatible with relational numerical abstractions. Section 4.2 defines a variation of the recency abstraction changing the allocation-site sensitivity, where abstract addresses are more suitable to analyze Python.We briefly talk about abstract garbage collection in Section 4.3. The last two sections are an extended version of a previous work we published at SOAP [112].

4.1 The recency abstraction

The recency abstraction has originally been developed by Balakrishnan and Reps [4] to handle dynamic memory allocation in the analysis of binaries or low-level C code. We present the recency abstraction in a similar setting, where our toy imperative language is extended with simple records.

4.1.1 Motivation

`Imp` extension. We consider an extension of `Imp` with OCaml-like records. The type declaration of records is done using the `struct` keyword, followed by the name of the type and the declaration of the record structure. The expression `new <struct_type>` allocates a new record, where all fields are initialized with a random value. Fields of a record are accessed using the `.` operator; they can be read from and written to. Records are passed by reference, so no copy is performed by assignments. An example is shown in Listing 4.1. A `Box` type is created: it contains an integer named `v`. Two variables of type `Box` are declared `b` and `old`. A `Box` is allocated, and a reference to it stored in `b`. Its field `v` is set to -1. Each time the loop is executed, it copies the reference to the previous `Box`, stored in `b`, into `old`. Then, it allocates a new `Box`, and puts the value of the counter `i` in its field `v`. In the end, we know that `old.v` is 99, and `b.v` is 100.

The need for abstraction. If we do not abstract dynamic memory allocation and create one address for each allocation the analysis will not be guaranteed to terminate when allocations

are performed in unbounded loops, such as in our example. We thus need to abstract away from this behavior.

Listing 4.1: `Imp` program with dynamic memory allocation

```
1  struct Box { int v;};
2
3  Box b;
4  Box old;
5
6  int n = [1, +∞];
7  int i = 0;
8
9  b = new Box;
10 b.v = -1;
11 while(i < n) {
12   i = i + 1;
13   old = b;
14   b = new Box;
15   b.v = i;
16 }
17 // old.v = n-1, b.v = n
```

Allocation-site abstraction. A first approach is to summarize all concrete allocated addresses. These addresses are still partitioned by allocation sites (i.e., the location, defined by the line and column numbers, in the source code). A single abstract address thus represents all concrete addresses allocated at a given site. For example, all addresses allocated by `new Box` in the loop at line 14 would use the same abstract address written $@^\sharp_{14}$. Since this address summarizes multiple concrete elements, only weak updates will be performed. This approach guarantees the termination of the analysis but is not very precise due to weak updates. The non-deterministic initialization of records means that the value of the field v of $@^\sharp_{14}$ at line 14 can be any integer. Since $@^\sharp_{14}$ summarizes multiple concrete addresses, it is not possible to perform a strong update at line 15. Even if the initialization was deterministic, the weak update of the assignement `b.v = i` at line 15 would mean the analysis is only able to infer that `-1 <= b.v <= n`.

Recency abstraction. To circumvent the limitations of the allocation-site abstraction, the recency abstraction extends the allocation-site abstraction with another partitioning criterion. At a given allocation site l, the recency discriminates the most recent allocation, $@^\sharp_{l,\mathtt{r}}$ where strong updates are possible, from the older addresses $@^\sharp_{l,\mathtt{o}}$, which are summarized into a weak address. The recency criterion was introduced to keep precision during the initialization of a structure, which usually happens just after its allocation (or at least before the next allocation at the same site). If we go back to the example of Listing 4.1, during the allocation at line 14, the recency first renames the recent address into the old one. The field `v` of the recent address initially has for value any integer, but we can then represent the constraint from `b.v = i` in the abstract since the address is now strong. With a relational domain, we can infer that after the loop exits, `-1 <= old.v <= n-1` and `b.v = n`.

4.1.2 Concrete semantics

This section builds upon the semantics of `Imp` presented in Chapter 2 to define the semantics of record allocation, reading from a field, and writing to a field.

Definition 4.1 **Addresses**

In order to simplify the correspondence with the abstract state, we assume that addresses have the form $\textsf{Addr} = \textsf{Loc} \times \mathbb{N}$. We write $@_{l,n}$ to represent the n-th address allocated at program location l.

Definition 4.2 **Program State**

A program state consists of an environment and a heap. The environment $\mathcal{E}$ maps variables to their values (for scalar types) and addresses (for record types). Values can be strings or integers. The heap maps tuples of addresses and records names (strings) to values or addresses.

$$\mathcal{E} = \mathcal{V} \rightharpoonup \textsf{Value} \cup \textsf{Addr} \qquad \mathcal{H} = (\textsf{Addr} \times \mathit{string}) \rightharpoonup \textsf{Value} \cup \textsf{Addr}$$

Example 4.3 **Program State**

After line 9 of Listing 4.1, the state is one of

$$\{\, (e,h) \mid e(n) \geq 1, e(i) = 0, e(b) = @_{9,0}, h(@_{9,0}, \text{"}v\text{"}) \in \mathbb{Z} \,\}$$

We define the semantics of record allocations, reads, and writes to fields. Since record allocation changes the heap, $\mathbb{E}[\![\ \cdot\ ]\!]$ evaluates a set of states into value or addresses in an updated state, i.e. $\mathbb{E}[\![\ \cdot\ ]\!] \in \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S} \times (\textsf{Value} \cup \textsf{Addr}))$, with $\mathcal{S} = \mathcal{E} \times \mathcal{H}$. We introduce an auxiliary expression $\mathit{alloc\_addr}(l \in \textsf{Loc})$, asking for the allocation of a new address at a given program point. To simplify notations, we assume the program has been type-checked before, and all fields accesses are valid with respect to the records declared. We assume we have two auxiliary functions: $\texttt{fields}(RType)$ returns the field names of record type `RType`, and $\texttt{typ}(RType, f)$ returns the type of field f in record $RType$.

The semantics are provided in Figure 4.1. $\mathit{alloc\_addr}(l)$ returns the next address at program location l that is not already used in the heap abstraction. The allocation of a new record $RType$ at program location l first allocates a new address for the record. The heap is extended so that each field of the newly allocated record has a value of the corresponding type. A field access $x.a$ searches for the address to which x points by querying the environment, and returns the value of the heap at this given address and with field a. We can assume $e(x)$ is an address since the program is well-typed. We have defined the semantics of expressions on a single state $(e,h) \in \mathcal{S}$, but the semantics is join-morphism, so we can easily lift it to sets of states (the lift is defined in Figure 4.1 too). The semantics of field writes consists in evaluating the new value v, and updating the heap with the new value.

Example 4.4 **Field assignment**

From the program states of Example 4.3, the assignment `b.v = -1` yields:

$$\{\, (e,h) \mid e(n) \geq 1, e(i) = 0, e(b) = @_{9,0}, h(@_{9,0}, \text{"}v\text{"}) = -1 \,\}$$

4.1.3 The recency abstraction

In the recency abstraction, abstract addresses do not keep track of the number of addresses allocated at a given point, but of a flag describing whether the address is the most recent allocated or not.

Definition 4.5 **Abstract address**

An abstract address consists of an allocation site, and a flag: r describes the most recent

$$
\begin{aligned}
&\mathbb{E}[\![\, \mathit{alloc\_addr}(l) \,]\!](e,h) = \text{let } n = \ \max\{\, @_{l,i} \in \mathrm{dom} h \,\} \text{ in } @_{l,n+1} \\
&\mathbb{E}[\![\, \texttt{new}\ \ RType^{l} \,]\!](e,h) = \\
&\quad \text{let } @_{l,n} = \mathbb{E}[\![\, \mathit{alloc\_addr}(l) \,]\!](e,h) \text{ in} \\
&\quad \{\, (e,h'), @_{l,n} \mid (@,a) \in \mathrm{dom} h \Leftrightarrow (h'(@,a) = h(@,a) \wedge \\
&\qquad\qquad \forall f \in \texttt{fields}(RType), h'(@_{l,n}, f) \in \mathsf{Value}_{\texttt{typ}(RType,f)}) \,\} \\
&\mathbb{E}[\![\, x.a \,]\!](e,h) = h(e(x),a) \\
&\mathbb{E}[\![\, expr \,]\!]\Sigma = \cup_{(e,h)\in\Sigma} \mathbb{E}[\![\, expr \,]\!](e,h) \\
&\mathbb{S}[\![\, x.a = v \,]\!](e,h) = \\
&\quad \text{let } val = \mathbb{E}[\![\, v \,]\!](e,h) \text{ in } (e, h[(e(x),a) \mapsto val])
\end{aligned}
$$

Figure 4.1: Concrete semantics of record allocation, read and write

allocated address, while **o** describes older allocated addresses. For a given program, the set of abstract address is thus finite.

$$\mathsf{Addr}^\sharp = \mathsf{Loc} \times \{\, \mathsf{r}, \mathsf{o} \,\}$$

Abstract addresses are written $@^\sharp_{l,\mathsf{m}}$, with $l \in \mathsf{Loc}, m \in \{\, \mathsf{r}, \mathsf{o} \,\}$.

$$
\begin{aligned}
&\mathbb{E}^{\#}_{\mathsf{mem}}[\![\, \mathit{alloc\_addr}(l \in \mathsf{Loc}) \,]\!]\sigma^\sharp = \\
&\quad \text{let } \sigma^\sharp_{mem} = \texttt{man.get}\ \ \sigma^\sharp \text{ in} \\
&\quad \text{if } @^\sharp_{l,\mathsf{r}} \in \sigma^\sharp_{mem} \text{ then} \\
&\qquad \text{if } @^\sharp_{l,\mathsf{o}} \in \sigma^\sharp_{mem} \text{ then } @^\sharp_{l,\mathsf{r}}, \mathbb{S}^{\#}[\![\, \texttt{fold}(@^\sharp_{l,\mathsf{o}}, @^\sharp_{l,\mathsf{r}}) \,]\!]\sigma^\sharp \\
&\qquad \text{else} \\
&\qquad\quad \text{let } \sigma^\sharp_{mem} = \sigma^\sharp_{mem} \cup \{\, @^\sharp_{l,\mathsf{o}} \,\} \text{ in} \\
&\qquad\quad @^\sharp_{l,\mathsf{r}}, \mathbb{S}^{\#}[\![\, \texttt{rename}(@^\sharp_{l,\mathsf{r}}, @^\sharp_{l,\mathsf{o}}) \,]\!](\texttt{man.set}\ \ \sigma^\sharp_{mem}\ \sigma^\sharp) \\
&\quad \text{else} \\
&\qquad \text{let } \sigma^\sharp_{mem} = \sigma^\sharp_{mem} \cup \{\, @^\sharp_{l,\mathsf{r}} \,\} \text{ in} \\
&\qquad @^\sharp_{l,\mathsf{r}}, \texttt{man.set}\ \ \sigma^\sharp_{mem}\ \sigma^\sharp
\end{aligned}
$$

Figure 4.2: Semantics of address allocation for the recency abstraction

The domain of the recency abstraction is the set of allocated abstract addresses, $\mathcal{P}(\mathsf{Addr}^\sharp)$; its lattice operations are straightforward. The semantics of address allocation at a given program location l is shown in Figure 4.2. We use the "mem" subscript to denote the abstract domain of the recency abstraction. Given the whole abstraction $\sigma^\sharp$, we extract the state of the recency abstraction $\sigma^\sharp_{mem} \in \mathcal{P}(\mathsf{Addr}^\sharp)$. If both the recent address and the old address are already allocated, we return the recent address in a state where it has been folded into the old address (according to the `fold` definition of Gopan et al. [63], Siegel and Simon [135], already encountered in Section 2.4.5). If the old address has not been allocated yet, we add it to the recency's state, and we `rename` the recent address into the old one. If the recent address has not been allocated yet, we update the local state and return the recent address. The `fold` and

`rename` operations are executed as statements, so all domains using abstract addresses can update themselves. The global semantics operators $\mathbb{E}[\![\cdot]\!]$ (resp. $\mathbb{S}[\![\cdot]\!]$) correspond to `man.exec` (resp. `man.eval`) in Mopsa.

4.1.4 Abstract semantics

We introduce auxiliary address variables representing fields at abstract addresses. These auxiliary variables simplify our implementation, as the environment will also store information on records' fields. The environment abstraction is straightforward and close to the ones shown in Chapter 2. The heap abstraction rewrites fields accesses into auxiliary variables for the environment domain. It also handles record allocation by delegating the address allocation and initializing all fields of the record to any value of the corresponding type.

Definition 4.6 **Auxiliary address variables**

$\underline{@^\sharp_{l,\mathsf{m}} \cdot v}$ is an auxiliary address variable, representing the field v of the record stored at address $@^\sharp_{l,\mathsf{m}}$. If the abstract address is old, it represents multiple concrete elements. In that case, auxiliary address variables have to be tagged as weak (as defined in Remark 2.75): $\underline{@^\sharp_{l,\mathsf{o}} \cdot v}_{\texttt{weak}}$. In the following, this tag is implicit, depending on the mode of the address.

Remark 4.7 **Auxiliary address variables in Mopsa**

In the implementation of Mopsa, these auxiliary variables are created by extending the variant type of variables.

```
type var_kind +=
  | V_addr_attr of addr * string
```

Continuing our modular definitions of abstract domains, the heap abstraction presented in Figure 4.3 is independent of the recency abstraction: it only asks another domain to perform address allocations. In particular, abstract addresses are only written $@^\sharp$ to emphasize that the heap abstraction does not need to know the structure of abstract addresses.[1] However, the heap abstraction needs to support the `rename` and `fold` operations the recency can trigger (Figure 4.2), and requires to know if an address is weak or strong (denoted old or recent in the case of the recency abstraction). The evaluation of field access $x.a$ searches for the abstract address $@^\sharp$ to which x maps, and then delegates the evaluation by rewriting it in the auxiliary address variable $\underline{@^\sharp \cdot a}$. This evaluation will then be handled by the environment abstraction. Writing to a given field performs a similar delegation. The allocation of a new record is similar to the concrete: we evaluate a new address allocation (handled by the recency abstraction in our case), and put all the fields $a_1, \cdots, a_n \in \texttt{fields}(RType)$ of the record to any value of their corresponding type. If an abstract address has to be renamed into another, the heap abstraction delegates this operation by renaming all corresponding auxiliary address variables (as we mentioned in Remark 3.6, each domain is responsible for its ghost variables). The heap abstraction performs a similar delegation for `fold` operations.

Example 4.8 **New allocation in Listing 4.1**

We show in Figure 4.4 the statements and expressions analyzed, starting from the statement at line 14 of Listing 4.1 and assuming the loop has been unrolled once before. For the sake of concision, we assume the environment is based on intervals; in the case of constant

[1] Auxiliary variables building upon summarized address should only perform weak updates, so the fact that an abstract address is summarized or not should be available to the heap domain.

$$
\begin{aligned}
&\mathbb{E}^\sharp_{heap}[\![\, x.a \,]\!]\sigma^\sharp = \\
&\quad \text{let } @^\sharp = \mathbb{E}^\sharp[\![\, x \,]\!]\sigma^\sharp \text{ in } \mathbb{E}^\sharp[\![\, \underline{@^\sharp \cdot a} \,]\!]\sigma^\sharp \\
&\mathbb{S}^\sharp_{heap}[\![\, x.a = v \,]\!]\sigma^\sharp = \\
&\quad \text{let } @^\sharp = \mathbb{E}^\sharp[\![\, x \,]\!]\sigma^\sharp \text{ in } \mathbb{S}^\sharp[\![\, \underline{@^\sharp \cdot a} = v \,]\!]\sigma^\sharp \\
&\mathbb{E}^\sharp_{heap}[\![\, \texttt{new } RType^l \,]\!]\sigma^\sharp = \\
&\quad \text{let } @^\sharp, \sigma^\sharp = \mathbb{E}^\sharp[\![\, \mathit{alloc\_addr}(l \in \mathsf{Loc}) \,]\!]\sigma^\sharp \text{ in} \\
&\quad \mathbb{S}^\sharp[\![\, \underline{@^\sharp \cdot a_n} = \top_{\mathtt{typ}(RType,a_n)} \,]\!] \circ \cdots \circ \mathbb{S}^\sharp[\![\, \underline{@^\sharp \cdot a_1} = \top_{\mathtt{typ}(RType,a_1)} \,]\!]\sigma^\sharp \\
&\mathbb{S}^\sharp_{heap}[\![\, \texttt{rename}(@^\sharp_1, @^\sharp_2) \,]\!] = \\
&\quad \mathbb{S}^\sharp[\![\, \texttt{rename}(\underline{@^\sharp_1 \cdot a_n}, \underline{@^\sharp_2 \cdot a_n}) \,]\!] \circ \cdots \circ \mathbb{S}^\sharp[\![\, \texttt{rename}(\underline{@^\sharp_1 \cdot a_1}, \underline{@^\sharp_2 \cdot a_1}) \,]\!] \\
&\mathbb{S}^\sharp_{heap}[\![\, \texttt{fold}(@^\sharp_1, @^\sharp_2) \,]\!] = \\
&\quad \mathbb{S}^\sharp[\![\, \texttt{fold}(\underline{@^\sharp_1 \cdot a_n}, \underline{@^\sharp_2 \cdot a_n}) \,]\!] \circ \cdots \circ \mathbb{S}^\sharp[\![\, \texttt{fold}(\underline{@^\sharp_1 \cdot a_1}, \underline{@^\sharp_2 \cdot a_1}) \,]\!]
\end{aligned}
$$

Figure 4.3: Rewriting semantics of the heap abstraction

$$
\begin{aligned}
&\mathbb{S}^\sharp_{env}[\![\, b = \texttt{new } Box^{14} \,]\!]\sigma^\sharp \\
&\quad \hookrightarrow \mathbb{E}^\sharp_{heap}[\![\, \texttt{new } Box^{14} \,]\!]\sigma^\sharp \qquad \sigma^\sharp = \begin{cases} \mathsf{mem} & \{\, @^\sharp_{9,\mathsf{r}}; @^\sharp_{14,\mathsf{r}} \,\} \\ \mathsf{heap} & () \\ \mathsf{env} & i \mapsto 2, old \mapsto @^\sharp_{14,\mathsf{r}}, b \mapsto @^\sharp_{14,\mathsf{r}} \\ & \underline{@^\sharp_{9,\mathsf{r}} \cdot v} \mapsto -1, \underline{@^\sharp_{14,\mathsf{r}} \cdot v} \mapsto 1 \end{cases} \\
&\quad\quad \hookrightarrow \mathbb{E}^\sharp_{mem}[\![\, \mathit{alloc\_addr}(14) \,]\!]\sigma^\sharp \\
&\quad\quad\quad \hookrightarrow \sigma^\sharp_{1,\mathsf{mem}} = \sigma^\sharp_{\mathsf{mem}} \cup \{\, @^\sharp_{14,\mathsf{o}} \,\} \\
&\quad\quad\quad \hookrightarrow \mathbb{S}^\sharp_{heap}[\![\, \texttt{rename}(@^\sharp_{14,\mathsf{r}}, @^\sharp_{14,\mathsf{o}}) \,]\!]\sigma^\sharp_1 \\
&\quad\quad\quad\quad \hookrightarrow \mathbb{S}^\sharp_{mem}[\![\, \texttt{rename}(\underline{@^\sharp_{14,\mathsf{r}} \cdot v}, \underline{@^\sharp_{14,\mathsf{o}} \cdot v}) \,]\!]\sigma^\sharp_1 \\
&\quad\quad \leftarrow @^\sharp_{14,\mathsf{r}}, \sigma^\sharp_2 \qquad \sigma^\sharp_2 = \sigma^\sharp_1[\mathsf{env} \mapsto (i \mapsto 2, old \mapsto @^\sharp_{14,\mathsf{o}}, b \mapsto @^\sharp_{14,\mathsf{o}}, \\
&\quad\quad\quad\quad\quad\quad\quad\quad \underline{@^\sharp_{14,\mathsf{o}} \cdot v} \mapsto 1, \underline{@^\sharp_{9,\mathsf{r}} \cdot v} \mapsto -1)] \\
&\quad\quad \hookrightarrow \mathbb{S}^\sharp_{env}[\![\, \underline{@^\sharp_{14,\mathsf{r}} \cdot v} = \top_{int} \,]\!]\sigma^\sharp_2 \\
&\quad \leftarrow @^\sharp_{14,\mathsf{r}}, \sigma^\sharp_3 \qquad \sigma^\sharp_3 = \sigma^\sharp_2[\mathsf{env} \mapsto \sigma^\sharp_{2,\mathsf{env}}[\underline{@^\sharp_{14,\mathsf{r}} \cdot v} \mapsto [-\infty, +\infty]]] \\
&\quad \hookrightarrow \sigma^\sharp_{4,\mathsf{env}} = \sigma^\sharp_{3,\mathsf{env}}[b \mapsto @^\sharp_{14,\mathsf{r}}]
\end{aligned}
$$

$$
\sigma^\sharp_4 = \begin{cases} \mathsf{mem} & \{\, @^\sharp_{9,\mathsf{r}}; @^\sharp_{14,\mathsf{r}}; @^\sharp_{14,\mathsf{o}} \,\} \\ \mathsf{heap} & () \\ \mathsf{env} & i \mapsto 2, old \mapsto @^\sharp_{14,\mathsf{o}}, b \mapsto @^\sharp_{14,\mathsf{r}}, \underline{@^\sharp_{9,\mathsf{r}} \cdot v} \mapsto -1, \\ & \underline{@^\sharp_{14,\mathsf{r}} \cdot v} \mapsto [-\infty, +\infty], \underline{@^\sharp_{14,\mathsf{o}} \cdot v} \mapsto 1 \end{cases}
$$

Figure 4.4: Analysis of Listing 4.1, at line 14, assuming the loop has been unrolled once

intervals, we write $v \mapsto c$ as a shortcut for $v \mapsto [c, c]$. Any evaluation starts with the global operators $\mathbb{S}^\sharp[\![\,\cdot\,]\!], \mathbb{E}^\sharp[\![\,\cdot\,]\!]$, but we show in subscript which domain answers. In the initial state, two addresses are already allocated: $@^\sharp_{9,\mathsf{r}}$ was allocated before the loop (with v having value -1) and $@^\sharp_{14,\mathsf{r}}$ during the first unrolling (with v having value 1). We also know that $i = 2$ (we are in the second loop iteration). old and b currently point to the same record,

stored at $@^\sharp_{14,\mathsf{r}}$. The analyses starts with the statement $b = \texttt{new}\ \ Box$, at program location 14. This statement is analyzed by the environment domain, which starts by evaluating $\texttt{new}\ \ Box$ globally. The allocation of Box is handled by the heap abstraction (Figure 4.3), and starts by allocating a new address at program location 14. This allocation is handled by the recency abstraction (Figure 4.2). We are in the case where no previous old address exists at line 14. The address $@^\sharp_{14,\mathsf{o}}$ is thus added to the state, and a global renaming is triggered. The heap abstraction handles address renaming by delegation over the auxiliary field variable on v. In the resulting abstract state, $\sigma^\sharp_2$, the environment now maps old and b to $@^\sharp_{14,\mathsf{o}}$, and $\underline{@^\sharp_{14,\mathsf{o}} \cdot v}$ is 1. The result of the allocation evaluation is thus the recent address $@^\sharp_{14,\mathsf{r}}$ in the state $\sigma^\sharp_2$. Then, the transfer function of the Box allocation initializes the only field v of Box with any integer value, using an auxiliary address variable. This statement is handled by the environment domain, and the resulting domain is $\sigma^\sharp_3$. Only the environment is changed to introduce the binding $\underline{@^\sharp_{14,\mathsf{r}} \cdot v} \mapsto [-\infty, +\infty]$. The result of the Box allocation is $@^\sharp_{14,\mathsf{r}}$ in the state $\sigma^\sharp_3$. The environment domain can then update its local state with the binding $b \mapsto @^\sharp_{14,\mathsf{r}}$.

4.1.5 Concretizations

4.1.5.1 Recency abstraction

The recency abstraction transforms concrete states (sets of environments and heaps) using abstract addresses into concrete states using concrete addresses. Since the recency abstraction summarizes addresses, the concretization transforms a set of coherent concrete states into a single one, similarly to the concretization of the string summarization domain (Section 2.4.5). In order to ease the definition of the concretization, we define an auxiliary function n_ρ, giving the number of the most recently allocated address at location l in the concrete state ρ:

$$\mathrm{dom}\rho \cap \{\ @^i_l \mid l \in \mathsf{Loc}, i \in \mathbb{N}\ \} = \{\ @^i_l \mid l \in \mathsf{Loc}, 1 \leq i \leq n_\rho(l)\ \}$$

The concretization of the recency abstraction enforces the following constraints:

- bindings over objects that are not abstract addresses are kept, provided the input states are coherent on those objects.
- if an old address allocated at location l is in the state of the recency abstraction, it abstracts a given number $n_\rho(l) - 1$ of concrete addresses, to be picked in the input states of the old address.
- a recent address abstracts only one concrete address, which is moreover the most recent one allocated at this location. The set of input states must be coherent on this recent address.

$$\begin{aligned}
\gamma_{mem}(\sigma^\sharp_{mem}) = \{\ (R, \{\ \rho\ \}) \mid \\
v \in \mathrm{dom}R \setminus \{\ @^\sharp_{l,\mathsf{m}} \mid m \in \{\ \boldsymbol{r}, \boldsymbol{o}\ \}, l \in \mathsf{Loc}\ \} \Leftrightarrow |R(v)| = 1 \wedge \rho(v) = R(v); \\
@^o_l \in \sigma^\sharp_{mem} \Leftrightarrow n_\rho(l) > 1 \wedge \forall 1 \leq i \leq n_\rho(l) - 1, \rho(@^i_l) \in R(@^\sharp_{l,\mathsf{o}}) \\
@^r_l \in \sigma^\sharp_{mem} \Leftrightarrow n_\rho(l) \geq 1 \wedge |R(@^\sharp_{l,\mathsf{r}})| = 1 \wedge \rho(@^{n_\rho(l)}_l) = R(@^\sharp_{l,\mathsf{r}})\ \}
\end{aligned}$$

4.1.5.2 Heap abstraction

The heap abstract domain is stateless. Its concretization transforms a concrete environment using auxiliary address variables into a concrete environment that does not include auxiliary address variables anymore and a concrete heap. All auxiliary address variables are used to

define the heap, and the rest is kept in the environment. Since the concretization performs a one-to-one mapping, we lift it similarly to the string length domain in Section 2.5, using the $\uparrow$ operator defined in Remark 2.83.

$$\gamma_{heap}(()) = \uparrow \circ \{\, (e, (e', h')) \mid \underline{@^\sharp_{l,\mathsf{m}} \cdot a} \in \mathrm{dom} e \Leftrightarrow h'(@^\sharp_{l,\mathsf{m}}, a) = e(\underline{@^\sharp_{l,\mathsf{m}} \cdot a}); \\ v \in \mathrm{dom} e \setminus \{\, \underline{@^\sharp_{l,\mathsf{m}} \cdot \_} \,\} \Leftrightarrow e'(v) = e(v) \,\}$$

4.2 Variable policies for the recency abstraction

The recency abstraction was originally designed for static languages where types are fixed. In the case of more dynamic programming languages such as Python, the partitioning based on the allocation site may be too precise or imprecise, depending on the kind of analysis performed. While the semantics of Python programs is not yet established, we show a few motivating examples with Python programs, where only basic knowledge of Python is required.

When the allocation-site is not necessary. For example, let us assume that we are performing a type analysis on the program in Listing 4.2. There is no need to split `Task` allocation at line 6 by program location since we only want to know that `Task` instances have an integer `weight` attribute.

Listing 4.2: Python program computing the average of tasks

```
1  class Task:
2    def __init__(self, weight):
3      if weight < 0: raise ValueError
4        self.weight = weight
5
6  l = [Task(2), Task(1), Task(3), Task(5)]
7  m = 0
8  for i in range(len(l)):
9    m = m + l[i].weight
10 m = m  (i + 1)
```

When the allocation-site is imprecise. On the other hand, it is possible in Python to allocate objects of different types at the same allocation site since classes are first-class objects. For example, the function `logging_alloc` in Listing 4.3 logs the operation before creating the object. With the usual allocation-site abstraction, the addresses of `i` and `l` would be summarized into the same old block. This would trigger catastrophic imprecision in the analysis.

Listing 4.3: Python program, with logged allocations

```
1 def logging_alloc(x):
2     print("allocating " + x)
3     return x()
4
5 i = logging_alloc(int)
6 l = logging_alloc(list)
7 d = logging_alloc(dict)
```

Type-dependent precision. However, the allocation-site criterion still makes sense for containers such as lists: it would be too imprecise to abstract all lists using the same abstract address. Consider for example the program in Listing 4.4. The lists allocated at lines 1 and 2 would be summarized in the same abstract address after the allocation at line 3. The variables summarizing list contents are defined using the abstract address of the list. The list contents

of `qty` and `els` would thus be summarized in the same variable, meaning that all list accesses to `qty` or `els` would have to return both integers and strings, which lacks precision. In this case, it would be beneficial to partition abstract addresses by allocation site. In some cases, we can even go a step further and partition addresses by (partial) callstacks. A callstack-based partitioning would resolve the imprecision issue in the analysis of Listing 4.3. This shows that the usual homogeneous partitioning could benefit from different allocation precision depending on the type of address abstracted. In the case of containers, we do not try to take into account the type of elements, since Python containers can hold heterogeneous values (cf. Chapter 5).

Listing 4.4: Python program with lists of different types

```
1 qty = [3, 1, 1]
2 els = ['choc', 'flour', 'egg']
3 exp = []
4 for i in range(len(qty)):
5   for j in range(qty[i]):
6     exp.append(els[i])
```

Thus, we extend abstract addresses to be typed ($\mathbb{T}$ is a finite set of types) and to take optionally into account location and partial callsite information (written $\mathbb{C}$). Then, the recency abstraction is parameterized by an allocation policy, describing how addresses are abstracted.

$$\mathsf{Addr}^\sharp = \mathbb{T} \times \mathsf{Loc}^\top \times \mathbb{C}^\top \times \{\, \mathsf{r}, \mathsf{o} \,\}$$

Definition 4.9 **Allocation policy**

An allocation policy is a function $\pi \in \mathbb{T} \times \mathsf{Loc} \times \mathbb{C} \times \{\, \mathsf{r}, \mathsf{o} \,\} \rightharpoonup \mathsf{Addr}^\sharp$ defining how allocation-site and callstack information are abstracted to create an abstract address.

Two basics policies that can be used are the type-only policy and the location policy, defined below. We build upon these two policies to define those used in the type and value analyses of Python. The type analysis is described in Chapter 7. It aims at detecting type-related errors by inferring typed points-to information for variables. The value analysis will extend the type analysis in Chapter 8 to infer numerical invariants for variables holding builtin values.

Example 4.10 **Type-only policy**

The type-only policy $\pi_{to} = \lambda(t, l, c, m).(t, \top, \top, m)$ keeps only the type information. It is interesting for type analyses since it keeps essential information but it is not too expensive either.

Example 4.11 **Location policy**

The location policy $\pi_{loc} = \lambda(t, l, c, m).(t, l, \top, m)$ discards callstack information and keeps the rest.

Policy for Python's type analysis. The policy for the type analysis is defined below. It uses allocation-site information to handle containers that are generic in their contents in order to avoid the precision issues mentioned before. For the rest, it uses the type-only policy to be memory-efficient.

$$\pi^{\text{types}}(t, l, c, m) = \begin{cases} \pi_{loc}(t, l, c, m) & \text{if } t \in \{\, \text{list, tuple, dict} \,\} \\ \pi_{to}(t, l, c, m) & \text{otherwise} \end{cases}$$

Policy for Python's value analysis. The standard policy for the value analysis is the location policy π_{loc}. A coarser alternative, π_{to}^{values} can be defined, where the allocation site is only used for the `range, slice, list, tuple, dict` builtins (in order to be sufficiently precise on the numerical values that `range` and `slice` iterators hold).

$$\pi_{to}^{\text{values}}(t,l,c,m) = \begin{cases} \pi_{loc}(t,l,c,m) & \text{if } t \in \{\text{ range, slice, list, tuple, dict }\} \\ \pi_{to}(t,l,c,m) & \text{otherwise} \end{cases}$$

The transfer function of the recency abstraction is slightly changed with policies: abstract addresses are created by the chosen policy. We assume the type and callstack are provided to the *alloc_addr* expression. The new transfer function is shown in Figure 4.5.

$$\begin{array}{l}
\mathbb{E}^{\sharp}_{\text{mem}}[\![\, \mathit{alloc_addr}(t,l,c) \,]\!]\sigma^{\sharp} = \\
\quad \text{let } \sigma^{\sharp}_{mem} = \texttt{man.get } \sigma^{\sharp} \text{ in} \\
\quad \text{let } @^{\sharp}_{\mathsf{r}} = \pi(t,l,c,r) \text{ in} \\
\quad \text{let } @^{\sharp}_{\mathsf{o}} = \pi(t,l,c,o) \text{ in} \\
\quad \text{if } @^{\sharp}_{\mathsf{r}} \in \sigma^{\sharp}_{mem} \text{ then} \\
\qquad \text{if } @^{\sharp}_{\mathsf{o}} \in \sigma^{\sharp}_{mem} \text{ then } @^{\sharp}_{\mathsf{r}}, \mathbb{S}^{\#}[\![\, \texttt{fold}(@^{\sharp}_{\mathsf{o}}, @^{\sharp}_{\mathsf{r}}) \,]\!]\sigma^{\sharp} \\
\qquad \text{else} \\
\qquad\quad \text{let } \sigma^{\sharp}_{mem} = \sigma^{\sharp}_{mem} \cup \{\, @^{\sharp}_{\mathsf{o}} \,\} \text{ in} \\
\qquad\quad @^{\sharp}_{\mathsf{r}}, \mathbb{S}^{\#}[\![\, \texttt{rename}(@^{\sharp}_{\mathsf{r}}, @^{\sharp}_{\mathsf{o}}) \,]\!](\texttt{man.set } \sigma^{\sharp}_{mem} \, \sigma^{\sharp}) \\
\quad \text{else} \\
\qquad \text{let } \sigma^{\sharp}_{mem} = \sigma^{\sharp}_{mem} \cup \{\, @^{\sharp}_{\mathsf{r}} \,\} \text{ in} \\
\qquad @^{\sharp}_{\mathsf{r}}, \texttt{man.set } \sigma^{\sharp}_{mem} \, \sigma^{\sharp}
\end{array}$$

Figure 4.5: Semantics of address allocation with policies

These policies will be compared in Sections 7.5 and 8.2.

4.3 Abstract garbage collection (AGC)

In our first implementation of the recency abstraction, we were only performing address allocations. Even if the addresses were unused after a given program point, they were kept until the end of the analysis. However, when we instrumented our analysis (using a hook in Mopsa, cf. Section 3.4), we noticed that up to two-thirds of the allocated addresses were unreachable. Most objects are stored in local variables and can thus be cleaned after the function returns. Therefore we decided to implement an abstract garbage collector, detecting and removing unreachable abstract addresses.

Listing 4.5: Motivating example for abstract garbage collection

```
1  class A:
2    def __init__(self, v):
3      self.v = v
4
5  def f(i):
6    b = A(i)
7    c = A(i+1)
8    return b.v
9
10 r1 = f(0)
11 r2 = f(100)
```

Example 4.12 **Precision loss with old addresses**

Let us consider the program of Listing 4.5, where we assume a type-only policy applies to instances of class `A` (abstract addresses will thus be written $@^\sharp_{A,\mathsf{r}}$ and $@^\sharp_{A,\mathsf{o}}$). After the first call to f, both an old and a recent address are allocated for `A`. After the two allocations in the second call to f, we know that $c.v$ is 101, thanks to the precision of the last allocation, ensuring a strong update is performed (in the abstract, this is represented as $\underline{@^\sharp_{A,\mathsf{r}} \cdot v \mapsto [101, 101]}$). However, we can only infer that $\underline{@^\sharp_{A,\mathsf{o}} \cdot v \mapsto [0, 100]}$ due to the previous allocations in the first call.

In general, performing abstract garbage collection can improve the precision of the analysis since it may remove old addresses where only weak updates can be performed. We previously found [112] that the precision gains were not sufficient to remove false alarms in practice. They were, however, extremely beneficial from a performance standpoint: AGC halved the memory usage and brought a 38% analysis time improvement (with the AGC taking less than 6% of the analysis time). These results will be detailed in Sections 7.6.3 and 8.2.3.

Example 4.13 **Improving precision**

Going back to the example of Listing 4.5, we assume an AGC is performed after the first call to `f`. Since both recent and old addresses for $@^\sharp_{A,\mathsf{r}}, @^\sharp_{A,\mathsf{o}}$ are not reachable outside `f`, we can safely remove them. After the two allocations in the second call, we have: $\underline{@^\sharp_{A,\mathsf{o}} \cdot v \mapsto [100, 100]}$, $\underline{@^\sharp_{A,\mathsf{r}} \cdot v \mapsto [101, 101]}$, which is more precise.

Implementation. We have implemented the abstract garbage collection within the recency abstract domain. Due to the distributed structure of abstract domains in Mopsa, the abstract garbage collector asks domains about the set of live addresses they use through a query, defined below.

```
type ('a,_) query +=
  | Q_alive_addresses : ('a,addr list) query
```

The recency then performs set difference between its local state of allocated addresses and the result of the query. It can then remove the unreachable addresses. By default, the AGC is called after each assignment where the right-hand side is a (function, method, or object) call. We have tested running the AGC at only a fraction of those assignments, but the results were not as satisfying.

4.4 Related work

Dynamic memory allocation abstractions. Liang et al. [96] find that the recency abstraction leads to high precision in 75% percent of their cases. They also study precision improvements when the last n allocated blocks are kept separate. It thus seems natural that most previous analyses of dynamic programming languages taking soundly into account object mutation and aliasing used the recency abstraction [74, 54] over the allocation-site abstraction. Specialized heap abstractions for dynamic programming languages exist, such as the heap with open objects [39], aiming at precisely analyzing code where object attributes are unknown. In our case, we analyze Python programs through a top-down, context-sensitive analysis, and we have not encountered the need to analyze code where object attributes are unknown. Park et al. [124] observe that a more precision partitioning of abstract addresses does not always ensure a more precise analysis relying on the recency abstraction. They define a singleton abstraction resolving this issue. To the best of our knowledge, only Park et al. [124] and our work formally define a concretization function for the recency abstraction. Heap abstractions specialized to complex data structures (such as linked lists) are called shape analyses, surveyed by Chang et al. [25].

Abstract Garbage Collection. Abstract garbage collectors were first used in the analysis of higher-order languages. They are first used by Jagannathan et al. [71], and then described by Might and Shivers [104]. More recently, an abstract garbage collector based on reference counting was presented by Es et al. [51]. In the setting of higher-order languages, using an abstract garbage collector has been found to reduce the analysis time by an order of magnitude and sometimes to improve the precision. In TAJS, Jensen et al. [74] find that their abstract garbage collector reduces their memory consumption but has little impact on the precision or the analysis time.

4.5 Conclusion

In this chapter, we defined the recency abstraction and its concretization modularly. We showed how it would interact with other domains, by extending our toy language with OCaml-like records. We motivated the need for different partitionings of the abstract addresses in the case of the analysis of dynamic programming languages such as Python. We defined the notion of allocation policy, and provided instantiations of these policies, which will be used in the analyses of Python programs defined later. We showed that the recency abstraction can be combined with an abstract garbage collection mechanism, improving the precision of the analyses using it. The next chapter focuses on the analysis of containers in the setting of dynamic programming languages. These analyses rely on a dynamic memory allocation abstraction, such as the one presented here.

5

Abstracting Containers

This chapter defines containers' abstractions, that will be used in the Python analyses of Chapters 7 and 8. These abstractions and the concretizations are defined modularly, as we have done before. They work by delegation to other domains and do not make specific assumptions on them. Previous works (presented in Section 5.3) handle a setting typical to static programming languages: arrays are of fixed size, and these works aim at inferring numerical properties between numerical arrays and their indexes. The abstractions presented here are based on usual summarization (also called smashing) techniques and are not really complex. However, they handle specific features of dynamic languages: containers have a dynamic size (arrays' elements can be added and removed) and can hold heterogeneous values (including non-numeric values). Designing more precise container abstractions is an important part of our future work. We present abstractions on a simplified language; they can be easily lifted to a Python-specific setting. We focus on arrays (called lists in Python) in Section 5.1. We define length and summarization abstractions that are close to the ones defined for strings in Sections 2.4.2 and 2.4.5. Contrary to strings, they rely on a domain handling dynamic memory allocation, such as the recency abstraction of Section 4.1. They delegate all operations to underlying domains through rewriting using auxiliary variables. In the case of the summarization abstraction, this delegation means the domain is generic in the abstracted content. We define a summarization of dictionaries in Section 5.2.

5.1 Dynamic arrays

5.1.1 Array operations

Arrays are mutable and allocated on the heap. They can hold heterogeneous values (such as strings and integers). The static typing of `Imp` is unfit to handle heterogeneous arrays (we could introduce some gradual types, but it is beside the point of our toy language). Instead, we show our examples in a Python-like setting. Still, we do not focus on Python specifics, and we assume that functions are called on arguments having correct types. We consider that erroneous programs stop their execution rather than raise an exception as Python does.

The arrays presented here support the following operations:

▷ **Creation,** written $[e_1, \ldots, e_n]$, where e_i are expressions. It is possible to create empty lists, [].
▷ **Element addition,** $a.append(x)$ adds x to the end of the array a in place. Calls to $append$ are statements.
▷ **Membership testing,** $a.contains(x)$ returns 1 if and only if x is an element of a, and 0 otherwise.

▷ **Length,** $a.len()$ returns a non-negative integer being the number of elements of the array.
▷ **Index access,** $a.getitem(i)$ returns the element of a at position i.
▷ **Element removal,** $a.delitem(i)$ removes the i-th element from array a (the elements after this one are all moved to their predecessor's position). Calls to $delitem$ are statements.

The corresponding names of methods in Python are prefixed and suffixed by a double underscore ("contains" is "__contains__", etc.), except for "append".

Definition 5.1 **Program State**

The program state consists of an environment and heap (in a setting similar to the concrete state of Section 4.1.2). The environment maps variables to values or addresses. The heap maps addresses to objects. The only supported kind of object is array, which consists of a number of values or addresses.

$$\mathcal{E} = \mathcal{V} \to \textsf{Value} \cup \textsf{Addr} \qquad \mathcal{H} = \textsf{Addr} \to \textsf{Obj}$$

$$\textsf{Obj} = Array(c \in (\textsf{Value} \cup \textsf{Addr})^*)$$

The records of Section 4.1.2 could be added in the state by extending **Obj** with $Record(s \in string \to \textsf{Value})$.

The semantics of arrays is shown in Figure 5.1. To create an array, an address is allocated, and all the elements are evaluated, before the updated heap is returned. $append$ evaluates the array into an address, as well as the element to add, before updating the heap. $getitem$ evaluates the array and the index. If the index is valid, the corresponding element is returned. To compute the length, we just count the number of elements of the array. $contains$ returns 1 provided any of the array's elements match the element, and 0 otherwise. The definition of the equality comparison operator is left implicit in the semantics. In the case of Python, the semantics of the equality comparison will be shown in Section 6.2.15.2. $delitem$ evaluates the array and the index, before updating the heap, provided the index is valid.

Remark 5.2 **Random value notation**

To avoid confusion between arrays and `Imp`'s intervals, we use the notation $rand(l, u)$ for a random number within the interval $[l, u]$.

Example 5.3 **Running examples**

The program of Listing 5.1 creates an array a of size n, with all its elements in between the picked values of l and u. A program copying array a into b is shown in Listing 5.2. It does not assume anything on a and copies the arrays by iterating over the indexes of a.

Listing 5.1: Program random initilizating array a

```
n = rand(0, 2021)
l = rand(-100, 0)
u = rand(0, 100)

a = []
i = 0
while i < n:
  a.append(rand(l, u))
  i = i + 1
```

$$
\begin{array}{l}
\mathbb{E}[\![\, [x_1, \ldots, x_n] \,]\!](e,h) = \\
\quad \text{let } @, (e,h) = \mathbb{E}[\![\, \mathit{alloc\_addr}() \,]\!](e,h) \text{ in} \\
\quad \text{let } v_0, (e,h) = \mathbb{E}[\![\, x_1 \,]\!](e,h) \text{ in} \\
\quad \ldots \\
\quad \text{let } v_{n-1}, (e,h) = \mathbb{E}[\![\, x_n \,]\!](e,h) \text{ in} \\
\quad @, (e, h[@ \mapsto \mathit{Array}(v_0, \ldots, v_{n-1})]) \\
\mathbb{S}[\![\, a.\mathit{append}(x) \,]\!](e,h) = \\
\quad \text{let } @, (e,h) = \mathbb{E}[\![\, a \,]\!](e,h) \text{ in} \\
\quad \text{let } \mathit{Array}(v_0, \cdots, v_{n-1}) = h(@) \text{ in} \\
\quad \text{let } v = \mathbb{E}[\![\, x \,]\!](e,h) \text{ in} \\
\quad (e, h[@ \mapsto \mathit{Array}(v_0, \ldots, v_{n-1}, v)]) \\
\mathbb{E}[\![\, a.\mathit{getitem}(i) \,]\!](e,h) = \\
\quad \text{let } @, (e,h) = \mathbb{E}[\![\, a \,]\!](e,h) \text{ in} \\
\quad \text{let } \mathit{Array}(v_0, \cdots, v_{n-1}) = h(@) \text{ in} \\
\quad \text{let } i, (e,h) = \mathbb{E}[\![\, i \,]\!](e,h) \text{ in} \\
\quad \text{if } 0 \leq i < n \text{ then } v_i, (e,h) \text{ else } \bot, (e,h)
\end{array}
$$

$$
\begin{array}{l}
\mathbb{E}[\![\, a.\mathit{len}() \,]\!](e,h) = \\
\quad \text{let } @, (e,h) = \mathbb{E}[\![\, a \,]\!](e,h) \text{ in} \\
\quad \text{let } \mathit{Array}(v_0, \ldots, v_{n-1}) = h(@) \text{ in} \\
\quad n, (e,h) \\
\mathbb{E}[\![\, a.\mathit{contains}(x) \,]\!](e,h) = \\
\quad \text{let } @, (e,h) = \mathbb{E}[\![\, a \,]\!](e,h) \text{ in} \\
\quad \text{let } \mathit{Array}(v_0, \ldots, v_{n-1}) = h(@) \text{ in} \\
\quad \mathbb{E}[\![\, 1 \,]\!] \circ \mathbb{C}[\![\, x == v_0 \text{ or } \cdots \text{ or } x == v_{n-1} \,]\!](e,h) \cup \\
\quad \mathbb{E}[\![\, 0 \,]\!] \circ \mathbb{C}[\![\, x! = v_0 \text{ and } \cdots \text{ and } x! = v_{n-1} \,]\!](e,h) \\
\mathbb{S}[\![\, a.\mathit{delitem}(i) \,]\!](e,h) = \\
\quad \text{let } @, (e,h) = \mathbb{E}[\![\, a \,]\!](e,h) \text{ in} \\
\quad \text{let } \mathit{Array}(v_0, \ldots, v_{n-1}) = h(@) \text{ in} \\
\quad \text{let } i, (e,h) = \mathbb{E}[\![\, i \,]\!](e,h) \text{ in} \\
\quad \text{if } 0 \leq i < n \text{ then} \\
\quad (e, h[@ \mapsto \mathit{Array}(v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{n-1})]) \\
\quad \text{else } \bot
\end{array}
$$

Figure 5.1: Semantics of arrays

Listing 5.2: Program copying array a into b

```
b = []
i = 0
while i < a.len():
  b.append(a.getitem(i))
  i = i + 1
```

5.1.2 Length abstraction

The length abstraction uses the auxiliary address variable $\underline{\text{alen}}(@^\sharp)$ to denote the length of the array referenced at address $@^\sharp$. The domain is stateless, and its transfer functions are shown in Figure 5.2. During an array allocation, the domain asks for the allocation of an abstract address $@^\sharp$, and then delegates the assignment $\underline{\text{alen}}(@^\sharp) = n$. Calling *append* increases the length by one (which is a delegated assignment). The length domain does not keep track of the elements of the array. As such, it is imprecise when *contains* is called. On the other hand, it returns its auxiliary length variable when *len* is called. On calls to *getitem*, we ensure that the index is valid and return $\top$. For calls to *delitem*, the length is decreased by one provided that i is a valid index.

Remark 5.4 **Ghost addressing renaming**

Since each domain is responsible for its ghost variables, the array length domain handles the renaming of its ghost variables.

$$\mathbb{S}^\sharp_{alen}[\![\, \texttt{rename}(@^\sharp_1, @^\sharp_2) \,]\!] = \mathbb{S}^\sharp[\![\, \texttt{rename}(\underline{\text{alen}}(@^\sharp_1), \underline{\text{alen}}(@^\sharp_2)) \,]\!]$$

This renaming by delegation is similar to the heap domain of Section 4.1.4 and mentioned in Remark 3.6.

$$
\begin{aligned}
&\mathbb{E}^\sharp_{alen}[\![\, [e_1, \ldots, e_n] \,]\!]\sigma^\sharp = \\
&\quad \text{let } @^\sharp, \sigma^\sharp = \mathbb{E}^\sharp[\![\, \mathit{alloc\_addr}() \,]\!]\sigma^\sharp \text{ in } @^\sharp, \mathbb{S}^\sharp[\![\, \underline{\text{alen}}(@^\sharp) = n \,]\!]\sigma^\sharp \\
&\mathbb{S}^\sharp_{alen}[\![\, a.append(x) \,]\!]\sigma^\sharp = \\
&\quad \text{let } @^\sharp, \sigma^\sharp = \mathbb{E}^\sharp[\![\, a \,]\!]\sigma^\sharp \text{ in } \mathbb{S}^\sharp[\![\, \underline{\text{alen}}(@^\sharp) = \underline{\text{alen}}(@^\sharp) + 1 \,]\!] \\
&\mathbb{E}^\sharp_{alen}[\![\, a.contains(x) \,]\!]\sigma^\sharp = \mathbb{E}^\sharp[\![\, [0, 1] \,]\!]\sigma^\sharp \\
&\mathbb{E}^\sharp_{alen}[\![\, a.len() \,]\!]\sigma^\sharp = \\
&\quad \text{let } @^\sharp, \sigma^\sharp = \mathbb{E}^\sharp[\![\, a \,]\!]\sigma^\sharp \text{ in } \mathbb{E}^\sharp[\![\, \underline{\text{alen}}(@^\sharp) \,]\!] \\
&\mathbb{E}^\sharp_{alen}[\![\, a.getitem(i) \,]\!]\sigma^\sharp = \\
&\quad \text{let } @^\sharp, \sigma^\sharp = \mathbb{E}^\sharp[\![\, a \,]\!]\sigma^\sharp \text{ in } \mathbb{E}^\sharp[\![\, \top \,]\!] \circ \mathbb{C}^\sharp[\![\, 0 \leq i < \underline{\text{alen}}(@^\sharp) \,]\!]\sigma^\sharp \\
&\mathbb{E}^\sharp_{alen}[\![\, a.delitem(i) \,]\!]\sigma^\sharp = \\
&\quad \text{let } @^\sharp, \sigma^\sharp = \mathbb{E}^\sharp[\![\, a \,]\!]\sigma^\sharp \text{ in } \mathbb{S}^\sharp[\![\, \underline{\text{alen}}(@^\sharp) = \underline{\text{alen}}(@^\sharp) - 1 \,]\!] \circ \mathbb{C}^\sharp[\![\, 0 \leq i < \underline{\text{alen}}(@^\sharp) \,]\!]\sigma^\sharp
\end{aligned}
$$

Figure 5.2: Array length abstraction

Example 5.5 **Array initialization**

Going back to the example of Listing 5.1, our analysis – using an interval domain – will infer that $n \geq 0$ and that the length of the array a is non-negative too. With a relational numerical domain such as polyhedra or octagons, the first fixpoint iteration yields:

$$\left(\underline{\text{alen}}(@^\sharp) = 1, i = 1, 1 \leq n \leq 2021\right)$$

The fixpoint is reached at the second iteration, and the state at the end of the loop's body is:

$$\left(1 \leq \underline{\text{alen}}(@^\sharp) = i \leq n \leq 2021\right)$$

Intersected with the negation of the loop guard, we know that $0 \leq n = i = \underline{\text{alen}}(@^\sharp) \leq 2021$ at the end of the program, and thus that our array has length n.

Example 5.6 **Array copy**

On the example of array copy in Listing 5.2, an analysis using an underlying relational numerical domain can infer that a and b have the same length. Since we have a similar invariant to the previous example, we can also prove that the index access is correct.

Remark 5.7 **Empty lists**

We could be a bit more precise for the membership test: if the array is empty (of length 0), it contains nothing, thus, the function $\mathit{contains}$ returns 0.

Definition 5.8 **Concretization**

The concretization of the array length domain transforms states with auxiliary array length variables into concrete states with arrays. The concretization is similar to that of the string length domain of Section 2.4.2 and of the heap abstraction of Section 4.1.5.2. We lift a one-to-one mapping using the $\uparrow$ operator.

$$\gamma_{alen}(\,()\,) = \uparrow \circ \{\, (\rho, \rho') \mid v \in \mathsf{dom}\rho \setminus \{\, \underline{\mathsf{alen}}(\_)\,\} \Leftrightarrow \rho'(v) = \rho(v);$$
$$\underline{\mathsf{alen}}(@^\sharp) \in \mathsf{dom}\rho \Leftrightarrow \rho'(@^\sharp) = Array(v_1, \cdots, v_{\rho(\underline{\mathsf{alen}}(@^\sharp))}) \wedge$$
$$\forall i \in [1, n], v_i \in \mathsf{Value}^\sharp \cup \mathsf{Addr}^\sharp \,\}$$

5.1.3 Summarization abstraction

This abstraction summarizes the contents of arrays allocated at $@^\sharp$ using a single auxiliary variable $\underline{\mathsf{arr}}(@^\sharp)$ to denote elements at all indices. The transfer functions are shown in Figure 5.3. Array allocations are rewritten into successive weak updates of $\underline{\mathsf{arr}}(@^\sharp)$ once an address has been allocated. Adding a new element is delegated as a weak assign (which is ultimately a fold operation). $contains$ is evaluated as a disjunction depending on the filtering over the auxiliary variable. This domain does not keep track of lengths and thus returns any non-negative integer. Accessing a given index is done by returning the auxiliary variable, tagged weak (so it will be expanded into a fresh variable during assignments, as explained in Remark 2.75). As usual, renamings of array addresses are delegated as renamings of the corresponding auxiliary variable. The summarization domain does not track which values are held at which index, and thus its transfer function for $delitem$ is the identity.

$$\mathbb{E}^\sharp_{arr}[\![\, [e_1, \ldots, e_n] \,]\!]\sigma^\sharp =$$
$$\quad \text{let } @^\sharp, \sigma^\sharp = \mathbb{E}^\sharp[\![\, alloc\_addr() \,]\!]\sigma^\sharp \text{ in}$$
$$\quad @^\sharp, \mathbb{S}^\sharp[\![\, \underline{\mathsf{arr}}(@^\sharp)_{\mathtt{weak}} = e_n \,]\!] \circ \cdots \circ \mathbb{S}^\sharp[\![\, \underline{\mathsf{arr}}(@^\sharp)_{\mathtt{weak}} = e_2 \,]\!] \circ \mathbb{S}^\sharp[\![\, \underline{\mathsf{arr}}(@^\sharp) = e_1 \,]\!]\sigma^\sharp$$
$$\mathbb{S}^\sharp_{arr}[\![\, a.append(x) \,]\!]\sigma^\sharp =$$
$$\quad \text{let } @^\sharp, \sigma^\sharp = \mathbb{E}^\sharp[\![\, a \,]\!]\sigma^\sharp \text{ in } \mathbb{S}^\sharp[\![\, \underline{\mathsf{arr}}(@^\sharp)_{\mathtt{weak}} = x \,]\!]\sigma^\sharp$$
$$\mathbb{E}^\sharp_{arr}[\![\, a.contains(x) \,]\!]\sigma^\sharp =$$
$$\quad \text{let } @^\sharp, \sigma^\sharp = \mathbb{E}^\sharp[\![\, a \,]\!]\sigma^\sharp \text{ in}$$
$$\quad \mathbb{E}^\sharp[\![\, 1 \,]\!] \circ \mathbb{C}^\sharp[\![\, \underline{\mathsf{arr}}(@^\sharp)_{\mathtt{weak}} == x \,]\!]\sigma^\sharp \sqcup^\sharp \mathbb{E}^\sharp[\![\, 0 \,]\!] \circ \mathbb{C}^\sharp[\![\, \underline{\mathsf{arr}}(@^\sharp)_{\mathtt{weak}} != x \,]\!]\sigma^\sharp$$
$$\mathbb{E}^\sharp_{arr}[\![\, a.len() \,]\!] = \mathbb{E}^\sharp[\![\, rand(0, +\infty) \,]\!]$$
$$\mathbb{E}^\sharp_{arr}[\![\, a.getitem(i) \,]\!]\sigma^\sharp =$$
$$\quad \text{let } @^\sharp, \sigma^\sharp = \mathbb{E}^\sharp[\![\, a \,]\!]\sigma^\sharp \text{ in}$$
$$\quad \mathbb{E}^\sharp[\![\, \underline{\mathsf{arr}}(@^\sharp)_{\mathtt{weak}} \,]\!]\sigma^\sharp$$
$$\mathbb{E}^\sharp_{arr}[\![\, a.delitem(i) \,]\!]\sigma^\sharp = \sigma^\sharp$$
$$\mathbb{S}^\sharp_{arr}[\![\, \mathtt{rename}(@^\sharp_1, @^\sharp_2) \,]\!] = \mathbb{S}^\sharp[\![\, \mathtt{rename}(\underline{\mathsf{arr}}(@^\sharp_1), \underline{\mathsf{arr}}(@^\sharp_2)) \,]\!]$$

Figure 5.3: Array smashing abstraction

Remark 5.9 **Empty arrays**

The case of empty arrays is not formally defined in the abstraction. One way to handle precisely this case is to delay the introduction of the $\underline{\mathsf{arr}}(@^\sharp)$ auxiliary variable until the array is not empty. This way, we avoid initializing it with spurious values. When the array becomes non empty, after a call to $append$, the variable is introduced and can be initialized with a precise, strong update. This requires extra bookkeeping in the abstract state to

remember which variables currently exist (i.e., which arrays are empty), and to handle the case of joining abstract states with different sets of variables. The case of heterogeneous numerical environments has been extensively studiend by Journault [80, Chapter 6], including the complex case of joining relational abstract states over different sets of variables. Hence, we omit the discussion here and settle for illustrative examples.

Example 5.10 **Array initialization**

Let us assume we use a relational numerical abstract domain such as polyhedra or octagons. In that case, the following invariant is trivially reached on the analysis of Listing 5.1:

$$\Big(-100 \leq l \leq \underline{\text{arr}}(@^\sharp) \leq u \leq 100\Big)$$

Example 5.11 **Array copy**

Let $@^\sharp_a$ (resp. $@^\sharp_b$) be the abstract address of array a (resp. b). Sice $\underline{\text{arr}}(@^\sharp_a)$ and $\underline{\text{arr}}(@^\sharp_b)$ are weak variables, we cannot infer that $\underline{\text{arr}}(@^\sharp_a) = \underline{\text{arr}}(@^\sharp_b)$ at the end of Listing 5.2. However, if a has been created using the random initialization of Listing 5.1, the analysis may infer:

$$(-100 \leq l \leq \underline{\text{arr}}(@^\sharp_a) \leq u \leq 100, -100 \leq l \leq \underline{\text{arr}}(@^\sharp_b) \leq u \leq 100)$$

Remark 5.12 **State refinement membership testing**

Let us assume we evaluate $a.contains(10)$, assuming the abstract state is $a \mapsto @^\sharp, 0 \leq \underline{\text{arr}}(@^\sharp) \leq 10$. Since the summarization only provides an overapproximation of the values of the array, there is no way to know if 10 is actually in the array: it could also be an imprecision from the numerical part, or it could be an element that has been added and removed later on. Thus, $contains$ returns both 1 and 0:

- 1, in the same state (the state cannot be refined to express that 10 is always in the array);
- 0, in a reduced state where $0 \leq \underline{\text{arr}}(@^\sharp) < 10$ (we ensure the array does not contain 10 anymore).

Definition 5.13 **Concretization**

The concretization of the array length domain transforms environments with auxiliary array variables into concrete states with arrays. It shares a definition similar to the concretizations of the string summarization domain of Section 2.4.5 or the recency abstraction of Section 4.1.5.1.

$$\gamma_{arr}(\texttt{()}) = \{\, (R, \rho) \mid v \in \text{dom}R \setminus \{\, \underline{\text{arr}}(\_) \,\} \Leftrightarrow |R(v)| = 1 \wedge \rho(v) = R(v);$$
$$\underline{\text{arr}}(@^\sharp) \in \text{dom}R \Leftrightarrow \exists n \in \mathbb{N}, \rho(@^\sharp) = Array(v_1, \cdots, v_n) \wedge \forall i \in [1, n], v_i \in R(\underline{\text{arr}}(@^\sharp)) \,\}$$

Remark 5.14 **Nested arrays**

If the abstract address allocation is sensitive to the allocation site, each nested array will be allocated at a different address, which is precise but sometimes costly. For example, `m = [[0], [1], [2]]` allocates four different addresses in our current implementation. In

addition, the evaluation of `m[i]` will return a disjunction of three elements corresponding to each list.

Example 5.15 **Iteratively nesting arrays**

We consider the program of Listing 5.3. Two random numbers l and u are picked, and three elements in between l and u are stored in array a. The array is then nested in itself ten times.

Listing 5.3: Iteratively nesting arrays

```
1 l, u = rand(-100, 0), rand(0, 100)
2 a = [rand(l, u), rand(l, u), rand(l, u)]
3 for i in range(10): a = [a]
```

At the beginning of the loop, we have $a \mapsto @^\sharp_{2,\mathsf{r}}$ in our environment and $l \leq \underline{\mathsf{arr}}(@^\sharp_{2,\mathsf{r}}) \leq u$ in the numerical domain. After one loop iteration, the numerical domain is unchanged, and the environment becomes

$$\left(a \mapsto @^\sharp_{3,\mathsf{r}}, \underline{\mathsf{arr}}(@^\sharp_{3,\mathsf{r}}) \mapsto \underline{\mathsf{arr}}(@^\sharp_{2,\mathsf{r}})\right)$$

The final state is unchanged for the numerical part, and the environment is

$$\left(a \mapsto @^\sharp_{3,\mathsf{r}}, \underline{\mathsf{arr}}(@^\sharp_{3,\mathsf{r}}) \mapsto \underline{\mathsf{arr}}(@^\sharp_{3,\mathsf{o}}), \underline{\mathsf{arr}}(@^\sharp_{3,\mathsf{o}}) \mapsto \left\{ \underline{\mathsf{arr}}(@^\sharp_{2,\mathsf{r}}), \underline{\mathsf{arr}}(@^\sharp_{3,\mathsf{r}}) \right\}\right)$$

Thanks to its old address, the recency abstraction generalizes the nesting. In the end, a is an array of depth at least 2, and the contents of the scalar array correspond to the elements allocated at line 3.

Keeping the values l and u symbolic, the concrete states of the previous abstract state are shown below. Following the abstract state, a maps to the most recent allocated variable at line 3, written $@^n_3$ in the concrete ($n > 1$ since both the old and recent abstract addresses exist). There is only one array allocated at line 2, whose address is written $@^1_2$. It contains an array of integers between l and u. Since the summarization abstraction does not track length, the array can have any length. The concretization of the array at $@^\sharp_{3,\mathsf{r}}$ is the array at $@^n_3$. This array contains only references to arrays previously allocated at the same program point, i.e., having concrete addresses $(@^i_3)_{1 \leq i < n}$. In turn, the contents of the previously allocated arrays point either to most recently allocated array at line 2 or at line 3.

$$\bigcup_{n>1} \Big\{ (e,h) \;\Big|\; e(a) = @^n_3,$$
$$\exists s \in \mathbb{N}, \exists (v_1, \ldots, v_s) \in \mathbb{Z}^s, h(@^1_2) = Array(v_1, \ldots, v_s), \forall j \in [1,s], l \leq v_j \leq u;$$
$$\exists m \in \mathbb{N}, \exists (a_1, \ldots, a_m) \in \mathsf{Addr}^m,$$
$$h(@^n_3) = Array(a_1, \ldots, a_m) \wedge \forall j \in [1,m], a_j \in \{\, @^i_3 \mid i \in [1, n-1] \,\};$$
$$\forall i \in [1, n-1], \exists m_i \in \mathbb{N}, (a^i_1, \ldots, a^i_{m_i}) \in \mathsf{Addr}^m,$$
$$h(@^i_3) = Array(a^i_1, \ldots, a^i_{m_i}) \wedge \forall j \in [1, m_i], a^i_j \in \{\, @^1_2, @^n_3 \,\} \Big\}$$

5.1.4 Reduced product

The reduced product between the array length and summarization domains is close to the one defined on strings in Section 2.4.6. There are similar cases of diverging states of the underlying numerical domain, which are merged just as the string domains did. These merges happen during array allocations, element deletion, index accesses and element addition. Since both

domains are stateless, there is no need to define specific state reductions. However, we still need to define reductions between the evaluations of expressions. We describe informally how these evaluation reductions are performed:

▷ **Array allocation.** Both domains return the same allocated address $@^\sharp$, and the evaluation reduction is thus straightforward.
▷ **Membership test.** The summarization domain returns either 0 or 1 with different states, while the length domain returns $[0, 1]$ (except if the list is empty; it returns 0 in that case). The evaluation reduction intersects those results.
▷ **Length.** The summarization domain returns $[0, +\infty]$ and the length domain an auxiliary address variable. The latter is kept by the reduction.
▷ **Index access.** The summarization domain returns an auxiliary variable, and the length domain returns top. The reduction keeps the result of the summarization domain.

Example 5.16 **Reduced product of array abstractions**

Let us assume that a is the randomly initialized array of Listing 5.1, and that i is a positive integer. We study the analysis of $a.getitem(i)$, with the following input state:

$$\Big(-100 \leq l \leq \underline{\text{arr}}(@^\sharp) \leq u \leq 100, \underline{\text{alen}}(@^\sharp) = n, 0 \leq n \leq 2021, i \geq 0\Big)$$

The summarization domain returns $\underline{\text{arr}}(@^\sharp)_{\texttt{weak}}$. The length domain returns $\top$, in the state where $0 \leq i < \underline{\text{alen}}(@^\sharp)$ holds. These results are combined, and the reduced product returns $\underline{\text{arr}}(@^\sharp)_{\texttt{weak}}$, in the state below (the new part is written in bold font).

$$\Big(-100 \leq l \leq \underline{\text{arr}}(@^\sharp) \leq u \leq 100, \underline{\text{alen}}(@^\sharp) = n, 0 \leq n \leq 2021, \mathbf{0 \leq i < \underline{\text{alen}}(@^\sharp)}\Big)$$

Our semantics defined a simplified case where we keep only non-erroneous states (cf. Remark 2.6). In the actual analysis of Python, we would have created an erroneous case raising an exception. Its state would be the input state, filtered by the condition $i > \underline{\text{alen}}(@^\sharp)$.

Remark 5.17 **Current implementation**

Currently, the implementation of this domain for Python analyses combines both the length and smashing abstractions in a single domain. If we split the abstractions into the reduced product shown here, we could disable the length domain in the case of the type analysis presented in Chapter 7.

Remark 5.18 **Array expansion abstraction**

Another simple array abstraction is array expansion, which is much more precise but less scalable than the array summarization. It consists in considering each element of the array as a variable. We could for example introduce the auxiliary variable $\underline{\text{exparr}}(@^\sharp \in \textsf{Addr}^\sharp, i \in \mathbb{N})$ denoting the i-th element of the array allocated at $@^\sharp$.

5.1.5 Variation: abstracting sets

Python also has a builtin set data structure, where each element is present at most once. The same transfer functions apply to the summarization domain. The length domain needs to be adapted: it cannot know if the element is already in the set, so the definition of $append$ adds non-deterministically 1[1].

[1] If the set is empty, $append$ can add 1.

Example 5.19 **Set analysis**

In the program of Listing 5.4, we can infer that:

- the length of the set is between 1 and 2 (depending on whether $i = j$ or not),
- the elements of s are between 0 and 10.

Listing 5.4: Program using sets

```
1 i = rand(0, 10)
2 j = rand(0, 10)
3 s = set(i, j)
```

5.2 Dictionaries

Dictionaries are associative data structures. We consider the case where keys and values can have heterogeneous values since this is permitted by Python. For the sake of simplicity, we do not rule out dictionaries having multiple bindings with the same key in our concrete semantics, since they do not change the summarization abstraction.

5.2.1 Dictionary operations

We consider the following dictionary operations.

▷ **Creation,** written $\{ k_1 : v_1, \ldots, k_n : v_n \}$ where k_i and v_i are expressions.
▷ **Adding a binding,** $d.setitem(k, v)$ adds the binding from key k to value v to dictionary d.
▷ **Testing key membership,** $d.contains(k)$ returns 1 if and only if k is a key of d (and 0 otherwise).
▷ **Searching for a binding,** $d.getitem(k)$ returns the value of the dictionary at key k.
▷ **Listing all bindings,** $d.keys()$ returns an array containing the dictionary keys.

The corresponding names of functions in Python are prefixed and suffixed by a double underscore, except for $keys$.

Definition 5.20 **Program State**

The program state of Definition 5.1 is extended with a new object $Dict$ consisting of tuples of keys and values.

$$Dict(d \in ((\mathsf{Value} \cup \mathsf{Addr})^2)^*) \in \mathsf{Obj}$$

The concrete semantics is shown in Figure 5.4. The creation of a dictionary consists in allocating an address, evaluating all elements and storing them in the heap. $setitem$ evaluates the dictionary and the binding before adding it to the heap. $contains$ returns a disjunction, depending on whether a key is found. $getitem$ searches for the value of the key passed in argument. $keys$ allocates all the keys into an array.

Example 5.21 **Occurence number**

An example program counting occurences using dictionaries is shown in Listing 5.5. This program iterates over s. For each element x of s, if x is already bound in the dictionary, we add 1 to its value. Otherwise, we add a binding for key x, with value 1.

We have not specified what s is here. If s is a string such as "analysis", the resulting dictionary is `{'a': 2, 'n': 1, 'l': 1, 'y': 1, 's': 2, 'i': 1}`. s can also be an array, such as `[2, 1, 1, 1, 2, 1]`. In that case the result is `{2 : 2, 1 : 4}`.

$$
\begin{array}{l}
\mathbb{E}[\![\, \{ k_1 : v_1, \ldots, k_n : v_n \} \,]\!](e,h) = \\
\quad \text{let } @, (e,h) = \mathbb{E}[\![\, \mathit{alloc\_addr}() \,]\!](e,h) \text{ in} \\
\quad \text{let } k_1, (e,h) = \mathbb{E}[\![\, k_1 \,]\!](e,h) \text{ in} \\
\quad \text{let } v_1, (e,h) = \mathbb{E}[\![\, v_1 \,]\!](e,h) \text{ in} \\
\quad \ldots \\
\quad \text{let } k_n, (e,h) = \mathbb{E}[\![\, k_n \,]\!](e,h) \text{ in} \\
\quad \text{let } v_n, (e,h) = \mathbb{E}[\![\, v_n \,]\!](e,h) \text{ in} \\
\quad @, (e, h[@ \mapsto \mathit{Dict}((k_1,v_1),\ldots,(k_n,v_n))]) \\
\mathbb{S}[\![\, d.\mathit{setitem}(k,v) \,]\!](e,h) = \\
\quad \text{let } @, (e,h) = \mathbb{E}[\![\, a \,]\!](e,h) \text{ in} \\
\quad \text{let } \mathit{Dict}(kv_1,\ldots,kv_n) = h(@) \text{ in} \\
\quad \text{let } k, (e,h) = \mathbb{E}[\![\, k \,]\!](e,h) \text{ in} \\
\quad \text{let } v, (e,h) = \mathbb{E}[\![\, v \,]\!](e,h) \text{ in} \\
\quad (e, h[@ \mapsto \mathit{Dict}(kv_1,\ldots,kv_n,(k,v))]
\end{array}
$$

$$
\begin{array}{l}
\mathbb{E}[\![\, d.\mathit{contains}(k) \,]\!](e,h) = \\
\quad \text{let } @, (e,h) = \mathbb{E}[\![\, a \,]\!](e,h) \text{ in} \\
\quad \text{let } \mathit{Dict}((k_1,v_1),\ldots,(k_n,v_n)) = h(@) \text{ in} \\
\quad \mathbb{E}[\![\, 1 \,]\!] \circ \mathbb{C}[\![\, k_1 == k \text{ or } \cdots \text{ or } k_n == k \,]\!] \cup \\
\quad \mathbb{E}[\![\, 0 \,]\!] \circ \mathbb{C}[\![\, k_1 != k \text{ and } \cdots \text{ and } k_n != k \,]\!] \\
\mathbb{E}[\![\, d.\mathit{getitem}(k) \,]\!](e,h) = \\
\quad \text{let } @, (e,h) = \mathbb{E}[\![\, a \,]\!](e,h) \text{ in} \\
\quad \text{let } \mathit{Dict}((k_1,v_1),\ldots,(k_n,v_n)) = h(@) \text{ in} \\
\quad \cup_{1 \le i \le n} \mathbb{E}[\![\, v_i \,]\!] \circ \mathbb{C}[\![\, k_i == k \,]\!](e,h) \\
\mathbb{E}[\![\, d.\mathit{keys}() \,]\!](e,h) = \\
\quad \text{let } @, (e,h) = \mathbb{E}[\![\, a \,]\!](e,h) \text{ in} \\
\quad \text{let } \mathit{Dict}((k_1,v_1),\ldots,(k_n,v_n)) = h(@) \text{ in} \\
\quad \mathbb{E}[\![\, [k_1,\ldots,k_n] \,]\!](e,h)
\end{array}
$$

Figure 5.4: Semantics of dictionaries

Listing 5.5: Computing the number of occurences of s in d

```
d = {}
for x in s:
  if d.contains(x): d.setitem(x, d.getitem(x)+1)
  else: d.setitem(x, 1)
```

Example 5.22 **Heterogeneous dictionaries**

Some dictionaries can have heterogeneous values (either in the keys or the values). Key heterogeneity rarely happens but is still possible. An example is shown in Listing 5.6, where some image metadata is stored. All keys are strings. The value of key "extension" is a string, while others are integers.

Listing 5.6: Example dictionary with heterogeneous values

```
image = {
  "extension": "jpg",
  "width": 1920,
  "height": 1080
}
```

Dictionary keys can be optional (i.e., inserted conditionally) or abstracted imprecisely (due to previous imprecisions, or an imprecise domain). These cases make the precise abstraction of dictionaries more difficult. If a domain keeps only an overapproximation of the set of keys, or is imprecise on some keys, it will be imprecise on the key membership and key access operations, which will create spurious false alarms.

The particular case where dictionary keys can only be strings is much simpler: the values of string keys are usually constant, and it is possible to keep an underapproximation of the set of keys that are defined in all the concrete environments. This case corresponds to the abstraction of object attributes in Python, used to handle Python's structural type system. A specific abstraction will be shown in Section 7.2.3.

$$\mathbb{E}^{\sharp}_{ds}[\![\, \{ k_1 : v_1, \ldots, k_n : v_n \} \,]\!]\sigma^{\sharp} =$$
$$\quad \text{let } @^{\sharp}, \sigma^{\sharp} = \mathbb{E}^{\sharp}[\![\, \mathit{alloc\_addr}() \,]\!]\sigma^{\sharp} \text{ in}$$
$$\quad @^{\sharp}, \mathbb{S}^{\sharp}[\![\, \underline{\text{vals}}(@^{\sharp})_{\texttt{weak}} = v_n \,]\!] \circ \mathbb{S}^{\sharp}[\![\, \underline{\text{keys}}(@^{\sharp})_{\texttt{weak}} = k_n \,]\!] \circ \cdots \circ \mathbb{S}^{\sharp}[\![\, \underline{\text{vals}}(@^{\sharp})_{\texttt{weak}} = v_2 \,]\!]$$
$$\qquad \circ \mathbb{S}^{\sharp}[\![\, \underline{\text{keys}}(@^{\sharp})_{\texttt{weak}} = k_2 \,]\!] \circ \mathbb{S}^{\sharp}[\![\, \underline{\text{vals}}(@^{\sharp}) = v_1 \,]\!] \circ \mathbb{S}^{\sharp}[\![\, \underline{\text{keys}}(@^{\sharp}) = k_1 \,]\!]\sigma^{\sharp}$$
$$\mathbb{S}^{\sharp}_{ds}[\![\, d.\mathit{setitem}(k, v) \,]\!]\sigma^{\sharp} =$$
$$\quad \text{let } @^{\sharp}, \sigma^{\sharp} = \mathbb{E}^{\sharp}[\![\, d \,]\!]\sigma^{\sharp} \text{ in}$$
$$\quad \mathbb{S}^{\sharp}[\![\, \underline{\text{vals}}(@^{\sharp})_{\texttt{weak}} = v \,]\!] \circ \mathbb{S}^{\sharp}[\![\, \underline{\text{keys}}(@^{\sharp})_{\texttt{weak}} = k \,]\!]\sigma^{\sharp}$$
$$\mathbb{E}^{\sharp}_{ds}[\![\, d.\mathit{contains}(k) \,]\!]\sigma^{\sharp} =$$
$$\quad \text{let } @^{\sharp}, \sigma^{\sharp} = \mathbb{E}^{\sharp}[\![\, d \,]\!]\sigma^{\sharp} \text{ in}$$
$$\quad \mathbb{E}^{\sharp}[\![\, 1 \,]\!] \circ \mathbb{C}^{\sharp}[\![\, k == \underline{\text{keys}}(@^{\sharp})_{\texttt{weak}} \,]\!]\sigma^{\sharp} \sqcup^{\sharp} \mathbb{E}^{\sharp}[\![\, 0 \,]\!] \circ \mathbb{C}^{\sharp}[\![\, k! = \underline{\text{keys}}(@^{\sharp})_{\texttt{weak}} \,]\!]\sigma^{\sharp}$$
$$\mathbb{E}^{\sharp}_{ds}[\![\, d.\mathit{getitem}(k) \,]\!]\sigma^{\sharp}$$
$$\quad \text{let } @^{\sharp}, \sigma^{\sharp} = \mathbb{E}^{\sharp}[\![\, d \,]\!]\sigma^{\sharp} \text{ in}$$
$$\quad \text{let } \mathit{tmp} = \mathit{mk\_tmp\_var}() \text{ in}$$
$$\quad \mathbb{E}^{\sharp}[\![\, \underline{\text{vals}}(@^{\sharp})_{\texttt{weak}} \,]\!] \circ \mathbb{C}^{\sharp}[\![\, \underline{\text{keys}}(@^{\sharp})_{\texttt{weak}} == k \,]\!]\sigma^{\sharp}$$
$$\mathbb{E}^{\sharp}_{ds}[\![\, d.\mathit{keys}() \,]\!]\sigma^{\sharp} =$$
$$\quad \text{let } @^{\sharp}, \sigma^{\sharp} = \mathbb{E}^{\sharp}[\![\, d \,]\!]\sigma^{\sharp} \text{ in}$$
$$\quad \mathbb{E}^{\sharp}[\![\, [\underline{\text{keys}}(@^{\sharp})_{\texttt{weak}} \texttt{ for x in range}(\mathit{rand}(0, +\infty))] \,]\!]\sigma^{\sharp}$$
$$\mathbb{S}^{\sharp}_{ds}[\![\, \texttt{rename}(@^{\sharp}_1, @^{\sharp}_2) \,]\!] = \mathbb{S}^{\sharp}[\![\, \texttt{rename}(\underline{\text{vals}}(@^{\sharp}_1), \underline{\text{vals}}(@^{\sharp}_2)) \,]\!] \circ \mathbb{S}^{\sharp}[\![\, \texttt{rename}(\underline{\text{keys}}(@^{\sharp}_1), \underline{\text{keys}}(@^{\sharp}_2)) \,]\!]$$

Figure 5.5: Dictionary smashing domain

5.2.2 Whole smashing

A coarse abstraction of dictionaries is to separate keys and values in different arrays, and apply the array summarization abstraction on each of those arrays. We summarize all keys in one auxiliary variable (written $\underline{\text{keys}}(@^{\sharp})$) and all values in another auxiliary variable (written $\underline{\text{vals}}(@^{\sharp})$). This domain is stateless, and its transfer functions are shown in Figure 5.5. The transfer functions are similar to the array summarization domain shown in Section 5.1.3, except that different auxiliary variables are used for keys and values. The transfer function of keys returns a list of any size, containing the keys of the dictionary (we reuse notation of list comprehensions of Python).

Example 5.23 **Summarizing homogeneous dictionaries**

The smashing domain works well in the case of homogeneous dictionaries. Let us assume we have a type analysis. This analysis aims at detecting type-related errors by inferring typed points-to information for variables. Going back to the example of occurrences computation in Listing 5.5 and assuming s is a string, this type analysis could infer that the keys are strings and the values are integers.

Example 5.24 **Summarizing heterogeneous dictionaries**

In the case of heterogeneous dictionaries, this abstraction is quickly imprecise and may introduce false types that will trigger many false alarms in the rest of the analysis. In the

example of Listing 5.6, using an interval domain and a string powerset domain, accessing `image["extension"]` would give the string `"jpg"` or any number in the interval $[1080, 1920]$.

Definition 5.25 **Concretization**

The concretization is similar to the other summarization-based domains (Sections 2.4.5, 4.1.5.1 and 5.1.3).

$$\gamma_{ds}(\texttt{()}) = \{\, (R,\rho) \mid v \in \mathrm{dom}R \setminus \{\, \underline{\mathrm{keys}}(\_), \underline{\mathrm{vals}}(\_) \,\} \Leftrightarrow |R(v)| = 1 \wedge \rho(v) = R(v);$$
$$(\underline{\mathrm{keys}}(@^\sharp) \in \mathrm{dom}R \wedge \underline{\mathrm{vals}}(@^\sharp) \in \mathrm{dom}R) \Leftrightarrow \exists n \in \mathbb{N},$$
$$\rho(@^\sharp) = Dict((k_1, v_1) \cdots (k_n, v_n)) \wedge \forall i \in [1,n], k_i \in R(\underline{\mathrm{keys}}(@^\sharp)) \wedge v_i \in R(\underline{\mathrm{vals}}(@^\sharp))\,\}$$

Remark 5.26 **More precise smashing for heterogeneous dictionaries**

An interesting future work is to have a smashing abstraction where each dictionary smashing is partitioned by the types of its values, to avoid such imprecision.

Remark 5.27 **Dictionary expansion**

Similarly to arrays (Remark 5.18), dictionaries can also be abstracted by expansion, i.e., different auxiliary variables are used to abstract each binding in the dictionary. However, the choice of auxiliary variables to continue our approach by delegation to underlying domains is more complex in this case:

- If we keep one auxiliary variable per key and per value, they could be numbered by insertion order, but this concrete numbering would create issues when analyzing loops. Moreover, it would be inefficient to perform dictionary operations since even membership testing would be costly.
- We could keep the set of keys of each dictionary as the state of the domain, and auxiliary variables for each value would consist in the dictionary's address and the corresponding key, which has been evaluated. The evaluation of the key, however is an issue. Let us assume we have a dictionary `d` where we perform `d.setitem(2x+1, 3)`, and the address of `d` is $@^\sharp$. Following Mopsa's evaluation mechanism, if keys are only evaluated in expressions, we could use a ghost variable $@^\sharp{\cdot}(2x+1)$. However, the expression's values change depending on the environment. We would have to perform a unique copy for the variables of the expressions in which the key evaluated, which would be costly. Otherwise, we could decide to store abstract values (for example, query the polyhedron domain for an interval approximation of $2x+1$). This would, however remove relationality.

These solutions are not satisfactory, and dictionary expansion would probably be too expensive in most analyses, so this is currently not implemented in Mopsa. An important future work of this thesis is to have better dictionary abstractions.

5.3 Related work

Array abstractions. Blanchet et al. [14] first mention the notion of array smashing and expansion. Gopan et al. [63] define summarization operators on numerical domains, including relational domains. Their approach applies in particular to the summarization of arrays using

one variable, corresponding to the auxiliary variable $\underline{\text{arr}}(\cdot)$ in our case. They introduce the notion of folding and expanding dimensions, which we introduced ∈Section 2.4.5. These operators have then been used to define the transfer function of the recency abstraction in Section 4.1, and of the summarization domains of this chapter. Siegel and Simon [135] define more precise expand/fold operations when multiple dimensions are involved. Gopan et al. [64] partition arrays dynamically into different segments (some being summarized and others not), that can depend on other numerical variables. A specific usecase is to prove that array bounds are preserved through a loop performing a copy. Contrary to the array copy example of Listing 5.2 where elements are iteratively added, their usecase focuses on copy from arrays uninitialized with some values, in a C-like setting. In that case, each array is split into two summaries (of subarrays strictly before and after the current index) and the single cell at the current index [64, Figure 2]. Halbwachs and Péron [68] extend the approach to express relational properties between the partitioned subarrays and the corresponding indexes (by contrast Gopan et al. [64] use summarized variables). In the case of array copy, they are able to prove that arrays are equal. Gopan et al. [64] use a heuristic focusing on array writes to define its segments, while Halbwachs and Péron [68] rely on a pre-analysis. Contrary to these approaches, Cousot et al. [37] perform a single-pass analysis, inferring semantic array segments and properties on the segmented arrays at the same time. Liu and Rival [97] define array analyses where contents can be summarized in non-contiguous groups, contrary to previous works which relied on contiguous segments. All these works assume arrays are of fixed size (although Liu and Rival [97] suggest how their work can be extended to dynamically allocated arrays). Dillig et al. [46] encode results of index accesses as constraints over a points-to graph. Constraints consist in a pair of an under and an overapproximation of the set of concrete elements reachable from a given array index access. These constraints are called “bracketing constraints” and generalize the dichotomy of weak and strong updates. In particular, they ensure that if the necessary and sufficient conditions are the same, the constraints model a strong update, and if nothing is known, a weak update is performed. The constraints are then solved using an SMT solver.

Other abstractions. Fulara [56, 55] abstracts dictionaries as sets of tuples of key and value abstractions. This set is partitioned according to the key’s values. Since this abstraction only performs an overapproximation of the keys, it cannot answer precisely membership tests. The work of Fulara [56, 55] is the only one instantiated with a non-numerical abstract domain (they use a regular expression abstract domain for dictionary keys). Cox et al. [38] define a set abstraction able to reason about relations between sets (e.g., set inclusion), as well as the sets’ contents.

Dillig et al. [47] encode accesses on containers (including arrays, maps, and sets) as constraints which are solved using an SMT solver. In the case of dictionaries, keys are converted to integers in the constraints encoding. Monniaux and Alberti [116] abstract programs using arrays (or maps) to scalar programs, which can then be fed to analyzers. This translation could be performed on the fly in Mopsa.

Shape analyses [25] focus on precisely analyzing specific data structures such as linked lists or arrays.

5.4 Conclusion

This finishes the part defining the base abstractions used in the rest of the analyses. This chapter provided modular definitions of coarse, summarization-based array and dictionary abstractions. They handle the container’s dynamic size and potential heterogeneity. In particular, the design of more precise dictionary abstractions in the setting of dynamic programming languages is an interesting future work.

Part III

Pure Python Programs

6

Concrete Semantics of Python

This chapter defines a collecting semantics for Python. The description intends to serve as a reference upon which the abstract semantics will be based.

The essence of Python. Python is appreciated for its powerful and permissive high-level syntax, e.g., it allows programmers to redefine all operators (addition, attribute access, etc.) in custom classes, and comes equipped with a vast standard library. Python is an object-oriented, dynamic programming language. Particular features of interest are:

▷ **Object orientation.** Everything is object in Python. The type of an object is the class from which it has been instantiated. For example (Figure 6.1), the type of the object `42` is `int`, and the type of `int` is the class `type`. An object is an instance of a class if the former type is a subclass of the latter. All objects are instances of the `object` class. As such, `B()` is an instance of `B`, but also of `A` and of `object`. Python classes are first-class objects and can be stored into variables.

▷ **Type system.** Python's type system is dynamic: variables are neither statically declared nor typed, and no type-checking is performed before running the program. In addition, the type of the value held by a variable may change during the program execution (e.g., if `x` holds a string at the beginning of the program, nothing prevents it from holding an integer later on). Although dynamic, Python is strongly typed. This means that operations that are not well defined (e.g, `1 + 'a'`) will raise an exception (such as a `TypeError`) rather than try an automatic coercion. Python relies on both a nominal and a structural type system. The nominal one, uses classes from which an object is instantiated, and the structural one, informally called duck typing, which is based on attributes.

▷ **Introspection.** Programs can inspect the type of variables at run-time to alter their execution. Operators exist to support both nominal and structural types. `isinstance(o, cls)` checks whether `o` has been instantiated from class `cls` or a class inheriting from `cls`. For example, `isinstance("a", str)` returns `True`, while `isinstance(object(), str)` returns `False`. `hasattr(o, attr)` checks whether `o` (or its class, or one of its parent class) has an attribute with name equal to the `attr` string. Methods defined by classes are attributes mapping to functions. For instance, `hasattr(42, "__add__")` returns `True` because `int`, the class of `42`, has an addition method, named `"__add__"`.

▷ **Self-modification.** Programs can dynamically alter the structure of Python objects. It is possible to add attributes to any object (including classes) at run-time or remove them. Dynamic class creation is also possible.

▷ **Code evaluation.** It is possible to evaluate arbitrary string expressions as Python code at run-time with the `eval` statement.

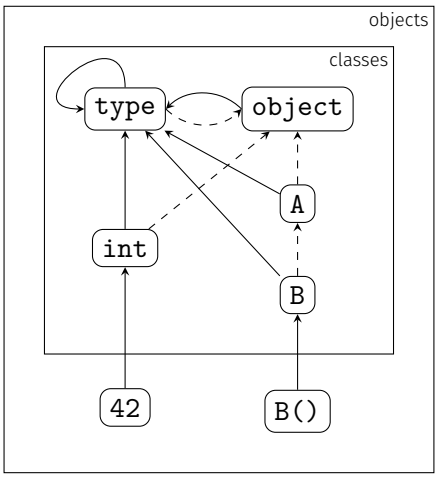

Figure 6.1: Python type and subclass relations
Assuming A is a class and B inherits from A.

Establishing the semantics. There is no standard defining the Python language, contrary to JavaScript for example. The reference implementation is called CPython. We focus on CPython version 3.8, which was first released in October 2019. CPython version 3.9 has been released since, without major changes in the semantics. Version 3.10 is expected to be released in October 2021, and includes pattern-matching operators. Python Enhancement Proposals (PEPs) are textual descriptions aiming at improving or changing the Python language. We have thus established the semantics by reading the Python documentation, the CPython source code, and by experimenting with CPython on small, hand-crafted programs.

Python relies on an abstract machine to execute its code. We used the `dis` module to find how CPython compiles programs to the abstract machine's bytecode. We can then find the behavior of each bytecode instruction in `Python/ceval.c` For more complex cases, we made custom programs emitting SIGTRAP signals[1] and used `gdb` to understand the backtrace.

A few blogs also cover specific details of Python's semantics [140, 118].

A concurrent implementation of CPython is Pypy, which is more performance-oriented. Despite the existence of alternative Python interpreters, CPython's source code acts as the reference defining the semantics of Python. In some specific cases, we compare CPython's results with Pypy's one to decide if some behaviors are part of the semantics or implementation details. For example, we rule that the builtin value caching is CPython-specific and thus not part of our semantics (cf. Remark 6.11).

Remark 6.1 **Crossreferences with CPython's source code**

In order to improve the traceability of the semantics, we display along with each case of the semantics (starting from Figure 6.6) which bytecode instructions (in uppercase letters) and functions are used (in gray, clickable links to the source on GitHub for Python 3.8).

[1] once the `os` and `signal` modules are imported, this can be done using `os.kill(os.getpid(), signal.SIGTRAP)`

Semantic level. We define our semantics by structural induction on the expressions (and statements) as a function manipulating reachable program states. This semantics allows the definition to be close to the implementation of our abstract interpreter.

We could have defined the semantics at the bytecode level of CPython's abstract machine. The first reason we do not analyze the bytecode is to keep the benefits of the source code: lower-level bytecode may reduce the precision of static analyses. In addition, it is easier to report errors back to the user when the source code is analyzed. The second is that the bytecode of CPython is an implementation detail, according to the documentation.[2]

Limitations. This semantics does not describe: the import statement (handled by our analyzer but not presented here), asynchronous operators, the `eval` & `exec` statements and the garbage collector. In the latter case, this means that we ignore custom finalizers defined through the `__del__` method. Those are seldom defined, and the documentation explicitly states that exceptions occurring during the execution of finalizers are ignored[3]. Since our end-goal is to define analyses detecting uncaught exceptions, ignoring finalizers is thus not a major issue, as long as they do not have side effects on global variables. The parser handles specific scope keywords (`nonlocal, global`), which we do not describe here. Our semantics omits the detailed explanation messages accompanying the raised exceptions.

Outline. We start by defining the state on which the collecting semantics operates in Section 6.1. Then, we define the semantics of Python operators (Section 6.2). This is complemented with the behavior of builtin classes and functions (Section 6.3). Indeed, most operators delegate their work to methods of the involved objects, which for some are rarely overloaded by user classes. We nevertheless support the overloading of these methods through user-provided classes. For example, an attribute access `x.attr` will start by calling $type(x)$.`__getattribute__`$(\texttt{x}, \texttt{attr})$, which is usually resolved into `object.__getattribute__`$(\texttt{x}, \texttt{attr})$. The semantics is not mechanized for now. It is a significant extension on previous work by Fromherz et al. [54]. We discuss its correctness in Section 6.4 and related work in Section 6.6.

6.1 Concrete state

The concrete state is defined in Figure 6.2. We assume there is an infinite set of heap addresses **Addr**. Python objects are split into a nominal part (for the type and a potential builtin value) and a structural part (for the attributes). The nominal part **ObjN** can be:

- an integer (integers are unbounded in Python),
- a boolean,
- a floating-point number,
- a string,
- the `None` or `NotImplemented` constants,
- a list of objects referenced by their addresses,
- a tuple of objects referenced by their addresses,
- a function, defined through its name, arguments, local variables and body,
- a method of a function f bound to an instance (referenced by their addresses),
- a class, defined by its name, parents, metaclass, and declarations. In order to handle multiple inheritance, the parents are linearized to provide a coherent, total order, following the C3 linearization of Barrett et al. [6]. This order is called MRO (for "method resolution order"). In the example of Figure 6.1, the MRO of `B` is `(B, A, object)`.
- an instance (defined by the address of the class from which it is instantiated).

[2] `https://docs.python.org/3.8/library/dis.html`

[3] `https://docs.python.org/3.8/reference/datamodel.html#object.__del__`

The structural part **ObjS** is a finite map between attribute names and their contents' addresses. Builtin classes – such as integers – have a read-only structure denoted by 🔒. Attributes of builtin classes are defined in **ObjN**. A state consists of an environment and a heap. The environment $\mathcal{E}$ maps variable identifiers **Id** to addresses. Global variables that have not been currently defined are not bounded in the environment. However, some variables can be declared locally without an initial value. In that case, we use `LocalUndef` to represent this undefined value of a local variable. `LocalUndef` is not a Python object but an artificial construction of our semantics to handle that case. The heap $\mathcal{H}$ maps address to objects. Section 2.4.3 introduced flow tokens to handle the `break` statement of our toy language. We reuse this notion here and tag our state $\mathcal{S}$ with a flow token $f \in \mathcal{F}$ to handle non-local control-flow: *cur* represents the current flow on which all instructions that do not disrupt the control flow operate (e.g., assignments, but not `raise` or `return`). *brk* and *cont* represents early loop exits due to the `break` and `continue` statement. *ret* is used to handle returned values in the interprocedural analysis. *exn* collects the states given by a raised exception. *exn* is indexed by the address of the exception object, so each exception will be kept separate.

$$
\begin{aligned}
&\textsf{ObjN} \stackrel{\text{def}}{=} \texttt{int}(i \in \mathbb{Z}) \cup \texttt{bool}(b \in \{\texttt{True}, \texttt{False}\}) \cup \texttt{float}(f \in \mathbb{F}_{64}) \cup \texttt{str}(s \in \mathit{string}) \cup \texttt{None} \\
&\quad \cup \texttt{NotImpl} \cup \textsf{List}(ls \in \textsf{Addr}^*) \cup \textsf{Tuple}(t \in \textsf{Addr}^*) \cup \textsf{Fun}(f \in \textsf{Id} \times \textsf{Id}^* \times id^* \times \mathit{stmt}) \\
&\quad \cup \textsf{Method}(a \in \textsf{Addr}, f \in \textsf{Addr}) \cup \textsf{Class}(c \in \textsf{Id} \times \textsf{Id}^* \times \textsf{Id} \times \mathit{stmt}) \cup \textsf{Inst}(a \in \textsf{Addr}) \\
&\textsf{ObjS} \stackrel{\text{def}}{=} 🔒 \cup (\mathit{string} \rightharpoonup \textsf{Addr}) \\
&\mathcal{E} \stackrel{\text{def}}{=} \textsf{Id} \rightharpoonup \textsf{Addr} \cup \texttt{LocalUndef} \\
&\mathcal{H} \stackrel{\text{def}}{=} \textsf{Addr} \rightharpoonup \textsf{ObjN} \times \textsf{ObjS} \\
&\mathcal{F} \stackrel{\text{def}}{=} \{\, \mathit{cur}, \mathit{ret}, \mathit{brk}, \mathit{cont}, \mathit{exn}\ a, a \in \textsf{Addr} \,\} \\
&\mathcal{S} \stackrel{\text{def}}{=} \mathcal{E} \times \mathcal{H} \times \mathcal{F}
\end{aligned}
$$

Figure 6.2: Domain of the semantics

Semantic operators. The semantic operators are defined in Figure 6.3. Executing a statement s through $\mathbb{S}[\![\, s \,]\!]$ transforms a set of states into another set of states. Evaluating an expression e in a set of states Σ with $\mathbb{E}[\![\, e \,]\!]\Sigma$ yields Python values in a new state, as expressions may have side effects. These values may be bottom if the evaluation fails. We also define the conditional operator of an expression $\mathbb{C}[\![\, e \,]\!]$ as a filter keeping only the states where e evaluates to `True`.

$$
\begin{aligned}
&\mathbb{S}[\![\, \mathit{stmt} \,]\!] : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S}) \\
&\mathbb{E}[\![\, \mathit{expr} \,]\!] : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S} \times \textsf{Addr}^{\perp}) \\
&\mathbb{C}[\![\, \mathit{expr} \,]\!] : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S}) \\
&\mathbb{C}[\![\, e \,]\!]\Sigma = \{\, (f, e, h) \mid (f, e, h, a) \in \mathbb{E}[\![\, e \,]\!]\Sigma \wedge h(a) = (\texttt{bool}(\texttt{True}), 🔒) \,\}
\end{aligned}
$$

Figure 6.3: Operators of the semantics

Remark 6.2 **Simplified definition of $\mathbb{S}[\![\, \cdot \,]\!]$ using $\mathbb{S}_{\textsf{cur}}[\![\, \cdot \,]\!]$**

When a statement handle exceptional control-flow, it will change input states associated

to the *cur* token into output states, and leave unchanged states with other tokens. In these cases, we define the semantics using the $\mathbb{S}_{\mathsf{cur}}[\![\cdot]\!]$ (respectively $\mathbb{E}_{\mathsf{cur}}[\![\cdot]\!]$) operator, which is lifted to the following definition, ignoring non-local execution flows:

$$\mathbb{S}[\![\, stmt \,]\!]\Sigma = \{\, (f,e,h) \mid (f,e,h) \in \Sigma \wedge f \neq cur \,\} \cup \{\, \mathbb{S}_{\mathsf{cur}}[\![\, stmt \,]\!](cur,e,h) \mid (cur,e,h) \in \Sigma \,\}$$
$$\mathbb{E}[\![\, expr \,]\!]\Sigma = \{\, (f,e,h), \bot \mid (f,e,h) \in \Sigma \wedge f \neq cur \,\} \cup \{\, \mathbb{E}_{\mathsf{cur}}[\![\, stmt \,]\!](cur,e,h) \mid (cur,e,h) \in \Sigma \,\}$$

Definition 6.3 **Monadic `letcases` operator**

In most cases, our semantics will be defined on a single state at a time, and lifted (Remark 6.2). However, any subevaluation or subexecution may return a set of states. We use the `letcases` operator to hide the disjunction over the set of states and continue defining the semantics on a single state at a time.

$$\text{letcases } (f,e,h), a = \mathbb{E}[\![\, e \,]\!]S \text{ in } body \stackrel{\text{def}}{=} \bigcup_{(f,e,h,a) \in \mathbb{E}[\![\, e \,]\!]S} body$$

Definition 6.4 **Monadic `letb` operator**

In order to easily lift the semantics, we may use the `letb` operator, which will execute `body` using the non-failing evaluations (these cases correspond to states tagged with the *cur* control-flow token[4]).

$$\text{letb } (f,e,h), a = \mathbb{E}[\![\, e \,]\!]S \text{ in } body \stackrel{\text{def}}{=} \bigcup_{\substack{(f,e,h,a) \in \mathbb{E}[\![\, e \,]\!]S \\ a \neq \bot}} body \cup \{(f,e,h,\bot) \in \mathbb{E}[\![\, e \,]\!]S\}$$

Auxiliary functions. In order to ease the definition of the semantics, we define two builtin functions, and introduce five artificial expressions, written in gray.

Definition 6.5 **Non-current flow predicate,** *isNotCur*

The predicate *isNotCur* holds when the flow token of a state $\sigma \in \mathcal{S}$ is not *cur*, i.e., $f \neq cur$ where $\sigma = (f,e,h)$.

Definition 6.6 **Not implemented predicate** *isNotImplemented*

The predicate *isNotImplemented* holds when the address passed in argument refers to the `NotImplemented` object.

Definition 6.7 **Address allocation expression** *alloc_addr*

alloc_addr returns a fresh address.

Definition 6.8 **Low-level field operators** *get_field*, *has_field*, *set_field*, *del_field*

We distinguish the notions of Python attributes and (low-level) fields. Fields are local to an object. Python attributes are resolved transitively from an instance: if the field is not found

[4] Except for the specific case of generators.

in an instance, it will be looked up in its parent class, and so forth.

While Python offers attribute resolution operators, these low-level operators working at the field level are not exposed in Python. They are however used in the C implementation of the interpreter. To address this discrepancy, we introduce the *get_field*, *has_field*, *set_field* and *del_field* operators, defined in Figure 6.4. *get_field* first searches in the nominal part of the object for builtins (using get_builtin_field) and then in the structural part. *has_field* returns `True` if *get_field* succeeded. *set_field* searches for the attribute in the heap of the state. If we try to set a field of a builtin (such as the integer class), the structural part is locked and *set_field* will fail. Otherwise, it updates the structural part. *del_field* is close to *set_field*; it removes the given field.

Remark 6.9 **Evaluated arguments for low-level field operators**

We have assumed in the semantics of the low-level field operators that the arguments were already evaluated, since only our semantics can call these operators (thus ensuring that the arguments were evaluated before). We cannot make this assumption for Python expressions, since these expressions can be an original part of a program.

Definition 6.10 **Attribute resolution function** *mro_search*

The *mro_search* function (defined in Figure 6.4) searches for a field in the MRO of a class. The MRO of a class consists in itself and all the class it inherits from, in a linearized version. The *mro_search* function returns the object found at the given field, or $\bot$ if nothing is found.

$$
\begin{array}{l}
\mathbb{E}_{\text{cur}}[\![\, \mathit{get\_field}(@_{obj} \in \textsf{Addr}, attr \in \mathit{string}) \,]\!](\mathit{cur}, e, h) \stackrel{\text{def}}{=} \\
\quad \text{let } b = \text{get\_builtin\_field}(@_{obj}, attr) \text{ in} \\
\quad \text{if } b \neq \bot \text{ then return } (f, e, h), b \text{ else} \\
\quad \text{return } (f, e, h), (\text{snd} \circ h(@_{obj}))(attr) \\
\\
\mathbb{E}_{\text{cur}}[\![\, \mathit{has\_field}(@_{obj} \in \textsf{Addr}, attr \in \mathit{string}) \,]\!](f, e, h) \stackrel{\text{def}}{=} \\
\quad \text{letb } (f, e, h), @_r = \mathbb{E}[\![\, \mathit{get\_field}(@_{obj}, attr) \,]\!](\mathit{cur}, e, h) \text{ in} \\
\quad \text{if } @_r \neq \bot \text{ then return } \mathbb{E}[\![\, \texttt{True} \,]\!](f, e, h) \text{ else return } \mathbb{E}[\![\, \texttt{False} \,]\!](f, e, h) \\
\\
\mathbb{S}_{\text{cur}}[\![\, \mathit{set\_field}(@_{obj} \in \textsf{Addr}, attr \in \mathit{string}, @_{val} \in \textsf{Addr}) \,]\!](\mathit{cur}, e, h) \stackrel{\text{def}}{=} \\
\quad \text{if } \text{snd} \circ h(@_{obj}) = \text{🔒} \text{ then return } \bot \text{ else} \\
\quad \text{return } (\mathit{cur}, e, h[@_{obj} \mapsto h(@_{obj})[attr \mapsto @_{val}]]) \\
\\
\mathbb{S}_{\text{cur}}[\![\, \mathit{del\_field}(@_{obj} \in \textsf{Addr}, attr \in \mathit{string}) \,]\!](\mathit{cur}, e, h) \stackrel{\text{def}}{=} \\
\quad \text{if } \text{snd} \circ h(@_{obj}) = \text{🔒} \text{ then return } \bot \text{ else} \\
\quad \text{return } (\mathit{cur}, e, h[@_{obj} \mapsto h(@_{obj}) \setminus attr]) \\
\\
\mathbb{E}_{\text{cur}}[\![\, \mathit{mro\_search}(cls \in \textsf{Class}, attr \in \mathit{string}) \,]\!](\mathit{cur}, e, h) \stackrel{\text{def}}{=} \\
\quad \text{for } c \text{ in } cls.\texttt{\_\_mro\_\_} \\
\quad\quad \text{if } \mathit{has\_field}(c, attr) \text{ then return } \mathit{get\_field}(c, attr) \\
\quad \text{return } (f, e, h), \bot
\end{array}
$$

Figure 6.4: Semantics of field operators

6.2 Core language

6.2.1 Literals

The evaluation of singleton constants such as `True`, `False`, `None`, `NotImplemented` returns the address of the corresponding object (in the current flow, and returns bottom otherwise). The initial state of the interpreter contains a heap where these builtins are already allocated. A new object containing the value is allocated for integer and string constants, and its address is returned.

Remark 6.11 **Builtin value caching**

Some builtins (such as integers and strings) are cached, meaning that a single instance at a specified address will exist for some specific values. For example, all integers between -5 and 256 are cached, and strings containing only ASCII letters are also cached. These are specific implementation details of CPython. In particular, Pypy does not have the same cache mechanism. Thus, we consider that this cache is not part of the semantics.

$$\mathbb{E}_{cur}[\![\,\texttt{True}\,]\!](cur, e, h) \stackrel{\text{def}}{=} \text{return } (cur, e, h), @_{\texttt{True}}$$

$$\mathbb{E}_{cur}[\![\,\texttt{False}\,]\!](cur, e, h) \stackrel{\text{def}}{=} \text{return } (cur, e, h), @_{\texttt{False}}$$

$$\mathbb{E}_{cur}[\![\,\texttt{None}\,]\!](cur, e, h) \stackrel{\text{def}}{=} \text{return } (cur, e, h), @_{\texttt{None}}$$

$$\mathbb{E}_{cur}[\![\,\texttt{NotImplemented}\,]\!](cur, e, h) \stackrel{\text{def}}{=} \text{return } (cur, e, h), @_{\texttt{NotImplemented}}$$

$$\mathbb{E}_{cur}[\![\,i \in \mathbb{Z}\,]\!](cur, e, h) \stackrel{\text{def}}{=}$$
$$\quad \text{letb } (f, e, h), @_a = \mathbb{E}[\![\,alloc\_addr\,]\!](cur, e, h) \text{ in}$$
$$\quad \text{return } (f, e, h[a \mapsto \texttt{int}(i), \emptyset]), @_a$$

$$\mathbb{E}_{cur}[\![\,s \in string\,]\!](cur, e, h) \stackrel{\text{def}}{=}$$
$$\quad \text{letb } (f, e, h), @_a = \mathbb{E}[\![\,alloc\_addr\,]\!](cur, e, h) \text{ in}$$
$$\quad \text{return } (f, e, h[a \mapsto \texttt{str}(z), \emptyset]), @_a$$

Figure 6.5: Semantics of literals

6.2.2 Variables

The semantics related to variable identifiers is shown in Figure 6.6. The CPython abstract machine has different operation codes depending on the variable's scope, so all bytecode instructions are shown in the traceability part of the semantics (Remark 6.1). If the flow token denotes the current evaluation, the evaluation of a variable identifier searches for this identifier in the environment e. Otherwise, the state is returned unchanged, according to Remark 6.2. Due to Python's scoping rules, a variable may be globally or locally undefined. The first case raises a `NameError` and the second an `UnboundLocalError`.

The semantics of assignments consists in evaluating the expression and updating the environment in the normal execution. A variable binding can also be removed at any time. Provided that the variable was correctly defined before, `del` id removes it from the environment.

$\mathbb{E}_{cur}[\![\texttt{id}]\!](cur, e, h) \stackrel{\text{def}}{=}$ LOAD_NAME LOAD_GLOBAL LOAD_FAST

if $id \notin \text{dom}\ e$ then return $\mathbb{S}[\![\texttt{raise NameError}]\!](f, e, h), \bot$ else

if $e(id) = \texttt{LocalUndef}$ then return $\mathbb{S}[\![\texttt{raise UnboundLocalError}]\!](f, e, h), \bot$ else

return $(f, e, h), e(id)$

$\mathbb{S}_{cur}[\![\texttt{id = expr}]\!](cur, e, h) \stackrel{\text{def}}{=}$ STORE_NAME STORE_GLOBAL STORE_FAST

letb $(cur, e, h), @ = \mathbb{E}[\![\, expr\,]\!](cur, e, h)$ in

return $(cur, e[id \mapsto @], h)$

$\mathbb{S}_{cur}[\![\texttt{del id}]\!](cur, e, h) \stackrel{\text{def}}{=}$ DELETE_NAME DELETE_GLOBAL DELETE_FAST

if $id \notin \text{dom}\ e$ then return $\mathbb{S}[\![\texttt{raise NameError}]\!](cur, e, h), \bot$ else

if $e(id) = \texttt{LocalUndef}$ then return $\mathbb{S}[\![\texttt{raise UnboundLocalError}]\!](cur, e, h), \bot$ else

return $(cur, e \setminus \{id\}, h)$

Figure 6.6: Semantics of variable evaluation, assignment and deletion

$\mathbb{E}_{cur}[\![\, type(a)\,]\!](cur, e, h) \stackrel{\text{def}}{=}$ type_new

letb $(f, e, h), @_a = \mathbb{E}[\![\, a\,]\!](cur, e, h)$ in

let $@_t =$ match $\text{fst} \circ h(@_a)$ with

- $\textsf{Inst}(@_t) \Rightarrow @_t$
- $\textsf{Class}(name, meta, supers, dict) \Rightarrow meta$
- $\textsf{Fun}(\ldots) \Rightarrow @_{function}$
- $\textsf{Method}(\ldots) \Rightarrow @_{method}$
- $\textsf{List}(\ldots) \Rightarrow @_{list}$
- $\textsf{Tuple}(\ldots) \Rightarrow @_{tuple}$
- $\texttt{NotImpl} \Rightarrow @_{NotImplementedType}$
- $\texttt{None} \Rightarrow @_{NoneType}$
- $\texttt{str}(\_) \Rightarrow @_{str}$
- $\texttt{float}(\_) \Rightarrow @_{float}$
- $\texttt{bool}(\_) \Rightarrow @_{bool}$
- $\texttt{int}(\_) \Rightarrow @_{int}$ in

return $\sigma, @_t$

Figure 6.7: Semantics of nominal type operators

6.2.3 Nominal types

The semantics of introspection operators related to nominal typing are described in Figures 6.7 to 6.9.

The *type* builtin returns the class from which an object has been instantiated. If the object is an instance of non-builtin objects, it will be defined as $\textsf{Inst}(@_t)$, where $@_t$ is the address of the class from which it was instantiated. For a class, the metaclass' address is returned. In the case of other builtins, the returned address corresponds to that of the builtin class.

$\mathbb{E}_{cur}[\![\, isinstance(obj, cls) \,]\!](cur, e, h) \stackrel{\text{def}}{=}$ builtin_isinstance_impl PyObject_IsInstance

letb $(f, e, h), @_{obj} = \mathbb{E}[\![\, obj \,]\!](cur, e, h)$ in
letb $(f, e, h), @_{cls} = \mathbb{E}[\![\, cls \,]\!](cur, e, h)$ in
if $h(@_{cls}) = \mathsf{Tuple}(\_)$ then
  return $\mathbb{E}[\![\, isinstance(@_{obj}, @_{cls}[0])$ `or` ... `or` $isinstance(@_{obj}, @_{cls}[\texttt{len}(cls) - 1]) \,]\!](f, e, h)$
if $h(@_{cls}) \neq \mathsf{Class}(\_)$ then return $\mathbb{S}[\![$ `raise TypeError` $]\!](f, e, h), \bot$ else
else return $\mathbb{E}[\![\, type(@_{cls}).\texttt{\_\_instancecheck\_\_}(@_{cls}, @_{obj}) \,]\!](f, e, h)$

$\mathbb{E}_{cur}[\![$ `type.__instancecheck__`$(cls, obj) \,]\!](cur, e, h) \stackrel{\text{def}}{=}$
type_instancecheck recursive_isinstance

letb $(f, e, h), @_{cls} = \mathbb{E}[\![\, cls \,]\!](cur, e, h)$ in
letb $(f, e, h), @_{obj} = \mathbb{E}[\![\, obj \,]\!](cur, e, h)$ in
if $type(@_{obj}) \sqsubseteq @_{cls}$ then return $\mathbb{E}[\![$ `True` $]\!](cur, e, h)$ else return $\mathbb{E}[\![$ `False` $]\!](cur, e, h)$

Figure 6.8: Semantics of nominal type operators

$\mathbb{E}_{cur}[\![\, issubclass(derived, cls) \,]\!](cur, e, h) \stackrel{\text{def}}{=}$ builtin_issubclass_impl PyObject_IsSubclass

letb $(f, e, h), @_{derived} = \mathbb{E}[\![\, derived \,]\!](cur, e, h)$ in
letb $(f, e, h), @_{cls} = \mathbb{E}[\![\, cls \,]\!](cur, e, h)$ in
if $h(@_{cls}) = \mathsf{Tuple}(\_)$ then
  return $\mathbb{E}[\![\, issubclass(@_{derived}, @_{cls}[0])$ `or` ...
    `or` $issubclass(@_{derived}, @_{cls}[\texttt{len}(cls) - 1]) \,]\!](cur, e, h)$
if $h(@_{cls}) \neq \mathsf{Class}(\_)$ then return $\mathbb{S}[\![$ `raise TypeError` $]\!](cur, e, h), \bot$ else
else return $\mathbb{E}[\![\, type(@_{cls}).\texttt{\_\_subclasscheck\_\_}(@_{derived}, @_{cls}) \,]\!](cur, e, h)$

$\mathbb{E}_{cur}[\![$ `type.__subclasscheck__`$(derived, cls) \,]\!](cur, e, h) \stackrel{\text{def}}{=}$
type_subclasscheck recursive_issubclass

letb $(f, e, h), @_{derived} = \mathbb{E}[\![\, derived \,]\!](cur, e, h)$ in
letb $(f, e, h), @_{cls} = \mathbb{E}[\![\, cls \,]\!](cur, e, h)$ in
if $derived \sqsubseteq cls$ then return $\mathbb{E}[\![$ `True` $]\!](cur, e, h)$ else return $\mathbb{E}[\![$ `False` $]\!](cur, e, h)$

Figure 6.9: Semantics of nominal type operators

Remark 6.12 **Address notation**

The semantics evaluates expressions into addresses. Addresses are not part of Python's expressions, but are used as drop-in replacements for evaluated parts. We prefix them with the @ symbol to avoid confusion with Python expressions.

The *isinstance* predicate is distributive over tuples. If the second argument is not a tuple, it should be a class (i.e., an instance of `type`). In that case, the `__instancecheck__` method of cls's class is called (i.e., $type(@_{cls})$.`__instancecheck__`). In general, this call is resolved by the `type` class itself (i.e., `type.__instancecheck__`). In that case, `type.__instancecheck__` returns `True` if and only if the type of the object is a subtype of the class (which corresponds to the natural definition of an object being an instance of the class).

Similarly to `isinstance`, an `issubclass` builtin is defined. Both `isinstance` and `issubclass` rely on the class subtype relation, defined as:

$$\textsf{Class}(c_1) \sqsubseteq \textsf{Class}(c_2) \stackrel{\text{def}}{=} c_2 \in mro(c_1)$$

$mro(c_1)$ is a tuple consisting of the MRO of c_1, i.e a linearized version of its parents classes. The MRO computation relies on the monotonic superclass linearization algorithm of Barrett et al. [6]. The corresponding `__mro__` attribute of a class returns its MRO. This is a read-only attribute that cannot be redefined at runtime. We also define the strict subtype relation:

$$\textsf{Class}(c_1) \sqsubset \textsf{Class}(c_2) \stackrel{\text{def}}{=} c_1 \neq c_2 \wedge \textsf{Class}(c_1) \sqsubseteq \textsf{Class}(c_2)$$

6.2.4 Structural types (attributes)

The concrete semantics related to attributes is shown in Figure 6.10.

To access a string attribute s (by the grammar of the language) of an expression x, we first start to evaluate x into $@_x$. We then search for the `__getattribute__` method in the MRO of $@_x$'s class. The search will always succeed as the object class has a `__getattribute__` method, and `object` is the last class in all MROs. In most cases the objects do not overload `object.__getattribute__`, which is defined in Section 6.3.1 (our semantics as well as our analyses support method overloading). Then, we call the method with $@_x$ and s as parameters. If the attribute is not found, an `AttributeError` exception has been raised. This exception is signaled using a flow token (the semantics of exceptions is shown in Section 6.2.8), on which we perform a pattern matching to filter the case where the `AttributeError` exception occurred. In that case, if $@_x$'s class has a `__getattr__` method, the exception is suppressed and the method called.

The *hasattr* introspection operator returns `True` if the evaluation of the attribute access is successful, and `False` otherwise if it failed with an `AttributeError` (otherwise, the raised exception is propagated).

Setting an attribute's value (resp. deleting an attribute) is done by calling the `__setattr__` method of the object's class (resp. `__delattr__`).

Remark 6.13 **Calls to introspection operators in the semantics**

Conditional filtering based on the test of a type (structural or nominal) of an object through *isinstance* and *hasattr* are frequent in the semantics. Instead of using a cumbersome explicit filtering such as:

$$\begin{aligned}&(\text{let } \sigma = \mathbb{C}[\![\, \mathit{isinstance}(attr, str) \,]\!](f, e, h) \text{ in } tt)\\ \cup &(\text{let } \sigma = \mathbb{C}[\![\, \neg \mathit{isinstance}(attr, str) \,]\!](f, e, h) \text{ in } ff)\end{aligned}$$

We use the following notation instead: if $\mathit{isinstance}(attr, str)$ then tt else ff. Similarly, calls to the *type* builtin are not always enclosed into semantic operators.

Remark 6.14 **Class vs instance-based attribute access**

Accesses to the attributes are done through the type of the object each time, using:

$$\mathit{type}(a).\texttt{\_\_setattr\_\_}$$

Accessing the attribute through the instance would be shorter (e.g., `a.__setattr__`) and yield the same result in most cases. However, the `__setattr__` field can be overloaded at the instance level. This will not be taken into account by CPython if we write directly

$\mathbb{E}_{cur}[\![\, x.s \,]\!](cur, e, h) \stackrel{\text{def}}{=}$ LOAD_ATTR PyObject_GetAttr (slot_tp_getattr_hook)

letb $(cur, e, h), @_x = \mathbb{E}[\![\, x \,]\!](cur, e, h)$ in
letb $(cur, e, h), @_c = \mathbb{E}[\![\, \mathit{mro\_search}(\mathit{type}(@_x), \texttt{"\_\_getattribute\_\_"}) \,]\!](cur, e, h)$ in
letcases $(f, e, h), @_{x.s} = \mathbb{E}[\![\, @_c(@_x, s) \,]\!](cur, e, h)$ in
match f with
- *exn* $@_{exc}$ when $\mathit{isinstance}(@_{exc}, \texttt{AttributeError}) \Rightarrow$
  let $(f, e, h), @_d = \mathbb{E}[\![\, \mathit{mro\_search}(\mathit{type}(@_x), \texttt{"\_\_getattr\_\_"}) \,]\!](f, e, h)$ in
  if $d \neq \bot$ then return $\mathbb{E}[\![\, @_d(@_x, s) \,]\!](cur, e, h)$
  else return $(f, e, h), \bot$
- _ $\Rightarrow$ return $(f, e, h), @_{x.s}$

$\mathbb{E}_{cur}[\![\, \mathit{hasattr}(obj, attr) \,]\!](cur, e, h) \stackrel{\text{def}}{=}$ builtin_hasattr_impl _PyObject_LookupAttr

letb $(cur, e, h), @_{obj} = \mathbb{E}[\![\, obj \,]\!](cur, e, h)$ in
letb $(cur, e, h), @_{attr} = \mathbb{E}[\![\, attr \,]\!](cur, e, h)$ in
if $\neg \mathit{isinstance}(@_{attr}, str)$ then return $\mathbb{S}[\![\, \texttt{raise TypeError} \,]\!](cur, e, h), \bot$ else
letcases $(f, e, h), @_r = \mathbb{E}[\![\, @_{obj}.@_{attr} \,]\!](cur, e, h)$ in
if $r = \bot$ then
  match f with
  - *exn* $@_{exc}$ when $\mathit{isinstance}(a, \texttt{AttributeError}) \Rightarrow$ return $\mathbb{E}[\![\, \texttt{False} \,]\!](cur, e, h)$
  - _ $\Rightarrow$ return $(f, e, h), \bot$
else return $\mathbb{E}[\![\, \texttt{True} \,]\!](f, e, h)$

$\mathbb{S}_{cur}[\![\, x.s = expr \,]\!](cur, e, h) \stackrel{\text{def}}{=}$ STORE_ATTR PyObject_SetAttr

letb $(cur, e, h), @_x = \mathbb{E}[\![\, x \,]\!](cur, e, h)$ in
if $\mathit{hasattr}(\mathit{type}(@_x), \texttt{"\_\_setattr\_\_"})$ then
  return $\mathbb{S}[\![\, \mathit{type}(@_x).\texttt{\_\_setattr\_\_}(@_x, s, expr) \,]\!](cur, e, h)$
return $\mathbb{S}[\![\, \texttt{raise TypeError} \,]\!](cur, e, h)$

$\mathbb{S}_{cur}[\![\, \texttt{del } x.s \,]\!](cur, e, h) \stackrel{\text{def}}{=}$ DELETE_ATTR PyObject_SetAttr

letb $(cur, e, h), @_x = \mathbb{E}[\![\, x \,]\!](cur, e, h)$ in
if $\mathit{hasattr}(\mathit{type}(@_x), \texttt{"\_\_delattr\_\_"})$ then
  return $\mathbb{S}[\![\, \mathit{type}(@_x).\texttt{\_\_delattr\_\_}(@_x, s) \,]\!](cur, e, h)$
return $\mathbb{S}[\![\, \texttt{raise TypeError} \,]\!](cur, e, h)$

Figure 6.10: Semantics of attribute access, assignment and deletion

`a.__setattr__` ; hence we will write $\mathit{type}(a)$.`__setattr__` explicitly and correctly handle (in our semantics and our analysis) the case where fields are overloaded.

6.2.5 Subscript

The concrete semantics of subscripts (used mainly to perform index accesses or slices of data structures) is defined in Figure 6.11.

To access e_1 at e_2, we start by evaluating e_1 and e_2. Then, we try to call e_1's `__getitem__` method if it exists. Otherwise, we raise a type error exception. A particular case exists for

classes (which are instances of the *type* object): we try to call the `__class_getitem__` method.

Subscript assignement and deletion are similar to the corresponding attribute operations: they delegate to the `__setitem__` (resp. `__delitem__`) methods.

$$
\begin{array}{l}
\mathbb{E}_{cur}[\![\, e_1[e_2] \,]\!](cur, e, h) \overset{\text{def}}{=} \qquad \texttt{BINARY\_SUBSCR PyObject\_GetItem slot\_mp\_subscript} \\
\quad \text{letb } (cur, e, h), @_1 = \mathbb{E}[\![\, e_1 \,]\!](cur, e, h) \text{ in} \\
\quad \text{letb } (cur, e, h), @_2 = \mathbb{E}[\![\, e_2 \,]\!](cur, e, h) \text{ in} \\
\quad \text{if } hasattr(type(@_1), \texttt{"\_\_getitem\_\_"}) \text{ then} \\
\qquad \text{return } \mathbb{E}[\![\, type(@_1).\texttt{\_\_getitem\_\_}(@_1, @_2) \,]\!](cur, e, h) \\
\quad \text{else if } isinstance(@_1, type) \wedge hasattr(@_1, \texttt{"\_\_class\_getitem\_\_"}) \text{ then} \\
\qquad \text{return } \mathbb{E}[\![\, @_1.\texttt{\_\_class\_getitem\_\_}(@_2) \,]\!](cur, e, h) \\
\quad \text{else return } \mathbb{S}[\![\, \texttt{raise TypeError} \,]\!](cur, e, h), \bot \\
\mathbb{S}_{cur}[\![\, e_1[e_2] = e_3 \,]\!](cur, e, h) \overset{\text{def}}{=} \qquad \texttt{STORE\_SUBSCR PyObject\_SetItem slot\_mp\_ass\_subscript} \\
\quad \text{letb } (cur, e, h), @_2 = \mathbb{E}[\![\, e_2 \,]\!](cur, e, h) \text{ in} \\
\quad \text{letb } (cur, e, h), @_1 = \mathbb{E}[\![\, e_1 \,]\!](cur, e, h) \text{ in} \\
\quad \text{letb } (cur, e, h), @_3 = \mathbb{E}[\![\, e_3 \,]\!](cur, e, h) \text{ in} \\
\quad \text{if } hasattr(type(@_1), \texttt{"\_\_setitem\_\_"}) \text{ then} \\
\qquad \text{return } \mathbb{E}[\![\, type(@_1).\texttt{\_\_setitem\_\_}(@_1, @_2, @_3) \,]\!](f, e, h) \\
\quad \text{return } \mathbb{S}[\![\, \texttt{raise TypeError} \,]\!](f, e, h) \\
\mathbb{S}_{cur}[\![\, \texttt{del } e_1[e_2] \,]\!](cur, e, h) \overset{\text{def}}{=} \qquad \texttt{DELETE\_SUBSCR PyObject\_DelItem slot\_mp\_ass\_subscript} \\
\quad \text{letb } (cur, e, h), @_1 = \mathbb{E}[\![\, e_1 \,]\!](cur, e, h) \text{ in} \\
\quad \text{letb } (cur, e, h), @_2 = \mathbb{E}[\![\, e_2 \,]\!](cur, e, h) \text{ in} \\
\quad \text{if } hasattr(type(@_1), \texttt{"\_\_delitem\_\_"}) \text{ then} \\
\qquad \text{return } \mathbb{E}[\![\, type(@_1).\texttt{\_\_delitem\_\_}(@_1, @_2) \,]\!](cur, e, h) \\
\quad \text{return } \mathbb{S}[\![\, \texttt{raise TypeError} \,]\!](cur, e, h)
\end{array}
$$

Figure 6.11: Semantics of subscript access, assignment and deletion

Remark 6.15 **Purpose of `__class_getitem__`**

Since Python 3.9, `__class_getitem__` is used to handle type hints on generic classes[5]. For example, this allows us to write and evaluate `list[int]`, where `list` is the builtin list class. Previously, the list class needed a different type annotation `List[int]`.

Remark 6.16 **Evaluation order**

According to Python's documentation, expressions are evaluated from left to right[6]. In the case of the subscript assignment, we follow a different evaluation order, provided in the implementation of the bytecode instruction `STORE_SUBSCR`.

It is also possible to unpack iterables through assignments. The semantics of such an unpacking is shown in Figure 6.12. We start by evaluating $expr$, and create an iterator over it. The

[5] `https://www.python.org/dev/peps/pep-0585/`

[6] `https://docs.python.org/3.8/reference/expressions.html#evaluation-order`

list comprehension consumes the n first elements of $expr$. If a `StopIteration` exception happens, e contained less than n elements: a `ValueError` is raised (replacing the `StopIteration` exception). If another exception happens, it is returned. If the list comprehension is successfully evaluated, we check that no more elements can be consumed from the iterator, since our unpacking requires that $expr$ contains exactly n elements. If the $expr$ contains more than n elements, a `ValueError` is raised. Otherwise, each variable is assigned to its corresponding value.

$\mathbb{S}_{cur}[\![(x_1, \ldots, x_n) = expr]\!](cur, e, h) \stackrel{\text{def}}{=}$ `UNPACK_SEQUENCE unpack_iterable`

letb $(cur, e, h), @_{expr} = \mathbb{E}[\![e]\!](cur, e, h)$ in
letb $(cur, e, h), @_{it} = \mathbb{E}[\![$ `iter(`$@_{expr}$`)` $]\!](cur, e, h)$ in
letcases $(f, e, h), @_r = \mathbb{E}[\![$ `[next(`$@_{it}$`) for x in range(n)]` $]\!]\sigma$ in
match f with
- *exn* $@_{exc} \Rightarrow$
  if *isinstance*$(@_{exc},$ `StopIteration`$)$ then return $\mathbb{S}[\![$ `raise ValueError` $]\!](cur, e, h)$ else
  else return (f, e, h)
- _ $\Rightarrow$
  letcases $(f, e, h), @_f = \mathbb{E}[\![$ `next(`$@_{it}$`)` $]\!]\sigma$ in
  match f with
  - *exn* $@_{exc} \Rightarrow$
    if $\neg$*isinstance*$(@_{exc},$ `StopIteration`$)$ then return (f, e, h)
    else return $\mathbb{S}[\![x_1 = @_r[0]]\!] \circ \ldots \circ \mathbb{S}[\![x_n = @_r[n-1]]\!](cur, e, h)$
  - _ $\Rightarrow$ return $\mathbb{S}[\![$ `raise ValueError` $]\!](f, e, h)$

Figure 6.12: Semantics of unpacking assignments

Remark 6.17 **Alternative unpacking**

An alternative unpacking, written $x_1, \ldots, x_{n-1}, *x_n = e$ only ensures that e has at least $n-1$ elements, and stores the remaining elements in x_n (or the empty list if no elements remain).

6.2.6 Conditionals

The semantics of conditionals (Figure 6.13) is standard: we first filter according to the condition, and then evaluate each branch. The condition is cast to booleans through an explicit call.

Remark 6.18 **Syntax of Python blocks**

Python's block statements (such as conditionals and loops) are usually defined over multiple lines due to the indentation-based block definition. For the sake of presentation, we revert to a more classical syntax with braces to define statements' blocks.

6.2.7 Loops

The `break` and `continue` statements have their usual semantics. When one of these statements is encountered, the states having a normal flow are interrupted by changing the flow token. This behavior allows injecting their states back in the semantics of the while loop.

$$\mathbb{S}[\![\,\texttt{if}\ (e)\ \{s_t\}\ \texttt{else}\ \{s_f\}\,]\!]S \stackrel{\text{def}}{=} \mathbb{S}[\![\,s_t\,]\!] \circ \mathbb{C}[\![\,\texttt{bool}(e)\,]\!]\Sigma \cup \mathbb{S}[\![\,s_f\,]\!] \circ \mathbb{C}[\![\,\texttt{not bool}(e)\,]\!]\Sigma$$

Figure 6.13: Semantics of conditionals

$$\mathbb{S}[\![\,\texttt{break}\,]\!]\Sigma \stackrel{\text{def}}{=} \{\,(f,e,h) \in \Sigma \mid f \neq \mathit{cur}\,\} \cup \{\,(\mathit{brk},e,h) \mid (\mathit{cur},e,h) \in \Sigma\,\}$$

$$\mathbb{S}[\![\,\texttt{continue}\,]\!]\Sigma \stackrel{\text{def}}{=} \{\,(f,e,h) \in \Sigma \mid f \neq \mathit{cur}\,\} \cup \{\,(\mathit{cont},e,h) \mid (\mathit{cur},e,h) \in \Sigma\,\}$$

$$\begin{array}{l}
\mathbb{S}[\![\,\texttt{while}\ (e)\ \{\mathit{body}\}\ \texttt{else}\ \{\mathit{belse}\}\,]\!]\Sigma \stackrel{\text{def}}{=} \\
\quad \text{let } \Sigma_0 = \{\,(f,e,h) \in \Sigma \mid f \notin \{\mathit{brk},\mathit{cont}\}\,\} \text{ in} \\
\quad \text{let } \Sigma_{lfp} = \text{lfp}\ F \text{ in} \\
\quad \{\,(f,e,h) \in \Sigma \mid f \in \{\mathit{brk},\mathit{cont}\}\,\} \cup \\
\quad \mathbb{C}[\![\,\texttt{not bool}(e)\,]\!]\{\,(\mathit{cur},e,h) \mid (\mathit{brk},e,h) \in \Sigma_{lfp}\,\} \cup \\
\quad \mathbb{S}[\![\,\mathit{belse}\,]\!] \circ \mathbb{C}[\![\,\texttt{not bool}(e)\,]\!]\{\,(\mathit{cur},e,h) \in \Sigma_{lfp}\,\} \cup \\
\quad \{\,(f,e,h) \in \Sigma_{lfp} \mid f \notin \{\mathit{brk},\mathit{cont},\mathit{cur}\}\,\}
\end{array}$$

$$\begin{array}{l}
F(\Sigma) \stackrel{\text{def}}{=} \\
\quad \text{let } \Sigma = \mathbb{S}[\![\,\mathit{body}\,]\!] \circ \mathbb{C}[\![\,\texttt{bool}(e)\,]\!]\Sigma \text{ in} \\
\quad \Sigma_0 \cup \{\,(f,e,h) \in \Sigma \mid f \neq \mathit{cont}\,\} \cup \{\,(\mathit{cur},e,h) \mid (\mathit{cont},e,h) \in \Sigma\,\}
\end{array}$$

Figure 6.14: Semantics of while loops

Python loops have an optional else clause, whose body will be executed upon exiting the loop normally (i.e., if no `break` has been encountered, and no exception has been raised either). In the case of a while loop, we first start by removing states labeled with a *brk* or *cont* flow token (which corresponds to the effect of a `break` or `continue` statement of an outer loop). These states will be returned at the end. Then, we compute the least fixpoint of f, which models the effect of the loop body. As such, f starts by applying the loop's guard (with a boolean cast) and executing the loop body afterward. If the execution of the body encountered a `continue` statement, the corresponding flow token is changed back to the normal execution token *cur*. The state returned by the transfer function of the while loop is the union of (i) the input states labeled with a `break` or `continue` statement (i.e., the states coming from outer loops), (ii) the fixpoint states that reached a `break` statement, filtered by the negation of the loop's guard, (iii) the fixpoint states in a normal control-flow state, filtered by the negation of the loop's guard, to which the body of the else clause is then applied, (iv) exceptional states (i.e., states of the fixpoint not tagged by *cur*, *cont* or *brk*)

Listing 6.1: Semantics of `for i in it: body`

```
fresh_tmp = iter(it)
while True:
  try: i = next(fresh_tmp)
  except StopIteration: break
  body
```

For loops can be written into while loops on an explicit iterator (Listing 6.1). The variable `fresh_tmp` denotes a fresh temporary variable which is deleted in all states after the rewriting (as such, it is not possible to write it `del fresh_tmp`, since the variable would only be deleted in the normal control-flow state).

Remark 6.19 **Else in for loops**

For loops can also have an else branch. In that case, the rewriting presented should be amended with a global variable to track if the break statement comes from the `StopIteration` or the body.

6.2.8 Exceptions

The raise statement (Figure 6.15) starts by evaluating the expression it is passed. If the result is a class deriving from `BaseException`, the class is instantiated. We then check that the created instance derives from the `BaseException` class since the class creation method `__new__` can return any object. If that is the case, the state is now labeled by an exception token, where a is the address of the exception instance. Otherwise, a type error is raised.

The exception catching mechanism is shown in Figure 6.15. We start by executing the body of the try clause. If no exception has been raised by `tbody`, `telse` is executed. Otherwise, let $@_{raised}$ be the address of the raised exception, and $@_{exn}$ the address of the exception class provided in the except statement. We check that $@_{exn}$ is indeed an exception class. If the raised exception is an instance of exn, we bind the variable v to the raised exception object and execute the body of the except statement. The body of the finally clause is always executed, and exceptions that have been uncaught in $tbody$ or raised during $texc$, $telse$ or $tfin$ are propagated.

$$
\begin{array}{l}
\mathbb{S}_{\mathsf{cur}}[\![\,\texttt{raise}\ exc\,]\!](cur,e,h) = \qquad \texttt{RAISE\_VARARGS do\_raise} \\
\quad \text{letb } (cur,e,h), @_{exc} = \mathbb{E}[\![\,exc\,]\!](cur,e,h) \text{ in} \\
\quad \text{letcases } (f,e,h), @ = \\
\qquad \text{if } \mathit{isinstance}(@_{exc}, \mathit{type}) \wedge \mathit{issubclass}(@_{exc}, \texttt{BaseException}) \text{ then } \mathbb{E}[\![\,@_{exc}()\,]\!]\sigma \\
\qquad \text{ else } \sigma, @_{exc} \text{ in} \\
\quad \text{if } \mathit{isinstance}(@, \texttt{BaseException}) \text{ then } (\text{if } f = cur \text{ then } \mathit{exn} \ @ \text{ else } f,e,h) \\
\quad \text{else } \mathbb{S}[\![\,\texttt{raise TypeError}\,]\!](f,e,h) \\
\mathbb{S}_{\mathsf{cur}}[\![\,\texttt{try } \{tbody\} \texttt{ except } (exn \texttt{ as } v) \ \{texc\} \texttt{else } \{telse\} \texttt{ finally } \{tfin\}\,]\!](cur,e,h) = \\
\quad \text{letcases } (f,e,h) = \mathbb{S}[\![\,tbody\,]\!](cur,e,h) \text{ in} \\
\quad \text{letcases } (f,e,h) = \\
\qquad \text{if } f \neq \mathit{exn}\ \_ \text{ then } \mathbb{S}[\![\,telse\,]\!](f,e,h) \text{ else} \\
\qquad \text{let } \mathit{exn}\ @_{raised} = f \text{ in} \\
\qquad \text{letb } (f,e,h), @_{exn} = \mathbb{E}[\![\,exn\,]\!](cur,e,h) \text{ in} \\
\qquad \text{if } \neg \mathit{issubclass}(@_{exn}, \texttt{BaseException}) \text{ then} \\
\qquad\quad \mathbb{S}[\![\,\texttt{raise TypeError}\,]\!](cur,e,h) \\
\qquad \text{else if } \mathit{isinstance}(@_{raised}, @_{exn}) \text{ then} \\
\qquad\quad \text{letb } (f,e,h) = \mathbb{S}[\![\,texc\,]\!](cur, e[\mathsf{v} \mapsto @_{raised}], h) \text{ in } (f, e \setminus \{\mathsf{v}\}, h) \\
\qquad \text{else } (f,e,h) \text{ in} \\
\quad \text{letcases } (f_{fin},e,h) = \mathbb{S}[\![\,tfin\,]\!](cur,e,h) \text{ in } (\text{if } f_{fin} \neq cur \text{ then } f_{fin} \text{ else } f,e,h)
\end{array}
$$

Figure 6.15: Semantics of exceptions

Remark 6.20 **Source code for try/except statement**

The translation of try/except statement into the CPython abstract machine is long. For this case, our semantics does not describe the relevant operators and functions in the source code.

Remark 6.21 **Interaction between exception effects and other control-flow effects**

The interaction between the exception-handling operators and other non-local control-flow operators such as `return` are described in the Python documentation[7]:

- If the finally clause executes a break, continue or return statement, exceptions are not re-raised.
- If the try statement reaches a break, continue or return statement, the finally clause will execute just prior to the break, continue or return statement's execution.
- If a finally clause includes a return statement, the returned value will be the one from the finally clause's return statement, not the value from the try clause's return statement.

Remark 6.22 **Except clauses variations**

There can be multiple except clauses. These are executed in order until a match is found. Binding the exception to a variable with the `as` keyword is optional.

An except clause can also match multiple exceptions described in a tuple, similarly to what *isinstance* and *issubclass* allow.

Remark 6.23 **Raise without arguments**

The raise statement can also be called without any arguments. It is used within an except clause to re-raise the caught exception. If the except clause is `except exn as v:`, `raise` will be rewritten into `raise v`. Outside an except clause, it will raise a `RuntimeError` since there is no exception to re-raise.

6.2.9 With context manager

The `with` context manager is a Python-specific feature. Given `with e as t: body`, the statement relies on the usage of two methods of `e` called `__enter__` and `__exit__`. As shown in Listing 6.2, this statement is a specialized version of exception-catching statements, where the `__exit__` method is guaranteed to be executed after $body$. This construction is handy to manage file operations, for example. The statement starts by evaluating e, and ensuring that its class has both the `__enter__` and `__exit__` methods. The `__enter__` method is then called, and its result is assigned to the variable t. The body of the statement is executed afterward. If an exception has been raised in the body, `__exit__` is called, and the last three arguments are exception information: the exception's type, the exception's value, and traceback information. Depending on the return value of `fresh_exit`, the exception is caught or raised again. If no exception has been raised, `fresh_exit` is still called. In the end, t is assigned `None`.

[7] `https://docs.python.org/3.9/tutorial/errors.html#defining-clean-up-actions`

Listing 6.2: Semantics of `with e as t: body`

```
fresh_mgr = e
fresh_enter = type(fresh_mgr).__enter__
fresh_exit = type(fresh_mgr).__exit__

t = fresh_enter(fresh_mgr)
try: body
except:
  if not fresh_exit(fresh_mgr, *sys.exc_info()): raise
else: fresh_exit(fresh_mgr, None, None, None)
finally: t = None
```

Example 6.24 **Usecase of the `with` statement**

We illustrate the use of this statement in Listing 6.3. We define a class of temporary files, similar to the `tempfile` module of the standard library. A `Tempfile` is initialized with the name of the temporary file we use. The `__enter__` method checks that the file does not exist and creates the file in write-only access. The `__exit__` method closes the file descriptor and deletes the file. Exceptions deriving from the class `io.UnsupportedOperation` will be absorbed.

In the first `with` statement, the file is opened (provided `foo` does not exist already). We write "Hello, world" in it. At the end of the statement, the file will be closed and removed.

In the second `with` statement, the file is opened (it was created before but removed at the end of the previous `with` statement). We try to read from the file, but this will raise an `io.UnsupportedOperation` exception. The write operation is thus skipped, and the `__exit__` method is called. Since it returns `True`, the exception is caught, and the execution continues.

In the third `with` statement, `t.write(42)` will raise a TypeError as the write method expects strings. The `__exit__` method is called to close and delete the file. However it returns `False`: the `TypeError` is raised again, interrupting the standard control-flow.

Listing 6.3: A simplified version of Python's `tempfile` library

```
import os, io

class Tempfile:
  def __init__(self, f):
    self.name = f

  def __enter__(self):
    if os.path.isfile(self.name):
      raise ValueError(f"file {self.name} exists already")
    self.file = open(self.name, 'w')
    return self.file

  def __exit__(self, exc, value, tb):
    self.file.close()
    os.remove(self.name)
    return exc is None or issubclass(exc, io.UnsupportedOperation)

with Tempfile("foo") as t:
  t.write("Hello, world\n")

with Tempfile("foo") as t:
  t.read()
  t.write("Anyone?")

with Tempfile("foo") as t:
```

```
26   t.write(42)
```

6.2.10 Function declaration

When a function is declared, a function object (with the function's name, arguments, local variables extracted using an auxiliary variable and body) is allocated on the heap. The environment is also updated to map the function's name to its allocated address.

$$
\begin{array}{l}
\mathbb{S}_{cur}[\![\texttt{def fname(args): body}]\!](cur, e, h) \stackrel{\text{def}}{=} \\
\quad \text{letb } (cur, e, h), @ = \mathbb{E}[\![\, alloc\_addr() \,]\!](cur, e, h) \text{ in} \\
\quad \text{let } f = \texttt{(fname, args, locals(body), body)} \text{ in} \\
\quad \text{return } (cur, e[\texttt{fname} \mapsto @], h[@ \mapsto \mathsf{Fun}(f)])
\end{array}
$$

Figure 6.16: Semantics of function declaration

We describe the different ways function arguments can be defined below. This part is only formalized for the base case and is accompanied by illustrating examples for the other cases.

Function arguments. The arguments of the function are defined through parameter names separated by commas. Arguments can have default values specified using the equal sign. For example, `def f(x, y, z=1): ...` indicates that `z` has default value `1`. `f` can thus be called with two to three parameters.

During call, the parameters' values can be passed in order (positional style) `f(3, 2)` meaning `x=3, y=2`, or in any order if the arguments are named `f(y=2, x=3)` (keyword style). It is also possible to mix calling styles: `f(1, z=2, y=3)` means that the arguments `(x, y, z)` are the tuple `(1, 3, 2)`. In both argument definitions and calls, positional arguments should be used before keyword arguments.

Since Python 3.8, it is possible to define operators that can be passed in positional-only or keyword-only styles using the `/` and `*` separators. Here is the specification proposed by PEP 570:

```
def name(positional_only_parameters, /,
         positional_or_keyword_parameters, *,
         keyword_only_parameters): ...
```

Variable-length arguments. One function argument can also be variable-length. It is then prefixed with the star operator `*`. For example with `def f(x, *y): ...`, `f(1,2,3,4)` means that `x=1` and `y=(2,3,4)`. Similarly, the notation `**` can be used to handle variable-length keyword arguments. With `def f(*args, **kwargs)` and `f(1,2,k=3,z=5)` means that `args` is a tuple for the first two arguments `args = (1, 2)` and `kwargs` a dictionary with the last two arguments `kwargs = {'k': 3, z: '5'}`.

Container destructuration. Lists, tuples, and dictionaries can be passed and destructured using the operators `*` and `**` during function calls. For example, given `def f(x, y, **kwargs)`, the dictionary `d = {'y': 1, 'x': 3, 'z':4}`, we can call `f(**d)` to get:

```
y = 1, x = 3, kwargs = {'z': 4}
```

6.2.11 Class declaration

A class declaration declares a name (cls), parents of the class (supers), a metaclass (meta, which can only be passed as a keyword argument), and a body. The metaclass is `type` by default; it ends up calling `type.__new__`, which is described later, in Figure 6.38. The body of the class is executed in order to define the attributes of the class. We assume we have an auxiliary $declarations$ function, which given a class' body, returns a Python dictionary where each attribute maps to its corresponding object. For example, in the case of the `Tempfile` class of Listing 6.3, $declarations$ would return a dictionary where `__init__` maps to the function object created by its declaration, and similarly for the `__enter__` and `__exit__` methods. The class is created by calling the metaclass.

$$
\begin{aligned}
&\mathbb{S}_{cur}[\![\texttt{class cls(supers, metaclass=meta): body}]\!](cur, e, h) \overset{\text{def}}{=} \\
&\quad \text{letb } \sigma = \mathbb{S}[\![\texttt{body}]\!](cur, e, h) \text{ in} \\
&\quad \text{let } \texttt{dict} = declarations(\texttt{body}) \text{ in} \\
&\quad \mathbb{S}[\![\texttt{cls = meta("cls", supers, dict)}]\!]\sigma
\end{aligned}
$$

Figure 6.17: Semantics of class declaration

6.2.12 Decorators

Functions and classes can be decorated. The decorator acts as a wrapper around the class or function object, as shown in Figure 6.18.

```
@decor                          def f(args):
def f(args):          ⇝           body
  body                          f = decor(f)
```

Figure 6.18: Transformation of decorators

Example 6.25 **`classmethod` decorator usecase**

A typical example is the use of the `classmethod` decorator to implement alternative constructors. This decorator creates a method bound to its class rather than its instance (the first argument is thus called `cls` instead of `self`). In the example of Listing 6.4, `from_str` behaves as an alternative constructor used to create `Date` objects from strings.

Listing 6.4: Example of classmethod decorator

```
1  class Date:
2    def __init__(self, d, m, y):
3      self.day = d
4      self.month = m
5      self.year = y
6
7    @classmethod
8    def from_str(cls, s):
9      d, m, y = s.split("/")
10     return cls(int(d), int(m), int(y))
11
12 d = Date.from_str("03/06/2021")
```

6.2.13 Calls

This section describes the general semantics of calls. Calls start by evaluating their caller and arguments. The class of the caller should have the `__call__` method, which is then invoked (a type error is raised otherwise). Function calls thus go through `function.__call__`, defined in Section 6.3.2. Similarly, methods eventually call `method.__call__`, which is also described in Section 6.3.2. Class instantiations rely on `type.__call__`, defined in Section 6.3.3.

$$\mathbb{E}_{cur}[\![\, c(e_1,\ldots,e_n)\,]\!](cur,e,h) \stackrel{\text{def}}{=} \quad \texttt{CALL\_FUNCTION \_PyObject\_Vectorcall \_PyObject\_MakeTpCall}$$
$$\text{letb } (cur,e,h), @_c = \mathbb{E}[\![\, e_c\,]\!](cur,e,h) \text{ in}$$
$$\text{letb } (cur,e,h), @_1 = \mathbb{E}[\![\, e_1\,]\!](cur,e,h) \text{ in}$$
$$\ldots$$
$$\text{letb } (cur,e,h), @_n = \mathbb{E}[\![\, e_n\,]\!](cur,e,h) \text{ in}$$
$$\text{if } \mathit{hasattr}(\mathit{type}(@_c), \texttt{"\_\_call\_\_"}) \text{ then}$$
$$\quad \text{return } \mathbb{E}[\![\, \mathit{type}(@_c).\texttt{\_\_call\_\_}(@_c, @_1, \ldots, @_n)\,]\!](cur,e,h)$$
$$\text{else return } \mathbb{S}[\![\, \texttt{raise TypeError}\,]\!](cur,e,h), \bot$$

Figure 6.19: Semantics of calls

6.2.14 Unary operators

The semantics of unary operators $\sim$, $+$ and $-$ consists in delegating to a corresponding method.

$$\mathbb{E}_{cur}[\![\, \sim expr\,]\!](cur,e,h) \stackrel{\text{def}}{=} \quad \texttt{UNARY\_INVERT PyNumber\_Invert slot\_nb\_invert}$$
$$\text{letb } (cur,e,h), @_e = \mathbb{E}[\![\, expr\,]\!](f,e,h) \text{ in}$$
$$\text{if } \mathit{hasattr}(\mathit{type}(@_e), \texttt{"\_\_invert\_\_"}) \text{ then return } \mathbb{E}[\![\, \mathit{type}(@_e).\texttt{\_\_invert\_\_}(@_e)\,]\!](cur,e,h)$$
$$\text{else return } \mathbb{S}[\![\, \texttt{raise TypeError}\,]\!](cur,e,h), \bot$$

Figure 6.20: Semantics of unary operator
(also applies to + (resp. -) with `__pos__` (resp. `__neg__`))

The unary `not` operator behaves differently. It explicitly casts its argument into a boolean and returns the opposite boolean.

$$\mathbb{E}_{cur}[\![\, \texttt{not } expr\,]\!](cur,e,h) \stackrel{\text{def}}{=} \quad \texttt{UNARY\_NOT PyObject\_IsTrue bool\_new}$$
$$\text{letb } (cur,e,h), @_b = \mathbb{E}[\![\, \texttt{bool}(expr)\,]\!](cur,e,h) \text{ in}$$
$$\text{if } @_b = @_{\texttt{True}} \text{ then return } (cur,e,h), @_{\texttt{False}} \text{ else return } (cur,e,h), @_{\texttt{True}}$$

Figure 6.21: Semantics of unary not

6.2.15 Binary operators

Contrary to unary operators, the semantics of binary operators is more involved, as Python will look both into the left-hand side and right-hand side objects for specific methods to perform

the operation. We start with the semantics of arithmetic operators and then describe the semantics of comparison operators.

6.2.15.1 Arithmetic operators

Python has 12 arithmetic operators, described in the first column of Figure 6.22. The other columns describe methods that will be used by the semantics of binary operators.

Given a binary operator †, the semantics of $e_1 \dagger e_2$ is described in Figure 6.23. The general idea is to follow a compatibility principle, where the binary method is resolved to the most derived class. We start by evaluating e_1 and e_2 into $@_1$ and $@_2$. We first look into $@_1$'s class to search for the standard operator's method, defined as $op(\dagger)$ in Figure 6.22. If it exists, we also check if $@_2$'s class has a reversed operator method $rop(\dagger)$, and if $@_2$'s class is a strict children of $@_1$'s class. If that is the case, we evaluate the call with the reversed method. If the result of this call is anything but the `NotImplemented` singleton object, we return the result. In other cases, we try to call the standard operator and similarly return if the result is different from `NotImplemented`. If none of these cases were successful and $@_2$'s reversed operator has not been tried yet, we try to call it, and check the result. If nothing works, a type error is raised.

†	op(†)	rop(†)	iop(†)
+	`__add__`	`__radd__`	`__iadd__`
-	`__sub__`	`__rsub__`	`__isub__`
*	`__mul__`	`__rmul__`	`__imul__`
@	`__matmul__`	`__rmatmul__`	`__imatmul__`
/	`__truediv__`	`__rtruediv__`	`__itruediv__`
//	`__floordiv__`	`__rfloordiv__`	`__ifloordiv__`
**	`__pow__`	`__rpow__`	`__ipow__`
<<	`__lshift__`	`__rlshift__`	`__ilshift__`
>>	`__rshift__`	`__rrshift__`	`__irshift__`
&	`__and__`	`__rand__`	`__iand__`
|	`__or__`	`__ror__`	`__ior__`
^	`__xor__`	`__rxor__`	`__ixor__`

Figure 6.22: Correspondance between binary operators and their methods

Remark 6.26 **Argument reversal in CPython**

In CPython's source code, the function `binary_op1` appears to be always called with the same argument order, compared to what has been described in the semantics. However, a wrapper in the low-level source code `wrap_binaryfunc_l` reverses the arguments for reversed calls (in `Objects/typeobject.c`). These wrappers will be described in more details in Remark 10.10.

Example 6.27 **Usecase of `__radd__` method in the subclassing case**

This example shows the purpose of the `__radd__` method, and in particular, the subclassing case. We use the code shown in Listing 6.5. It defines an `Int16` class denoting unsigned 16-bit integers. These objects can be transformed into integers through the `__int__` method, and the addition consists in calling the constructor once the computation over builtin integers is done. A subclass for 32-bit integers is defined; it overloads the constructor, the `__add__` method and adds a method `__radd__`. Given an `Int16` and an `Int32`, we would like their addition to be commutative. If we add a and b, `Int32.__add__` is called, and c

$$\mathbb{E}_{cur}[\![\, e_1 \dagger e_2 \,]\!](cur, e, h) \overset{\text{def}}{=} \quad \texttt{BINARY\_ADD PyNumber\_Add binary\_op1}$$

$$\begin{array}{l}
\quad \text{letb } (cur, e, h), @_1 = \mathbb{E}[\![\, e_1 \,]\!](f, e, h) \text{ in} \\
\quad \text{letb } (cur, e, h), @_2 = \mathbb{E}[\![\, e_2 \,]\!](f, e, h) \text{ in} \\
\quad \text{if } \mathit{hasattr}(\mathit{type}(@_1), \mathit{op}(\dagger)) \text{ then} \\
\quad\quad \text{if } \mathit{hasattr}(\mathit{type}(@_2), \mathit{rop}(\dagger)) \wedge \mathit{type}(@_2) \sqsubset \mathit{type}(@_1) \text{ then} \\
\quad\quad\quad \text{letb } (cur, e, h), @_r = \mathbb{E}[\![\, \mathit{type}(@_2).\mathit{rop}(\dagger)(@_2, @_1) \,]\!](cur, e, h) \text{ in} \\
\quad\quad\quad \text{if } \neg \mathit{isNotImplemented}(@_r) \text{ then return } (f, e, h), @_r \\
\quad\quad \text{letb } (cur, e, h), @_{@_r} = \mathbb{E}[\![\, \mathit{type}(@_1).\mathit{op}(\dagger)(@_1, @_2) \,]\!](cur, e, h) \text{ in} \\
\quad\quad \text{if } \neg \mathit{isNotImplemented}(@_r) \text{ then return } (f, e, h), @_r \\
\quad \text{if } \mathit{hasattr}(\mathit{type}(@_2), \mathit{rop}(\dagger)) \text{ then} \\
\quad\quad \text{letb } (cur, e, h), @_{@_r} = \mathbb{E}[\![\, \mathit{type}(@_2).\mathit{rop}(\dagger)(@_2, @_1) \,]\!](cur, e, h) \text{ in} \\
\quad\quad \text{if } \neg \mathit{isNotImplemented}(@_r) \text{ then return } (cur, e, h), @_{@_r} \\
\quad \text{return } \mathbb{S}[\![\, \texttt{raise TypeError} \,]\!](cur, e, h), \bot
\end{array}$$

Figure 6.23: Semantics of binary arithmetic operators

is thus an `Int32`. Without the `Int32.__radd__` method, the addition $b + a$ would yield an `Int16`.

Listing 6.5: Example usecase of both addition methods

```
class Int16:
    def __init__(self, value):
        self.value = value % 2**16

    def __int__(self):
        return self.value

    def __add__(self, other):
        return Int16(self.value + int(other))


class Int32(Int16):
    def __init__(self, value):
        self.value = value % 2**32

    def __add__(self, other):
        return Int32(self.value + int(other))

    __radd__ = __add__

a = Int32(2**16)
b = Int16(1)
c = a + b
d = b + a
assert isinstance(c, Int32)
assert isinstance(d, Int32)
```

The semantics of augmented assignments such as $x \dagger= expr$ is described in Figure 6.24. Python starts by checking if x has a special method for the augmented assignment, $iop(\dagger)$. This special method may be interesting for in-place operations reducing the memory footprint of the operation. For example, augmented additions to lists are performed in-place and avoid performing a new memory allocation. This method is called, and the result is returned if it is

different from `NotImplemented`. In all other cases, the operation is written as $x = x \dagger expr$ (but with the corresponding evaluated arguments to avoid double evaluations).

$$
\begin{array}{l}
\mathbb{S}_{cur}[\![\, x \dagger\!= expr \,]\!](cur, e, h) \stackrel{\text{def}}{=} \qquad \texttt{INPLACE\_ADD PyNumber\_InPlaceAdd binary\_iop1} \\
\quad \text{letb } (cur, e, h), @_x = \mathbb{E}[\![\, x \,]\!](cur, e, h) \text{ in} \\
\quad \text{letb } (cur, e, h), @_e = \mathbb{E}[\![\, expr \,]\!](cur, e, h) \text{ in} \\
\quad \text{if } hasattr(type(@_x), iop(\dagger)) \text{ then} \\
\qquad \text{letb } (cur, e, h), @_r = \mathbb{E}[\![\, type(@_x).iop(\dagger)(@_x, @_e) \,]\!](cur, e, h) \text{ in} \\
\qquad \text{if } \neg isNotImplemented(@_r) \text{ then return } \mathbb{S}[\![\, @_x = @_r \,]\!](cur, e, h) \\
\quad \text{return } \mathbb{S}[\![\, @_x = @_x \dagger @_e \,]\!](cur, e, h)
\end{array}
$$

Figure 6.24: Semantics of augmented assignments

6.2.15.2 Comparison operators

The semantics of Python's standard comparison operators (Figure 6.26) is close to the one of arithmetic operators. Its implementation uses a slightly different structure which we reflect in our semantics. The calls to reversed operators still exist but perform the symmetric translation of comparison operators (based on the equivalence of $x < y$ and $y > x$), described in Figure 6.25 Two last fallback cases are included for the equality and the difference operators: if nothing has been successful, Python will compare the addresses and return a boolean result accordingly.

$\diamond$	op($\diamond$)	swapop($\diamond$)
`<`	`__lt__`	`__gt__`
`<=`	`__le__`	`__ge__`
`==`	`__eq__`	`__eq__`
`!=`	`__ne__`	`__ne__`
`>`	`__gt__`	`__lt__`
`>=`	`__ge__`	`__le__`

Figure 6.25: Correspondance between binary operators and their methods

6.2.16 Other binary operators

There are four other binary operators defined through keywords in Python: `in`, `is`, and the lazy `or` and `and` boolean operators.

The `in` operator checks for membership of an element e_1 in an object e_2 (Figure 6.27). It tries to call the `__contains__` method of the container e_2, and returns a boolean. If the method does not exist, it searches for e_1 in the whole structure through an iteration.

The `is` operator checks that two objects are allocated at the same address (Figure 6.28).

Remark 6.28 **Negation of `in` and `is` operators**

The negation of the `in` and `is` operators can respectively be written `not in` and `is not`. It makes for more readable expressions such as `1 not in l` rather than `not 1 in l`.

The semantics of the boolean `or` and `and` operators are described in Figure 6.29. Both start by evaluating the left-hand side argument e_1, as well as a boolean cast of it. Depending on the result, either e_1 or e_2 is returned.

$\mathbb{E}_{cur}[\![e_1 \diamond e_2]\!](cur, e, h) \stackrel{\text{def}}{=}$ COMPARE_OP PyObject_RichCompare do_richcompare

letb $(cur, e, h), @_1 = \mathbb{E}[\![e_1]\!](cur, e, h)$ in
letb $(cur, e, h), @_2 = \mathbb{E}[\![e_2]\!](cur, e, h)$ in
if $type(@_2) \sqsubset type(@_1) \wedge hasattr(type(@_2), swapop(\diamond))$ then
  letb $(cur, e, h), @_r = \mathbb{E}[\![type(@_2).swapop(\diamond)(@_2, @_1)]\!]$ in
  if $\neg isNotImplemented(@_r)$ then $(cur, e, h), @_{@_r}$
if $hasattr(type(@_1), op(\diamond))$ then
  letb $(cur, e, h), @_{@_r} = \mathbb{E}[\![type(@_1).op(\diamond)(@_1, @_2)]\!]$ in
  if $\neg isNotImplemented(@_r)$ then $(cur, e, h), @_{@_r}$
if $type(@_2) \not\sqsubset type(@_1) \wedge hasattr(type(@_2), swapop(\diamond))$ then
  letb $(cur, e, h), @_r = \mathbb{E}[\![type(@_2).swapop(\diamond)(@_2, @_1)]\!]$ in
  if $\neg isNotImplemented(@_r)$ then $(cur, e, h), @_{@_r}$
if $\diamond =$ `==` then
  if $@_1 = @_2$ then return $\mathbb{E}[\![$ `True` $]\!](cur, e, h)$ else return $\mathbb{E}[\![$ `False` $]\!](cur, e, h)$
if $\diamond =$ `!=` then
  if $@_1 \neq @_2$ then return $\mathbb{E}[\![$ `True` $]\!](cur, e, h)$ else return $\mathbb{E}[\![$ `False` $]\!](cur, e, h)$
return $\mathbb{S}[\![$ `raise TypeError` $]\!](cur, e, h), \bot$

Figure 6.26: Semantics of binary comparison operators

$\mathbb{E}_{cur}[\![e_1$ `in` $e_2]\!](cur, e, h) \stackrel{\text{def}}{=}$ COMPARE_OP PySequence_Contains slot_sq_contains

letb $(cur, e, h), @_1 = \mathbb{E}[\![e_1]\!](cur, e, h)$ in
letb $(cur, e, h), @_2 = \mathbb{E}[\![e_2]\!](cur, e, h)$ in
if $hasattr(type(@_2),$ `"__contains__"`$)$ then
  return $\mathbb{E}[\![bool(type(@_2).$`__contains__`$(@_2, @_1))]\!](cur, e, h)$
else
  letb $(cur, e, h) = \mathbb{S}[\![$
```
fresh_found = False
for x in @2:
  if x == @1:
    fresh_found = True
    break
```
$]\!]\sigma$ in
  return $\mathbb{E}[\![$ `fresh_found` $]\!](cur, e, h)$

Figure 6.27: Semantics of the `in` operator

Remark 6.29 **Return type of `or` and `and` operators**

The semantics of the `or` and `and` operators does not always return a boolean. For example `1 and 'a'` will return `'a'`, since `bool(1)` holds.

$$\mathbb{E}_{cur}[\![\, e_1 \texttt{ is } e_2 \,]\!](cur, e, h) \stackrel{\text{def}}{=} \qquad \texttt{COMPARE\_OP cmp\_outcome}$$
$$\quad \text{letb } (cur, e, h), @_1 = \mathbb{E}[\![\, e_1 \,]\!](cur, e, h) \text{ in}$$
$$\quad \text{letb } (cur, e, h), @_2 = \mathbb{E}[\![\, e_2 \,]\!](cur, e, h) \text{ in}$$
$$\quad \text{if } @_1 = @_2 \text{ then return } \mathbb{E}[\![\, \texttt{True} \,]\!](cur, e, h)$$
$$\quad \text{else return } \mathbb{E}[\![\, \texttt{False} \,]\!](cur, e, h)$$

Figure 6.28: Semantics of the `is` operator

$$\mathbb{E}_{cur}[\![\, e_1 \texttt{ or } e_2 \,]\!](cur, e, h) \stackrel{\text{def}}{=} \qquad \texttt{JUMP\_IF\_TRUE\_OR\_POP}$$
$$\quad \text{letb } (cur, e, h), @_1 = \mathbb{E}[\![\, e_1 \,]\!](cur, e, h) \text{ in}$$
$$\quad \text{if } bool(@_1) \text{ then } (cur, e, h), @_1 \text{ else } \mathbb{E}[\![\, @_2 \,]\!](f, e, h)$$
$$\mathbb{E}_{cur}[\![\, e_1 \texttt{ and } e_2 \,]\!](cur, e, h) \stackrel{\text{def}}{=} \qquad \texttt{JUMP\_IF\_FALSE\_OR\_POP}$$
$$\quad \text{letb } (cur, e, h), @_1 = \mathbb{E}[\![\, e_1 \,]\!](cur, e, h) \text{ in}$$
$$\quad \text{if } bool(@_1) \text{ then } \mathbb{E}[\![\, e_2 \,]\!](cur, e, h) \text{ else } (cur, e, h), @_1$$

Figure 6.29: Semantics of the `or` and `and` operators

6.3 Builtin objects

6.3.1 Object

Every class is a subclass of `object`. As such, most method calls are resolved by default into this class.

We start by defining the semantics of the object creation and instantiation methods in Figure 6.30. Object creation starts by evaluating the argument, which should be a class to instantiate. If that is the case, an allocation is performed, and the heap is updated to reflect the effect. Otherwise, a type error is raised. Object initialization does not perform anything by default, and returns `None`.

$$\mathbb{E}_{cur}[\![\, \texttt{object.\_\_new\_\_}(expr) \,]\!](cur, e, h) \stackrel{\text{def}}{=} \qquad \texttt{tp\_field object\_new}$$
$$\quad \text{letb } (cur, e, h), @_e = \mathbb{E}[\![\, expr \,]\!](cur, e, h) \text{ in}$$
$$\quad \text{if } isinstance(@_e, \texttt{type}) \text{ then}$$
$$\quad\quad \text{letb } (cur, e, h), @_c = \mathbb{E}[\![\, alloc\_addr \,]\!](cur, e, h) \text{ in}$$
$$\quad\quad \text{return } (cur, e, h[@_c \mapsto \mathsf{Inst}(@_e), \emptyset]), @_c$$
$$\quad \text{else return } \mathbb{S}[\![\, \texttt{raise TypeError} \,]\!](f, e, h), \bot$$
$$\mathbb{E}_{cur}[\![\, \texttt{object.\_\_init\_\_}(self) \,]\!](cur, e, h) \stackrel{\text{def}}{=} \qquad \texttt{tp\_field object\_init}$$
$$\quad \text{return } \mathbb{E}[\![\, \texttt{None} \,]\!](cur, e, h)$$

Figure 6.30: Semantics of object creation and instantiation

As mentioned in the semantics of attributes, the `__getattribute__` method is rarely overloaded and usually defaults to `object.__getattribute__`, whose semantics is described in

Figure 6.31. The function starts by evaluating its arguments and checking that $name$ is a string object. Given the string n, we search for it in the parents' classes of the object using *mro_search*. If a result is found, we call this object $descr$, and check that it is a data descriptor. That is, we check that $descr$ has a `__get__` method, as well as a `__set__` or a `__delete__` method. If that is the case, the result of the access is the call to the `__get__` method of this data descriptor. Otherwise, we search for the field at the instance's level. Two fallback cases exist if the $descr$ object exists but is not a data descriptor and the field does not exist at the instance's level: we first try to call its `__get__` method if it exists and return $descr$ otherwise. If everything fails, an attribute error is raised.

$$\mathbb{E}_{\mathsf{cur}}[\![\,\texttt{object.\_\_getattribute\_\_}(obj, name)\,]\!](cur, e, h) \overset{\text{def}}{=}$$

`tp_field _PyObject_GenericGetAttrWithDict`

letb $(cur, e, h), @_o = \mathbb{E}[\![\, obj \,]\!](cur, e, h)$ in
letb $(cur, e, h), @_n = \mathbb{E}[\![\, name \,]\!](cur, e, h)$ in
if $\neg$*isinstance*$(@_n, \text{str})$ then return $\mathbb{S}[\![\,\texttt{raise TypeError}\,]\!](cur, e, h), \bot$ else
let $\texttt{str}(n) =$ fst $\circ h(@_s)$ in
letcases $(f, e, h), @_{descr} = \mathbb{E}[\![\, mro\_search(type(@_o), n)\,]\!](f, e, h)$ in
if $@_{descr} \neq \bot$ then
  if *hasattr*$(type(@_{descr}), \texttt{"\_\_get\_\_"}) \wedge$
      (*hasattr*$(type(@_{descr}), \texttt{"\_\_set\_\_"}) \vee$ *hasattr*$(type(@_{descr}), \texttt{"\_\_delete\_\_"})$ then
    return $\mathbb{E}[\![\, type(@_{descr}).\texttt{\_\_get\_\_}(@_{descr}, @_o, type(@_o))\,]\!](f, e, h)$
if *has_field*$(@_o, n)$ then return $\mathbb{E}[\![\, get\_field(@_o, n)\,]\!](f, e, h)$ else
if $@_{descr} \neq \bot$ then
  if *hasattr*$(type(@_{descr}), \texttt{"\_\_get\_\_"})$ then
    return $\mathbb{E}[\![\, type(@_{descr}).\texttt{\_\_get\_\_}(@_{descr}, @_o, type(@_o))\,]\!](f, e, h)$
  else return $@_{descr}, (f, e, h)$
return $\mathbb{S}[\![\,\texttt{raise AttributeError}\,]\!](f, e, h), \bot$

Figure 6.31: Semantics of attribute access through `object`

Remark 6.30 **Purpose of data descriptors**

The purpose of data descriptors is to hide calls to custom getter and setter functions when attributes are accessed or set. It is mainly used by Python classes defined in C, such as the member descriptors described later in this thesis (Remark 10.11 and Example 11.4). The fallback case of the `__get__` method is used more often. For example, methods are bound to instances using this approach (cf. Figure 6.35), and the `classmethod` and `staticmethod` use it too (cf. Figure 6.36). For now, we show a custom example in Example 6.31.

Example 6.31 **A data descriptor usecase**

An example use case of a data descriptor is shown in Listing 6.6. We are interested in describing a subset of the Unix file permissions. The meaning of each number is provided in the `perms` dictionary. The `PermissionDescriptor` class will store the permission as numbers and display the corresponding string when the attribute is accessed. The `File` class has a class-level attribute `rights`. We create a `File` instance line 14, and define its rights

to be “execute only” with the number 1. We also set the value of the instance field rights to 3 in a line 16. At the first attribute access line 18, the result is “execute only”, corresponding to the data descriptor case in the semantics. By removing the `__set__` method from the `PermissionDescriptor`, the class is not considered as a data descriptor anymore. Since a has a field `rights` set to 3, the second call to print displays 3. By removing the instance-level field, the next attribute access will fall back to calling the `__get__` method of the `PermissionDescriptor`. If we override the class-level `rights` attribute, the last fallback case is executed, and we get 42.

Listing 6.6: Data descriptor example

```
1  perms = {0: "no permission", 1: "execute only", 2: "write only", 4: "read only"}
2
3  class PermissionDescriptor:
4      def __get__(self, obj, objtype):
5          return perms[obj._rights]
6
7      def __set__(self, obj, value):
8          assert value in perms
9          obj._rights = value
10
11 class File:
12     rights = PermissionDescriptor()
13
14 a = File()
15 a.rights = 1
16 a.__dict__['rights'] = 3
17
18 print(a.rights)
19 # prints "execute only", data descriptor access
20 del PermissionDescriptor.__set__
21 print(a.rights)
22 # prints 3, instance access
23 del a.__dict__['rights']
24 print(a.rights)
25 # prints "execute only", __get__ fallback access
26 File.rights = 42
27 print(a.rights)
28 # prints 42, descr fallback access
```

Setting an attribute (Figure 6.32) is less complex: once the arguments have been evaluated and type-checked, we look for a descriptor in the parent classes. We do not check that this descriptor is a data descriptor. If it exists, we call its `__set__` method. Otherwise, the field is defined at the instance’s level. The semantics of attribute deletion (Figure 6.33) is similar to this case.

6.3.2 Functions and methods

The semantics of function calls and the return statement is shown in Figure 6.34. The return statements stores the returned expression into an auxiliary variable and interrupts the normal execution flow with the *ret* flow token. We present the simplified case of function call where no arguments are optional, or keyword-only. A function call starts by assigning the arguments their given value, and setting local variables to the locally undefined value, which is an artifact of our semantics. Once the body has been executed, the state is inspected. If the usual control flow has been interrupted by a return statement, we keep the address of the returned object. Otherwise, the returned value is `None`. We then clean the state from the arguments and the local variables of the function.

$\mathbb{E}_{cur}[\![\,\texttt{object.\_\_setattr\_\_}(obj, n, v)\,]\!](cur, e, h) \overset{\text{def}}{=}$ `tp_field _PyObject_GenericSetAttrWithDict`

letb $(cur, e, h), @_o = \mathbb{E}[\![\, obj \,]\!](cur, e, h)$ in
letb $(cur, e, h), @_n = \mathbb{E}[\![\, n \,]\!](cur, e, h)$ in
if $\neg$*isinstance*$(@_n, \text{str})$ then return $\mathbb{S}[\![\,\texttt{raise TypeError}\,]\!](f, e, h), \bot$ else
let $\texttt{str}(n) =$ fst $\circ h(@_n)$ in
letb $(cur, e, h), @_v = \mathbb{E}[\![\, v \,]\!](cur, e, h)$ in
letcases $(f, e, h), @_{descr} = \mathbb{E}[\![\, \mathit{mro\_search}(\mathit{type}(@_o), n) \,]\!](f, e, h)$ in
letcases $\sigma =$
  if $descr \neq \bot \wedge$ *hasattr*$(\mathit{type}(@_{descr}), \texttt{"\_\_set\_\_"})$ then
    $\mathbb{S}[\![\, \mathit{type}(@_{descr}).\texttt{\_\_set\_\_}(@_{descr}, @_o, @_v) \,]\!](f, e, h)$
  else $\mathbb{S}[\![\, \mathit{set\_field}(@_o, n, @_v) \,]\!](f, e, h)$
if $\sigma = \bot$ then $\mathbb{S}[\![\,\texttt{raise AttributeError}\,]\!](f, e, h)$ else $\mathbb{E}[\![\,\texttt{None}\,]\!]\sigma$

Figure 6.32: Semantics of attribute definition through `object`

$\mathbb{E}_{cur}[\![\,\texttt{object.\_\_delattr\_\_}(obj, name)\,]\!](cur, e, h) \overset{\text{def}}{=}$
`tp_field _PyObject_GenericSetAttrWithDict`

letb $(cur, e, h), @_o = \mathbb{E}[\![\, obj \,]\!](cur, e, h)$ in
letb $(cur, e, h), @_n = \mathbb{E}[\![\, name \,]\!](cur, e, h)$ in
if $\neg$*isinstance*$(@_n, \text{str})$ then return $\mathbb{S}[\![\,\texttt{raise TypeError}\,]\!](f, e, h), \bot$ else
let $\texttt{str}(n) =$ fst $\circ h(@_n)$ in
letcases $(f, e, h), @_{descr} = \mathbb{E}[\![\, \mathit{mro\_search}(\mathit{type}(@_o), n) \,]\!](f, e, h)$ in
letcases $\sigma =$
  if $descr \neq \bot \wedge$ *hasattr*$(\mathit{type}(@_{descr}), \texttt{"\_\_delete\_\_"})$ then
    $\mathbb{S}[\![\, \mathit{type}(@_{descr}).\texttt{\_\_delete\_\_}(@_{escr}, @_o) \,]\!](f, e, h)$
  else $\mathbb{S}[\![\, \mathit{del\_field}(@_o, n) \,]\!](f, e, h)$
if $\sigma = \bot$ then $\mathbb{S}[\![\,\texttt{raise AttributeError}\,]\!](f, e, h)$ else $\mathbb{E}[\![\,\texttt{None}\,]\!]\sigma$

Figure 6.33: Semantics of attribute deletion through `object`

Remark 6.32 **Supporting recursive calls**

In its current state, the semantics does not handle recursive calls. One way to support them would be to partition the environment by the callstack.

Recursion is not used a lot in Python. In particular, there is no tail-call optimization, and the default recursion stack has depth 1000.

We now explain how methods are allocated and called. We start with the example provided in Listing 6.7. The class `A` defines a function `id`. If we access the function through the instance, as done in line 5, the result will be a method corresponding to a partial application of the function where `self` is the instance. As such the call line 7 is valid and will return a tuple consisting in the instance defined at line 4 and the integer 3.

The construction of the method at line 5 relies on the semantics of the `__getattribute__` method of `object`, which calls `function.__get__`, defined in Figure 6.35 (this is not a data

$$
\begin{array}{l}
\mathbb{E}_{cur}[\![\ \texttt{return}\ expr\ ]\!]\Sigma \stackrel{\text{def}}{=} \\
\quad \text{letb } \Sigma_1 = \mathbb{S}[\![\ return\_var = expr\ ]\!]\Sigma \text{ in} \\
\quad \{\,(f,e,h) \in \Sigma_1 \mid f \neq cur\,\} \cup \{\,(ret,e,h) \mid (cur,e,h) \in \Sigma_1\,\} \\
\mathbb{E}_{cur}[\![\ \texttt{function.\_\_call\_\_}(func, e_1, \cdots, e_n)\ ]\!](cur,e,h) \stackrel{\text{def}}{=} \qquad \texttt{tp\_field function\_call} \\
\quad \text{letb } (cur,e,h), @_f = \mathbb{E}[\![\ func\ ]\!](cur,e,h) \text{ in} \\
\quad \text{if } \neg isinstance(@_f, \texttt{function}) \text{ then return } \mathbb{S}[\![\ \texttt{raise TypeError}\ ]\!](cur,e,h) \text{ else} \\
\quad \text{let } \mathsf{Fun}(fname, (arg_1,\ldots,arg_m), (local_1,\ldots,local_i), body) = \mathsf{fst} \circ h(@_f) \text{ in} \\
\quad \text{if } m \neq n \text{ then return } \mathbb{S}[\![\ \texttt{raise TypeError}\ ]\!](f,e,h) \\
\quad \text{letb } \sigma = \mathbb{S}[\![\ local_i = \texttt{LocalUndef}\ ]\!] \circ \ldots \circ \mathbb{S}[\![\ local_1 = \texttt{LocalUndef}\ ]\!] \\
\qquad\qquad \circ \mathbb{S}[\![\ arg_n = e_n\ ]\!] \circ \ldots \circ \mathbb{S}[\![\ arg_1 = e_1\ ]\!]\sigma \text{ in} \\
\quad \text{letcases } (f,e,h) = \mathbb{S}[\![\ body\ ]\!]\sigma \text{ in} \\
\quad \text{let } r = \\
\qquad \text{if } f = ret \text{ then } e[return\_var] \text{ else} \\
\qquad \text{if } f = cur \text{ then } @_{\texttt{None}} \text{ else} \\
\qquad \text{else } \bot \text{ in} \\
\quad \text{letb } (\_,e,h) = \mathbb{S}[\![\ \texttt{del}\ return\_var\ ]\!] \circ \mathbb{S}[\![\ \texttt{del}\ local_i\ ]\!] \circ \ldots \circ \mathbb{S}[\![\ \texttt{del}\ local_1\ ]\!] \\
\qquad\qquad \circ \mathbb{S}[\![\ \texttt{del}\ arg_n\ ]\!] \circ \ldots \circ \mathbb{S}[\![\ \texttt{del}\ arg_1\ ]\!](cur,e,h) \text{ in} \\
\quad \text{return } (f,e,h), r
\end{array}
$$

Figure 6.34: Semantics of function calls

descriptor case as defined in Section 6.3.1, but the fallback case of `object.__getattribute__`). `function.__get__` calls the method constructor, and its arguments are the function and the instance. The method is created by `method.__new__`, which allocates the method object and keeps the instance and the function inside its structure. A method call is transformed into a function call, where the instance to which the method is bound is added as the first argument.

Listing 6.7: Example of method creation

```
class A:
  def id(self, x): return (self, x)

a = A()
m = a.id
# <bound method A.id of <__main__.A object at 0x...>>
t = m(3)
# (<__main__.A object at 0x...>, 3)
```

Although the `get` descriptor of `function` creates methods for function accesses from classes, it is sometimes interesting to not have the first argument reserved for the instance. Two alternatives are to have the class being the first argument, or having no reserved argument at all. The decorators `classmethod` and `staticmethod` provide these behaviors.

The semantics of these decorators is shown in Figure 6.36. Their initialization stores the underlying function object into a `__func__` attribute. The `staticmethod` decorator removes the first instance argument in the functions. When accessed through an attribute, the underlying function will be directly returned, instead of the method usually created by `function.__get__`. The `classmethod` decorator transforms the first argument as the class rather than the instance.

$\mathbb{E}_{cur}[\![\,\texttt{function.\_\_get\_\_}(func, i, \_)\,]\!](cur, e, h) \stackrel{\text{def}}{=}$ `tp_field func_descr_get`
letb $(cur, e, h), @_f = \mathbb{E}[\![\, func\,]\!](cur, e, h)$ in
letb $(cur, e, h), @_i = \mathbb{E}[\![\, i\,]\!](cur, e, h)$ in
if $\neg isinstance(@_f, \texttt{function})$ then return $\mathbb{S}[\![\,\texttt{raise TypeError}\,]\!](cur, e, h)$ else
return $\mathbb{E}[\![\,\texttt{method}(@_f, @_i)\,]\!](f, e, h)$

$\mathbb{E}_{cur}[\![\,\texttt{method.\_\_new\_\_}(\texttt{method}, func, i)\,]\!](cur, e, h) \stackrel{\text{def}}{=}$ `tp_field PyMethod_New`
letb $(cur, e, h), @_m = \mathbb{E}[\![\, m\,]\!](cur, e, h)$ in
letb $(cur, e, h), @_f = \mathbb{E}[\![\, func\,]\!](cur, e, h)$ in
letb $(cur, e, h), @_i = \mathbb{E}[\![\, i\,]\!](cur, e, h)$ in
if $\neg isinstance(@_f, \texttt{function})$ then return $\mathbb{S}[\![\,\texttt{raise TypeError}\,]\!](cur, e, h)$ else
letb $(cur, e, h), @_a = \mathbb{E}[\![\, alloc\_addr\,]\!](cur, e, h)$ in
return $(f, e, h[@_a \mapsto \textsf{Method}(@_i, @_f)]), a$

$\mathbb{E}_{cur}[\![\,\texttt{method.\_\_call\_\_}(m, e_1, \ldots, e_n)\,]\!](cur, e, h) \stackrel{\text{def}}{=}$ `tp_field method_call`
letb $(cur, e, h), @_m = \mathbb{E}[\![\, m\,]\!](cur, e, h)$ in
if $\neg isinstance(@_m, \texttt{method})$ then return $\mathbb{S}[\![\,\texttt{raise TypeError}\,]\!](cur, e, h)$ else
let $\textsf{Method}(@_{self}, @_{func}) = \textsf{fst} \circ h(@_m)$ in
return $\mathbb{E}[\![\,\texttt{function.\_\_call\_\_}(@_{func}, @_{self}, e_1, \cdots, e_n)\,]\!](cur, e, h)$

Figure 6.35: Semantics of methods

$\mathbb{E}_{cur}[\![\,\texttt{staticmethod.\_\_init\_\_}(inst, f)\,]\!](cur, e, h) \stackrel{\text{def}}{=}$ `tp_field sm_init`
return $\mathbb{E}[\![\,\texttt{None}\,]\!] \circ \mathbb{S}[\![\, inst.\texttt{\_\_func\_\_} = f\,]\!](cur, e, h)$

$\mathbb{E}_{cur}[\![\,\texttt{staticmethod.\_\_get\_\_}(self, obj, type)\,]\!](cur, e, h) \stackrel{\text{def}}{=}$ `tp_field sm_descr_get`
return $\mathbb{E}[\![\, self.\texttt{\_\_func\_\_}\,]\!](cur, e, h)$

$\mathbb{E}_{cur}[\![\,\texttt{classmethod.\_\_init\_\_}(inst, f)\,]\!](cur, e, h) \stackrel{\text{def}}{=}$ `tp_field sm_init`
return $\mathbb{E}[\![\,\texttt{None}\,]\!] \circ \mathbb{S}[\![\, inst.\texttt{\_\_func\_\_} = f\,]\!](cur, e, h)$

$\mathbb{E}_{cur}[\![\,\texttt{classmethod.\_\_get\_\_}(self, obj, type)\,]\!](cur, e, h) \stackrel{\text{def}}{=}$ `tp_field cm_descr_get`
return $\mathbb{E}[\![\,\texttt{method}(self.\texttt{\_\_func\_\_}, type)\,]\!](cur, e, h)$

Figure 6.36: Semantics of the classmethod and staticmethod decorators

6.3.3 Type

Class instantiation is performed through calls such as `A(args)`. Since `A` is a class, this call usually ends up being rewritten into `type.__call__(A, args)` (the only exception is when `A` has a metaclass which is different from `type`). The semantics of this function is shown in Figure 6.37. It starts by calling the `__new__` method of the class. If the returned type is not a subtype of the class, or if the class has no initialization method, the instance is returned. Otherwise, the initialization method is called; it should return `None`.

The creation of a new class through `type.__new__` is shown in Figure 6.38. It computes a linear ordering of the parents' classes (this operation may fail and raise an exception). Then, the class is allocated and its address returned.

$$\mathbb{E}_{cur}[\![\,\texttt{type.\_\_call\_\_}(self, arg_1, \ldots, arg_n)\,]\!](cur, e, h) \stackrel{\text{def}}{=} \qquad \texttt{tp\_field type\_call}$$
$$\begin{array}{l}
\quad \text{letb } (cur, e, h), @_{self} = \mathbb{E}[\![\, self \,]\!](cur, e, h) \text{ in} \\
\quad \text{letb } (cur, e, h), @_1 = \mathbb{E}[\![\, arg_1 \,]\!](cur, e, h) \text{ in} \\
\quad \ldots \\
\quad \text{letb } (cur, e, h), @_n = \mathbb{E}[\![\, arg_n \,]\!](cur, e, h) \text{ in} \\
\quad \text{if } \neg \mathit{hasattr}(@_{self}, \texttt{"__new__"}) \text{ then return } \mathbb{S}[\![\,\texttt{raise TypeError}\,]\!](cur, e, h), \bot \\
\quad \text{letb } (f, e, h), @_{new} = \mathbb{E}[\![\, @_{self}.\texttt{__new__}(@_1, \ldots, @_n)\,]\!](f, e, h) \text{ in} \\
\quad \text{if } \mathit{type}(@_{new}) \not\sqsupseteq @_{self} \vee \neg \mathit{hasattr}(\mathit{type}(@_{new}), \texttt{"__init__"}) \text{ then return } (cur, e, h), @_{new} \\
\quad \text{letb } (f, e, h), @_{init} = \mathbb{E}[\![\, @_{self}.\texttt{__init__}(@_{new}, @_1, \ldots, @_n)\,]\!](f, e, h) \text{ in} \\
\quad \text{if } \mathit{isNone}\ @_{init} \text{ then return } (cur, e, h), @_{init} \\
\quad \text{else return } \texttt{raise TypeError}, \bot
\end{array}$$

Figure 6.37: Semantics of class call

$$\mathbb{E}_{cur}[\![\,\texttt{type.\_\_new\_\_}(meta, cls, supers, attrs)\,]\!](cur, e, h) \stackrel{\text{def}}{=} \qquad \texttt{tp field type\_new}$$
$$\begin{array}{l}
\quad \text{letb } (f, e, h), mro = \mathit{compute_mro}(supers) \text{ in} \\
\quad \text{letb } (cur, e, h), @ = \mathbb{E}[\![\, \mathit{alloc_addr} \,]\!](f, e, h) \text{ in} \\
\quad \text{return } (f, e, h[@ \mapsto \mathsf{Class}(cls, meta, mro, attrs)]), @
\end{array}$$

Figure 6.38: Semantics of class creation

Remark 6.33 **Different definitions of `type.__new__`**

Actually, both the class creation `type.__new__` (Figure 6.38) and the typeof operator *type* (Figure 6.7) call `type.__new__`, but we decided to split their semantics for the sake of clarity.

The semantics of attribute accesses for classes (Figure 6.39) is different from the one of objects described previously. The first reason is to handle attribute inheritance at the class level. For example, if class `A` has an attribute `a`, and `B` inherits from `A`, `object.__getattribute__(B, 'a')` raises an attribute error (since it searches in the type of B, i.e. the type object). By comparison, `type.__getattribute__` performs a search at the class level too, allowing to resolve the attribute for `B`. This different semantics also makes for more coherent use of the standard `staticmethod` and `classmethod` decorators. For example, with the `classmethod` decorator, the access to `A.f` in Listing 6.8 should give a bound method rather than expose the `classmethod` object. This works with `type.__getattribute__(A, 'f')`, but would expose the classmethod object with `object.__getattribute__`.

Listing 6.8: Classmethod example

```
class A:
  @classmethod
  def f(c): print(c)

# >>> type.__getattribute__(A, 'f') # same as A.f
# <bound method A.f of <class '__main__.A'>>
# >>> object.__getattribute__(A, 'f')
# <classmethod object at 0x7eff1fdbc130>
```

$\mathbb{E}_{cur}[\![\texttt{type.\_\_getattribute\_\_}(typ, name)]\!](cur, e, h) \stackrel{\text{def}}{=}$ `tp_field type_getattro`

letb $(cur, e, h), @_{typ} = \mathbb{E}[\![typ]\!](cur, e, h)$ in
letb $(cur, e, h), @_{name} = \mathbb{E}[\![name]\!](cur, e, h)$ in
letb $(cur, e, h), @_{meta} = \mathbb{E}[\![\mathit{mro\_search}(type(@_{typ}), @_{name})]\!](cur, e, h)$ in
if $@_{meta} \neq \bot$ then
  if $hasattr(type(@_{meta}), \texttt{"\_\_get\_\_"}) \wedge$
      $(hasattr(type(@_{meta}), \texttt{"\_\_set\_\_"}) \vee hasattr(type(@_{meta}), \texttt{"\_\_delete\_\_"}))$ then
    return $\mathbb{E}[\![type(@_{meta}).\texttt{\_\_get\_\_}(@_{meta}, @_{typ}, type(@_{typ}))]\!](cur, e, h)$
letb $(cur, e, h), @_{attr} = \mathbb{E}[\![\mathit{mro\_search}(@_{typ}, @_{name})]\!](cur, e, h)$ in
if $@_{attr} \neq \bot$ then
  if $hasattr(type(@_{attr}), \texttt{"\_\_get\_\_"})$ then
    return $\mathbb{E}[\![type(@_{attr}).\texttt{\_\_get\_\_}(@_{attr}, \texttt{None}, @_{typ})]\!](cur, e, h)$
  else return $(cur, e, h), @_{attr}$
if $@_{meta} \neq \bot$ then
  if $hasattr(type(@_{meta}), \texttt{"\_\_get\_\_"})$ then
    return $\mathbb{E}[\![type(@_{meta}).\texttt{\_\_get\_\_}(@_{meta}, @_{typ}, type(@_{typ}))]\!](cur, e, h)$
  else return $(cur, e, h), @_{meta}$
return $\mathbb{S}[\![\texttt{raise AttributeError}]\!](cur, e, h), \bot$

Figure 6.39: Semantics of attribute accesses for classes

6.3.4 Booleans

The boolean conversion function is described in Figure 6.40. It starts by evaluating its argument. If the evaluation returns a boolean, the conversion is finished. Otherwise, we try calling the `__bool__` method and check that it returns a boolean. If the object has no `__bool__` method but it has a `__len__` method, we call it through the `len` builtin function[8], and return `True` when the object's length is positive. If everything fails, the result is `True` (the boolean conversion never raises an exception).

6.3.5 Integers

The integer creation function is described in Figure 6.41. If the argument is an integer, it is just returned. If the argument's class has either of the `__int__`, `__index__` or `__trunc__` methods, the methods are called, and the result is type-checked. If the argument is a string-like object, a conversion from string to integer is performed. If nothing succeeds, a type error is raised.

6.3.6 Range objects

Range objects are defined by a starting point, a stopping point, and a step. All three arguments should be integers. When iterated upon, these `range(start, stop, step)` objects returns elements of the sequence $start + k * step$, where k is an non-negative integer, until $stop$ is reached. We do not detail their semantics, but `range` objects are supported by our analyses.

[8] Which checks that the returned object is a non-negative integer.

$\mathbb{E}_{cur}[\![\, \texttt{bool.\_\_new\_\_}(expr) \,]\!](cur, e, h) \stackrel{\text{def}}{=}$ `tp_field bool_new PyObject_IsTrue`

letb $(cur, e, h), @_e = \mathbb{E}[\![\, expr \,]\!](f, e, h)$ in
if *isinstance*$(@_e, \texttt{bool})$ then return $(f, e, h), @_e$
if *isinstance*$(@_e, \texttt{NoneType})$ then return $(f, e, h), @_{\texttt{False}}$
else if *hasattr*(*type*$(@_e)$, "`__bool__`") then
  letb $(cur, e, h), @_r = \mathbb{E}[\![\, type(@_e).\texttt{\_\_bool\_\_}(e) \,]\!](f, e, h)$ in
  if *isinstance*$(@_r, bool)$ then return $(cur, e, h), @_r$
  else return $\mathbb{S}[\![\, \texttt{raise TypeError} \,]\!](cur, e, h), \bot$
else if *hasattr*(*type*$(@_e)$, "`__len__`") then
  letb $(f, e, h), @_{int} = \mathbb{E}[\![\, \texttt{len}(e) \,]\!](f, e, h)$ in
  let $\texttt{int}(i), \bot = h(@_{int})$ in
  if $i > 0$ then return `True` else return `False`
else return `True`

Figure 6.40: Semantics of boolean cast

$\mathbb{E}_{cur}[\![\, \texttt{int.\_\_new\_\_}(\texttt{int}, x) \,]\!](cur, e, h) \stackrel{\text{def}}{=}$ `tp_field long_new_impl PyNumber_Long`

letb $(cur, e, h), @_x = \mathbb{E}[\![\, x \,]\!](cur, e, h)$ in
if *type*$(@_x) = \texttt{int}$ then return $(cur, e, h), @_x$
if *hasattr*(*type*$(@_x)$, "`__int__`") then
  letb $(cur, e, h), @_r = \mathbb{E}[\![\, type(@_x).\texttt{\_\_int\_\_}(@_x) \,]\!](cur, e, h)$ in
  if *isinstance*$(@_r, \texttt{int})$ then return $(cur, e, h), @_r$
  else return $\mathbb{S}[\![\, \texttt{raise TypeError} \,]\!](cur, e, h), \bot$
if *hasattr*(*type*$(@_x)$, "`__index__`") then
  letb $(cur, e, h), @_r = \mathbb{E}[\![\, type(@_a).\texttt{\_\_index\_\_}(@_a) \,]\!](f, e, h)$ in
  if *isinstance*$(@_r, \texttt{int})$ then return $(cur, e, h), @_r$
  else return $\mathbb{S}[\![\, \texttt{raise TypeError} \,]\!](cur, e, h), \bot$
if *hasattr*(*type*$(@_x)$, "`__trunc__`") then
  letb $(cur, e, h), @_r = \mathbb{E}[\![\, type(@_x).\texttt{\_\_trunc\_\_}(@_x) \,]\!](cur, e, h)$ in
  return $\mathbb{E}[\![\, int(@_r) \,]\!](cur, e, h)$
if *isinstance*$(@_x, (\texttt{str}, \texttt{bytes}, \texttt{bytearray}))$ then
  let $\texttt{str}(s) = h(@_x)$ in
  letb $(cur, e, h), i =$ int_of_string s in
  return $\mathbb{E}[\![\, \texttt{int}(i) \,]\!](cur, e, h)$
return $\mathbb{S}[\![\, \texttt{raise TypeError} \,]\!](cur, e, h), \bot$

Figure 6.41: Semantics of integer creation

6.3.7 Containers

There are four standard containers in Python:

▷ **Lists** are mutable. They can be considered as arrays given the constant time index access.
▷ **Sets** are mutable structures with no duplicate (a variant called `frozenset` is immutable).
▷ **Tuples** are immutable structures.
▷ **Dictionaries** are mutable associative maps.

Python requires that set elements and dictionaries' keys should be immutable elements. Since Python cannot check this property at runtime, the requirement is actually that these elements should have a `__hash__` method. The documentation asks for the following constraint to be satisfied:[9]

> The only required property is that objects which compare equal have the same hash value.

Due to Python's dynamic type system, all containers can contain heterogeneous values. They can also be arbitrarily nested.

Python contains lists, sets, and dictionaries comprehensions. They provide an elegant way to define lists and perform map/filter-like operations on them. Our semantics desugars these comprehensions into actual loops. An example transformation is provided in Listing 6.9

Listing 6.9: Example rewriting of the list comprehension `[f(x) for x in y if b(x)]`

```
fresh_l = []
for x in y:
  if b(x):
    fresh_l.append(x)
```

We do not detail the semantics of container's methods, since they do not hide subtleties and only perform the container update specified by the method.

Slice objects are used to define subsequences of containers. They are defined by a starting point, a stopping point, and a step, similarly to `range` objects. We do not detail the semantics of `slice` objects here. These objects are supported by our analyses.

6.3.8 Iterators

Two builtins related to iterators are used in Python: `iter` creates an iterator object from compatible objects, while `next` gives the next element of an iterator.

The semantics of `iter` is shown in Figure 6.42. We try calling the `__iter__` method of the object. If this is successful, we check that the result is an iterator, i.e., that it has a `__next__` method. Otherwise, if the object's class provides index-based access through the `__getitem__` method, we create a builtin iterator over this structure. This iterator will access the structure through increasing indexes until an `IndexError` or `StopIteration` exception is raised.

The semantics of next (Figure 6.43) consists in calling the `__next__` method of the object. The `StopIteration` exception is used to signal when there are no elements left in the iterator.

The semantics of `len` is shown in Figure 6.44. We start by calling the `__len__` method and pass its result to the `__index__` method. The result is returned if it is a non-negative integer. Otherwise, various exceptions may be raised.

6.3.9 super

The `super` class can be seen as a drop-in replacement designating a parent class of the class being defined. It is mainly called to access a method (i.e., `super().m(...)`). It dynamically searches for the method `m` in the parents of the class. We start by showing a classic use case of `super` when a class inherits from multiple other classes.

[9] `https://docs.python.org/3.8/reference/datamodel.html#object.__hash__`

$$\mathbb{E}_{cur}[\![\ \texttt{iter}(expr)\ ]\!](cur, e, h) \stackrel{\text{def}}{=} \quad \texttt{declaration builtin\_iter PyObject\_GetIter}$$
$$\begin{array}{l}
\quad \text{letb } (cur, e, h), @_e = \mathbb{E}[\![\ expr\ ]\!](f, e, h) \text{ in} \\
\quad \text{if } hasattr(type(@_e), \texttt{"__iter__"}) \text{ then} \\
\qquad \text{letb } (cur, e, h), @_{it} = \mathbb{E}[\![\ type(@_e).\texttt{__iter__}(@_e)\ ]\!](cur, e, h) \text{ in} \\
\qquad \text{if } hasattr(type(@_{it}), \texttt{"__next__"}) \text{ then return } @_{it} \\
\quad \text{if } hasattr(type(@_e), \texttt{"__getitem__"}) \text{ then} \\
\qquad \text{return } \mathbb{E}[\![\ iterator(@_e, 0)\ ]\!](cur, e, h) \\
\quad \text{return } \mathbb{S}[\![\ \texttt{raise TypeError}\ ]\!](cur, e, h), \bot
\end{array}$$

Figure 6.42: Semantics of `iter`

$$\mathbb{E}_{cur}[\![\ \texttt{next}(expr)\ ]\!](cur, e, h) \stackrel{\text{def}}{=} \quad \texttt{declaration builtin\_next}$$
$$\begin{array}{l}
\quad \text{letb } (cur, e, h), @_e = \mathbb{E}[\![\ expr\ ]\!](cur, e, h) \text{ in} \\
\quad \text{if } hasattr(type(@_e), \texttt{"__next__"}) \text{ then} \\
\qquad \text{return } \mathbb{E}[\![\ type(@_e).\texttt{__next__}(@_e)\ ]\!] \\
\quad \text{return } \mathbb{S}[\![\ \texttt{raise TypeError}\ ]\!](cur, e, h), \bot
\end{array}$$

Figure 6.43: Semantics of `next`

$$\mathbb{E}_{cur}[\![\ \texttt{len}(expr)\ ]\!](cur, e, h) \stackrel{\text{def}}{=} \quad \texttt{declaration builtin\_len slot\_sq\_length}$$
$$\begin{array}{l}
\quad \text{letb } (cur, e, h), @_e = \mathbb{E}[\![\ expr\ ]\!](cur, e, h) \text{ in} \\
\quad \text{if } hasattr(type(@_e), \texttt{"__len__"}) \text{ then} \\
\qquad \text{letb } (cur, e, h), @_r = \mathbb{E}[\![\ type(@_e).\texttt{__len__}(@_e)\ ]\!](f, e, h) \text{ in} \\
\qquad \text{if } hasattr(type(@_r), \texttt{"__index__"}) \text{ then} \\
\qquad\quad \text{letb } (cur, e, h), @_i = \mathbb{E}[\![\ type(@_r).\texttt{__index__}(@_r)\ ]\!](f, e, h) \text{ in} \\
\qquad\quad \text{if } \neg isinstance(@_i, int) \text{ then return } \mathbb{S}[\![\ \texttt{raise TypeError}\ ]\!](cur, e, h) \\
\qquad\quad \text{let } \texttt{int}(v_i) = \ \text{fst} \circ h(@_i) \text{ in} \\
\qquad\quad \text{if } v_i < 0 \text{ then return } \mathbb{S}[\![\ \texttt{raise ValueError}\ ]\!](cur, e, h) \\
\qquad\quad \text{else return } (cur, e, h), @_i \\
\quad \text{return } \mathbb{S}[\![\ \texttt{raise TypeError}\ ]\!](cur, e, h), \bot
\end{array}$$

Figure 6.44: Semantics of `len` builtin

Example 6.34 **`super` and multiple inheritance**

Let us assume we have a `Pen` class, itself parent of two classes `BluePen` and `WipeablePen`, as described in lines 1-13 of Listing 6.10. We would like to create a class `WhiteboardPen`, inheriting from `BluePen` and `WipeablePen`. If we try to explicitly call the initialization method of `BluePen` and `WipeablePen`, these methods will both call `Pen.__init__`, doubling the side effects. If we use the `super` class, we have the double benefit of not having to make explicit initialization calls, and avoiding the double call to `Pen.__init__`.

In our example, the instantiation of `WhiteboardPen` will call in order:

- `BluePen.__init__(self)`,
- `WipeablePen.__init__(self)`,
- `Pen.__init__(self)`.

The printing is done after the calls to `super`, so it appears in the reverse order (in the trace shown lines 21-24).

Listing 6.10: Example usecase of `super` with multiple inheritance

```
1  class Pen:
2      def __init__(self):
3          print("- it's a pen")
4
5  class BluePen(Pen):
6      def __init__(self):
7          super().__init__()
8          print("- it's blue")
9
10 class WipeablePen(Pen):
11     def __init__(self):
12         super().__init__()
13         print("- it's wipeable")
14
15 class WhiteboardPen(BluePen, WipeablePen):
16     def __init__(self):
17         print("WhiteboardPen creation:")
18         super().__init__()
19
20 WhiteboardPen()
21 # WhiteboardPen creation:
22 # - it's a pen
23 # - it's wipeable
24 # - it's blue
```

Remark 6.35 **Conversion from implicit `super` to explicit form**

In our example, `super` was called without any argument at line 18. It was however able to perform the calls to `BluePen.__init__(self)`, `WipeablePen.__init__(self)`, and `Pen.__init__(self)`. It turns out this is made possible thanks to a hack in the definition the initilization function `super_init`, where arguments are filled with the method's class and the first local variable of the current stack frame (i.e., the `self` parameter).

To transform these implicit calls to `super`, we perform a rewriting into `super(cls, self)`, where `cls` is the class being defined and `self` the first argument of the method being defined. In our example, this mean the call at line 18 would have been rewritten into:

```
super(WhiteboardPen, self).__init__(self)
```

Remark 6.36 **Variations of `super` calls**

As we have said previously, the first argument of `super` is, by default, the class being defined. It can be overridden by a parent of the class. In that case, the methods being called will only be the one defined after the provided class in the MRO of the class being defined. For example, the MRO of `WhiteboardPen` is:

```
(WhiteboardPen, BluePen, WipeablePen, Pen, object)
```

If we replace the call at line 18 with `super(BluePen, self).__init__()`, only two initializers are called: `WipeablePen.__init__` and `Pen.__init__`.

Once this rewriting is performed, we can rely on the pure-Python implementation of super shown in Listing 6.11. This implementation has been adapted from the Python documentation[10]. `__getattr__` is the method used to resolve the attribute accesses after calls to `super` (such as the accesses to `__init__` in our example). It traverses the MRO for the class from which the start for the attribute will start. Then, it searches for the provided attribute in the MRO (after the starting class). `cls.__dict__[name]` corresponds to a low-level field access. Once the search is successful, the method tries to call the `__get__` method of the result, or returns it raw if this method does not exist.

Listing 6.11: Pure-Python implementation of `super`

```
1  class super:
2    def __init__(self, typ, obj):
3      if not isinstance(typ, type): raise TypeError
4
5      if isinstance(obj, type) and issubclass(obj, typ):
6        self.__self_class__ = obj
7      elif issubclass(type(obj), typ):
8        self.__self_class__ = type(obj)
9      else: raise TypeError
10
11     self.__thisclass__ = typ
12     self.__self__ = obj
13
14   def __getattr__(self, name):
15     starttype = self.__self_class__
16     mro = starttype.__mro__
17     for i in range(len(mro)):
18       if mro[i] is self.__thisclass__: break
19     i += 1
20     if i < len(mro):
21       for cls in mro[i:]:
22         if name in cls.__dict__:
23           res = cls.__dict__[name]
24           if hasattr(type(res), '__get__'):
25             res = type(res).__get__(res, None if self.__self__ is starttype
26                                     else self.__self__, starttype)
27           return res
28     return object.__getattribute__(self, name)
29
30   def __get__(self, obj, typ):
31     if obj is None or self.__self__ is not None: return self
32     if type(self) is not super: return type(self)(self.__thisclass__, obj, None)
33     else: return super(typ, obj)
```

6.3.10 Generators

Generators are a kind of coroutines that can be defined in Python. When a generator is called, it will execute itself until a `yield` statement is encountered. The control flow is then passed back to the caller until the generator is queried again.

Generators can be used as an alternative to lists. Instead of using offline algorithms, which require the whole data to be stored in memory, generators allow an online, elementwise processing, reducing memory usage. A typical example is for the processing of large files, where a first function will load the file as a list of strings, each representing a line, and pass this list

[10] https://www.python.org/download/releases/2.2/descrintro/#superexample

to other processing functions. This might consume a lot of memory depending on the file's length, since the whole file needs to be loaded. With generators, it is possible to define the file opening and file processing functions such that each line will be fully processed one after the other.

Example 6.37 Counting occurrences of 'a' in a file line by line using generators

A program using generators is shown in Listing 6.12. It reads the file called `in.txt` line by line, using the function `process_file`. For each line, the function `count` displays its content, and counts the number of occurrences of the character 'a'.

We dive a bit more into the execution details of this program. Let us assume the file `in.txt` contains two lines, the first with the string "abstract" and the second with the string "interpretation". `process_file` is called at line 17. Since its body contains the generator-specific `yield` statement, the function is directly interrupted. Similarly, `count` is called and stopped immediately, as it is a generator. The `for` loop will now resume the execution of the functions (during the calls to `next`, following the rewriting shown in Listing 6.1).

At the first turn of the loop, we resume the execution of `count`. `count` itself uses a loop to resume the execution of `process_file`, which opens the file and reads the first line. The generator-specific statement `yield l.strip()` stops the current execution of `process_file` at the end of line 6 and returns the string "abstract". This returned value is stored in `s` (line 10), the string is displayed and the number of occurrences counted. We then reach the `yield count` statement at line 15, which interrupts the execution of `count`, and returns the integer 2. This value is stored in `cnt` (line 18), and the count is displayed.

The second turn of the loop is similar. We restart the execution of `count`, itself restarting `process_file`. This function was stopped at the end of the `yield` at line 6. We increment `i` from 0 to 1, and start again the execution of the loop lines 4-7. The second line is read and the execution is stopped again by `yield` at line 6. The returned value is "interpretation", which is stored in the variable `s` of `count`. `count` displays the string, counts the occurrences of "a" in `s`, and yields the integer 1 to the loop at lines 18 and 19. This loop displays the last count.

At the third turn of the loop, the execution of `process_file` raises a `StopIteration` exception, since the traversal of the file is finished. This stops the execution of the program.

Even if the `print` calls (at lines 5, 12 and 19) are at three different program locations, they will be executed successively, thanks to the `yield` statements stopping the execution of each function. Using lists, the same approach would have required to either define the whole processing as a single, bulky function, or to keep the function's results in different lists to perform the display in a single pass at the end.

Listing 6.12: Counting occurrences of 'a' in a file line by line using generators

```
1  def process_file(fi):
2    with open(fi, 'r') as f:
3      i = 0
4      for l in f:
5        print(f"Reading line {i}", end=", ")
6        yield l.strip()
7        i += 1
8
9  def count(gen, c):
10   for s in gen:
11     count = 0
12     print(f"s is {s}", end=", ")
13     for ch in s:
14       if ch == c: count += 1
15     yield count
```

```
16
17 g = count(process_file("in.txt"), 'a')
18 for cnt in g:
19   print(f"got {cnt} 'a'")
20
21 # Provided in.txt is:
22 # abstract
23 # interpretation
24 # The result of this program is:
25 # Reading line 0, s is abstract, got 2 'a'
26 # Reading line 1, s is interpretation, got 1 'a'
```

This first example showed that generators can interrupt a function's execution and communicate some data back to the caller. It is also possible to introduce communication from the caller to the generator using the `send` method of generators.

Example 6.38 **Bidirectional communication with generators**

The bidirectional communication of generators can be used to change which character's occurrences are counted at each line. We use the `send` method over generators to send the character on which to count the occurrences in the `count` function. Bidirectional generators need to be initialized by sending `None`.

Let us assume that at the first turn of the while loop lines 22-28, the picked character is "c". This value is sent to the generator, which starts its execution at line 12. The result of the function is 1, which is passed back through the yield at line 18.

Then, let us assume that the second turn of the loop picks the character "a". This value is sent to the generator, which resumes its execution at the assignment line 18 (i.e., `c = "a"` is performed). The result of the function is 1 again, and passed back through the yield at line 18.

At the next turn, the `StopIteration` exception raised by `process_file` will stop the while loop.

Listing 6.13: Generator example with bidirectional communication

```
1  import random
2
3  def process_file(fi):
4    with open(fi, 'r') as f:
5      i = 0
6      for l in f:
7        print(f"Reading line {i}", end=", ")
8        yield l.strip()
9        i += 1
10
11 def count(gen):
12   c = yield None
13   for s in gen:
14     count = 0
15     print(f"s is {s}", end=", ")
16     for ch in s:
17       if ch == c: count += 1
18     c = yield count
19
20 g = count(process_file("in.txt"))
21 g.send(None)
22 while True:
23   try:
24     c = random.choice(['a', 'b', 'c'])
25     cnt = g.send(c)
```

```
    print(f"got {cnt} '{c}'")
  except StopIteration:
    break

# Example result:
# Reading line 0, s is abstract, got 1 'c'
# Reading line 1, s is interpretation, got 1 'a'
```

We now define the semantics of generators. We extend nominal objects **ObjN** to handle generator objects. These objects are written $\mathsf{Gen}(name, locals, body)$, and are defined by their name, the local variables used in its body, and the body of the generator. We extend flow tokens with two new kinds:

- $\mathit{yield}(@_{gen} \in \mathsf{Addr}, @_e \in \mathsf{Addr}^\bot, l \in \mathsf{Loc} \cup \{\mathit{end}\})$, representing generators allocated at address $@_{gen}$, whose execution is currently stopped at program location l (or is now stopped), potentially yielding the object allocated at $@_e$.
- $\mathit{next}(@_{gen} \in \mathsf{Addr}, @_s \in \mathsf{Addr}^\bot, l \in \mathsf{Loc} \cup \{\mathit{start}\})$ representing a call to a generator at $@_{gen}$, which should resume its execution at program location l (and ignore everything before). It is optionally passed an object that can be sent.

When a function f containing `yield` statements in its body is called, a new generator object is created. A specific flow token to start this generator is created. The semantics is shown in Figure 6.45.

$$
\begin{array}{l}
\mathbb{E}_{\mathsf{cur}}[\![\,\texttt{generator.\_\_call\_\_}(f, e_1, \ldots, e_n)\,]\!](\mathit{cur}, e, h) \overset{\text{def}}{=} \\
\quad \text{letb } (\mathit{cur}, e, h), @_f = \mathbb{E}[\![\, f \,]\!](\mathit{cur}, e, h) \text{ in} \\
\quad \text{let } \mathsf{Fun}(\mathit{fname}, (\mathit{arg}_1, \ldots, \mathit{arg}_m), (\mathit{local}_1, \ldots, \mathit{local}_i), \mathit{body}) = \mathsf{fst} \circ h(@_f) \text{ in} \\
\quad \text{if } m \neq n \text{ then return } \mathbb{S}[\![\,\texttt{raise TypeError}\,]\!](f, e, h) \\
\quad \text{letb } \sigma = \mathbb{S}[\![\, \mathit{local}_i = \texttt{LocalUndef} \,]\!] \circ \ldots \circ \mathbb{S}[\![\, \mathit{local}_1 = \texttt{LocalUndef} \,]\!] \\
\qquad\qquad \circ \mathbb{S}[\![\, \mathit{arg}_n = e_n \,]\!] \circ \ldots \circ \mathbb{S}[\![\, \mathit{arg}_1 = e_1 \,]\!] \sigma \text{ in} \\
\quad \text{letb } (f, e, h), @_g = \mathbb{E}[\![\, \mathit{alloc\_addr} \,]\!] \sigma \text{ in} \\
\quad \text{let } h = h[@_g \mapsto \mathsf{Gen}(\mathit{fname}, (\mathit{local}_1, \ldots, \mathit{local}_i), \mathit{body}), \emptyset] \text{ in} \\
\quad \{\, ((\mathit{cur}, e, h), @_g); ((\mathit{next}(@_g, \bot, \{\, \mathit{start} \,\}), e, h), @_g) \,\}
\end{array}
$$

Figure 6.45: Semantics of generator creation, assuming f is a function containing a `yield` statement

The `yield` expression is used to interrupt the execution of the generator and get back to the execution of the caller. The semantics is shown in Figure 6.46, and is defined in two cases:

1. If the execution is normal, tagged by the *cur* flow token, `yield` evaluates its argument and stops the normal execution. Instead, it returns two states. The first state is tagged by *yield*(), which will be caught by `generator.send` to extract e's value. The second state is tagged by *next*. It will be used to resume the generator at the correct location (through the second definition of `yield` described in our second point). In both cases, we use the generator's address in the flow tokens. We assume this address has been stored globally.
2. The second case is used to resume a generator's execution. When a generator is resumed, we start at the beginning of the generator's body, but in a state tagged by *next*, meaning that the semantics will not execute anything until the correct `yield` statement is reached. After that, the execution is resumed by shifting the flow token from *next* to *cur*. If something was sent, it is the result of this evaluation.

$$
\begin{array}{l}
\mathbb{E}[\![\, \texttt{yield}_l\ expr \,]\!](cur, e, h) \stackrel{\text{def}}{=} \\
\quad \text{letb } (cur, e, h), @_e = \mathbb{E}[\![\, expr \,]\!](cur, e, h) \text{ in} \\
\quad \text{return } \{\, (\mathit{yield}(@_g, @_e, l), e, h), \bot; (\mathit{next}@_g, \bot, l, e, h), \bot \,\} \\
\mathbb{E}[\![\, \texttt{yield}_l\ expr \,]\!](\mathit{next}(@_g, l, @_s), e, h) \stackrel{\text{def}}{=} \\
\quad (cur, e, h), @_s
\end{array}
$$

Figure 6.46: Semantics of the `yield` expression

Generators support the iterator protocol. That is, the execution of generators is resumed by calling the `next` builtin. This builtin calls the `next` method of generators, which is a simplified value of the `send` method. We show both of their semantics in Figure 6.47. We start by evaluating the arguments in the current flow token. Then, we partition the input state Σ to keep states tagged by the *next* flow token. Other states are kept in Σ_{other}. We execute the *body* of the generator on the states tagged by *next* flow token (although if the generator was just created, we replace the token with *cur* to start at the beginning of the body), and call Σ_{out} the result. The result of the method consists in: (i) the other states in Σ_{other} that are passed on, (ii) the states of Σ_{out} which are not tagged by a *yield*() token (this includes states tagged by an *exn* of *next*), (iii) the states of Σ_{out} tagged by a *yield*() token (when the generator is not finished): we return the result of the yield in the token, and return a state tagged by *cur*, (iv) erroneous states where a `StopIteration` exception has been raised if the generator's end has been reached.

$$
\begin{array}{l}
\mathbb{E}[\![\, \texttt{generator.send}(gen, send) \,]\!]\Sigma \stackrel{\text{def}}{=} \\
\quad \text{let } \Sigma_{cur} = \{\, (f, e, h) \in \Sigma \mid f = cur \,\} \text{ in} \\
\quad \text{letb } (f, e, h), @_g = \mathbb{E}[\![\, gen \,]\!]\Sigma_{cur} \text{ in} \\
\quad \text{let } \mathsf{Gen}(name, locals, body) = h(@_g) \text{ in} \\
\quad \text{letb } (f, e, h), @_s = \mathbb{E}[\![\, send \,]\!](f, e, h) \text{ in} \\
\quad \text{let } \Sigma_{next,start} = \{\, (cur, e, h) \mid (f, e, h) \in \Sigma, f = \mathit{next}(@_g, \_, start) \,\} \text{ in} \\
\quad \text{let } \Sigma_{next,other} = \{\, (\mathit{next}(@_g, @_s, l), e, h) \in \Sigma \mid f = \mathit{next}(@_g, \_, l), l \neq start \,\} \text{ in} \\
\quad \text{let } \Sigma_{other} = \Sigma \setminus (\Sigma_{cur} \cup \Sigma_{next}) \text{ in} \\
\quad \text{let } \Sigma_{out} = \mathbb{S}[\![\, body \,]\!](\Sigma_{next,start} \cup \Sigma_{next,other}) \text{ in} \\
\quad \{\, (\sigma_o, \bot) \mid \sigma_o \in \Sigma_{other} \,\} \\
\quad \cup \{\, (f, e, h), \bot \mid (f, e, h) \in \Sigma_{out}, f \neq \mathit{yield}(\_, \_, \_) \,\} \\
\quad \cup \{\, (cur, e, h), @ \mid (f, e, h) \in \Sigma_{out}, f = \mathit{yield}(@_g, @, l), l \neq end \,\} \\
\quad \cup \{\, \mathbb{S}[\![\, \texttt{raise StopIteration}(@) \,]\!](cur, e, h), \bot \mid (f, e, h) \in \Sigma_{out}, f = \mathit{yield}(@_g, @, end) \,\} \\
\mathbb{E}[\![\, \texttt{generator.\_\_next\_\_}(gen) \,]\!] \stackrel{\text{def}}{=} \mathbb{E}[\![\, \texttt{generator.send}(gen, \texttt{None}) \,]\!]
\end{array}
$$

Figure 6.47: Semantics of calls to generators

6.4 Correctness

The semantics has been established by reading the Python documentation, CPython's source code, and through various experimentations. We strived to make it the most complete possible,

but this is a difficult and time-consuming task.

6.4.1 Tests from previous works

Some previous works on the semantics of Python [141, 125, 66] (detailed in Section 6.6) included the manual creation of semantics tests. Guth [66] collected (and updated) the tests of previous works, which are published alongside their semantics on Github[11]. The tests of Guth [66] rely on assertions. The tests of Smeding [141] and Politz et al. [125] relied on custom drivers[12,13] used to dynamically execute test files using a specific namespace. We modified those files to inline the functions, making them less dynamic and more explicit.

We show the results of our static analyzer on those tests in Figure 6.48. Mopsa is an abstract interpreter. This approach tests the abstract semantics, which is close to but not exactly the concrete semantics defined in this chapter. In particular, Mopsa can be imprecise in some cases, which may be why some tests fail. To match closely the concrete, loops are fully unrolled. An example source of imprecision in the abstract interpreter is the list abstraction. Lists are abstracted into a single, weak variable through a summarization for example. We can notice that Mopsa supports a majority of the tests, although the number of false alarms is still high.

We were unable to run by ourselves the semantics of Guth [66] but they claim to pass 83% of Politz et al.'s tests, and almost all of Smeding [141]' tests. Köhl [88] was not able to run the semantics of Guth [66] either[14]. We were able to run the artifact of Politz et al. [125]: it passes 69.8% tests for Smeding's test suite, 44.2% of Guth's tests, and 96.2% of their own tests. Köhl [88] is able to pass 65.7% of Guth's tests, and 59.6% of the tests of Politz et al. Smeding [141] defined a semantics of Python 2, while we focus on Python 3, which contains major, non-compatible changes. Thus, we did not try comparing it. The test suite has been updated by Guth so that their testcases focus on Python 3. The other prototype analyzer by abstract interpretation of Python programs by Fromherz et al. [54] has comparable running times. However, it supports fewer tests, as shown in Figure 6.49.

Due to its abstractions, Mopsa may raise false alarms and thus supports fewer tests for now. These tests have helped us choose which features should be implemented. Supporting more of these tests in Mopsa is ongoing work. One of the main limitations to address to pass more tests is the support of the argument unpacking operators $*$ and $**$ (cf. Section 6.2.10).

Test suite	Number of tests	LOC	Time	Passing	False alarm	Unsupported
Smeding [141]	129	1247	2.8s	66.67%	26.36%	6.98%
Politz et al. [125]	215	3602	5.5s	36.74%	29.3%	33.95%
Guth [66]	242	2524	5.6s	42.56%	35.12%	22.31%

Figure 6.48: Mopsa's analysis of semantic tests

Test suite	Number of tests	Passing	False alarm	Unsupported
Smeding [141]	129	40.31%	23.14%	36.55%
Politz et al. [125]	215	23.72%	33.49%	56.21%
Guth [66]	242	19.83%	23.14%	41.97%

Figure 6.49: Running the semantic tests using the analyzer of Fromherz et al. [54]

[11] `https://github.com/kframework/python-semantics/tree/master/programs`
[12] `https://github.com/kframework/python-semantics/blob/master/programs/smeding/driver/driver.py`
[13] `https://github.com/kframework/python-semantics/blob/master/programs/politz/driver/driver.py`
[14] `https://github.com/kframework/python-semantics/issues/1`

6.4.2 CPython's tests

Since the previous tests were handcrafted by non-Python developers, we do not know if they test all core Python features. Thus, we also test Mopsa on a subset of the official tests of the reference interpreter. These tests use the `unittest` library of Python to run, which is supported by Mopsa. We chose the tests that focused on the language and the builtins. The results are shown in Figure 6.50. For each test file (clickable link to source), we show its length (measured using `cloc`), the number of supported tests over the total number of tests, the time taken to analyze those tests, and the number of assertions that Mopsa is able to prove over the total number of assertions (defined in the supported tests). The artifact of Politz et al. [125] does not appear to support this `unittest` framework correctly. The analyzer of Fromherz et al. [54] does not support `unittest` either. The implementation of Köhl [88] does not support the `import` statement for now, which is used to import the `unittest` library. We can notice that a majority of the tests in this subset are supported by Mopsa. However, the abstractions of lists and dictionaries are not precise to prove many assertions.

Name	LOC	# tests	Time	# assertions
augassign	230	6/7	0.22s	9/16
baseexception	83	9/10	68ms	1/1
contains	77	4/4	76ms	3/5
decorators	95	6/13	61ms	3/4
dict	684	48/76	1.3s	16/78
exception_variations	140	11/11	69ms	25/25
index	199	18/20	0.51s	14/21
int	349	10/17	0.27s	15/35
int_literal	91	6/6	0.15s	81/81
isinstance	98	9/18	0.13s	11/12
iter	482	47/55	1.1s	12/38
list	686	48/61	3.5s	34/158
operator	303	36/42	0.85s	118/130
raise	364	33/35	0.16s	2/4
range	450	19/24	0.58s	23/55
richcmp	205	9/11	7.1s	4/19
typechecks	49	6/6	65ms	18/20

Figure 6.50: Mopsa's analysis of CPython's tests

6.4.3 Summary of the conformance tests

Mopsa supports a reasonable number of tests, coming from different sources. Supporting more of these tests is ongoing work. However, this is not our top priority: our goal is to analyze Python programs rather than replicate the concrete Python interpreter. An interesting future work would be the automatic generation of test cases for each case of the semantics, and the comparison of these test cases with CPython.

6.5 Comparison with JavaScript

According to GitHub [62], JavaScript is more popular than Python and also a dynamic programming language. This section highlights a few differences between the languages.

The first significant difference is that JavaScript is a standardized language through what is called ECMAScript. The semantics are thus defined through text that aims to be as precise

as possible. We can for example compare the semantics of the addition in JavaScript[15], with the documentation provided in Python[16]. A considerable part of the work on Python is first to grasp the semantics. ECMAScript also comes with a conformance test suite, which has no Python equivalent.

JavaScript has two special values: `undefined`, used for uninitialized variables, and `null`. The only numerical datatype available was floating-point numbers until the introduction of arbitrary-precision integers in ECMAScript 2020. The last three primitive values used by JavaScript are booleans, strings, and symbols (introduced in ECMAScript 2015, symbols are used to generate unique objects). The only two other types are function and object, encompassing all other objects.

From a typing perspective, Python favors raising exceptions during unsupported operations (`"1 hello" + 2` raises a `TypeError`), while JavaScript favors silent coercions (it evaluates `"1 hello" + 2` into `"1 hello2"`).

JavaScript historically uses a structural type system, where objects are only maps from strings to other objects.[17] A prototype mechanism defines which fields will be accessible to all objects having the same prototype. The inheritance mechanism also relies on it to inherit all attributes of the parent. The syntactic declaration of classes was introduced in 2015. It is syntactic sugar for prototype declarations. We can compare Python's type and inheritance mechanism, reminded in Figure 6.52 with the one of JavaScript in Figure 6.51. Python attributes are usually called fields in JavaScript.

A significant difference with Python is that JavaScript does not allow overloading most operators (such as binary operators or calls). The only exception is for fields, which can have custom getters and setters. Since ECMAScript 6, proxies can also be used to override field accesses over an object globally, similarly to a redefinition of the `__getattribute__` method in Python.

Similar to Python, generators and asynchronous operators are available. As a general rule, `eval` seems to be more used in JavaScript. To the best of our knowledge, ECMAScript does not describe how garbage collection should be performed.

6.6 Related work

Ranson et al. [127] define a mechanized semantics for a restricted subset of Python, consisting of basic values (integers, booleans) and control structures (loops, conditionals). A significant shortcoming of this work is the absence of formalization of objects. Smeding [141] proposes a semantics of Python 2.5 under the form of a Haskell interpreter. Politz et al. [125] define a small-step semantics for a core Python language, λ_π, as well as a compiler from Python to λ_π, and a λ_π interpreter written in Racket. Bodin et al. [16] define a mechanized semantics for ECMAScript 5. The K framework [133] allows defining semantics, on top of which interpreters and model-checkers can be automatically derived. It has notably been used to formalize the semantics of ECMAScript 5.1 [123], and Python 3.2 [66]. Köhl [88] recently published a Master's thesis, defining a new framework for the definition of structural operational semantics, and applying it to define the semantics of Python.

Fromherz et al. [54] define an interpreter-like semantics on which the concrete semantics of this chapter is based. Our version is significantly different: it is more complete (as shows Figure 6.50, compared to Table 1 of [54]), we have added references to the source code in the semantics, and we have separated the semantics of the language from its builtins (e.g., we have separated the semantics of `x.s` from `object.__getattribute__`). Both Fromherz et al. [54]'s work and ours aim at defining abstract interpreters, which can compute an approximation

[15] `https://262.ecma-international.org/10.0/#sec-addition-operator-plus`
[16] `https://docs.python.org/3/reference/datamodel.html#object.__add__`
[17] Special symbol objects can also be used as keys.

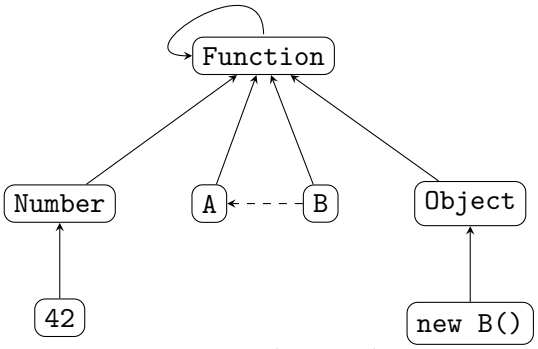

Figure 6.51: JavaScript type and subclass relations
Assuming A is a class and B inherits from A.

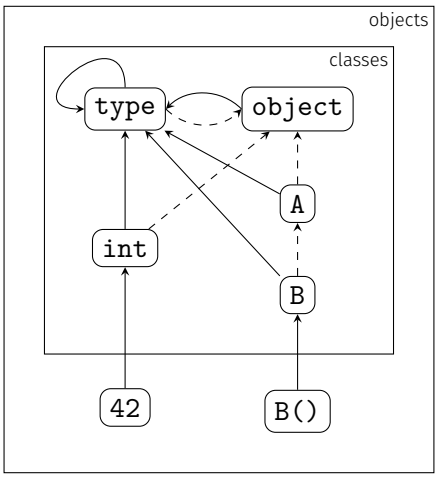

Figure 6.52: Python type and subclass relations
Assuming A is a class and B inherits from A.
Reminder of Figure 6.1

of the concrete semantics of Python. Contrary to [133, 66, 141, 125, 88], defining the concrete semantics in itself is not our endgoal, it is only defined to be used as a basis to define the abstract semantics. The shape of our concrete semantics is also influenced by the use we make of it in the abstract analysis, because the abstract semantics will have a similar shape and stay close to it structurally.

6.7 Conclusion

This chapter presented the semantics of Python. We defined the semantics by induction over the syntax of Python as a function manipulating reachable program states. This way, the abstract semantics will have a shape similar to the concrete one.

The semantics is complex, with lots of cases and multiple stages (operator level, builtin level, and user level). It relies on introspection operators of both type systems (`isinstance` and `type` for the nominal type system, `hasattr` for the structural type system). For now, it does

not handle asynchronous operator, the `eval` & `exec` statements, and the calls to finalizers made by the garbage collector.

We ran our abstract interpreter against various tests distributions to evaluate how complete our semantics is. Mopsa supports a reasonable amount of tests, coming from different sources. Improving the number of tests passed is ongoing work. However, our endgoal is to analyze more Python programs. In general, we decide to support new cases of Python's semantics when we encounter a concrete need during the analysis of real-world programs.

7

Type Analysis

As we mentioned in Chapter 6, Python is an object-oriented language, using dynamic typing, and supporting introspection operators, self-modification of objects, and arbitrary code evaluation. These characteristics make the analysis of Python different from the analysis of static programming languages such as C or Java. Due to dynamic typing, type errors are detected at runtime and cause `TypeError` exceptions, whereas such errors would be caught at compile time in a statically typed language. We propose a static analysis to infer type information and use this information to detect automatically all exceptions that can be raised and not caught. Our type analysis is flow-sensitive, to take into account the fact that variable types evolve during program execution and, conversely, runtime type information is used to alter the control flow of the program, either through introspection or method and operator overloading. Moreover, it is context-sensitive as, without any type information on method entry, it is not possible to infer its control flow at all. However, handling `eval` is left as future work (possibly leveraging ideas proposed by Jensen et al. [77], Arceri and Mastroeni [3] on JavaScript). In some sense, we combine an exception analysis and a points-to analysis, following the work of Bravenboer and Smaragdakis [19].

We compare our approach with traditional type systems in Section 7.1. Our analysis is described in Section 7.2. We show a motivating example in the next paragraph. Our analysis has special domains to support containers (such as lists) and infer type equalities (allowing it to express parametric polymorphism, Section 7.3). The analysis is soundly derived by abstract interpretation from a concrete semantics of Python. It has been implemented into Mopsa (Section 7.5), and leverages external type annotations from the Typeshed project to support the vast standard library (Section 7.4). We show in Section 7.6 that it scales to benchmarks a few thousand lines long, and preliminary results show it is able to analyze a small real-life command-line utility called PathPicker. Compared to previous work (Section 7.7), it is sound, while it keeps similar efficiency and precision.

Listing 7.1: Python programs relying on both typing mechanisms

```
def fspath(p):
  sb = (str, bytes)
  if isinstance(p, sb):
    return p
  if hasattr(p, "fspath"):
    r = p.fspath()
    if isinstance(r, sb):
      return r
    else: raise TypeError
  else: raise TypeError
```

```
11
12 class Path:
13   def fspath(self):
14     return 42
15
16 if rand(0, 1): p = "/dev"
17 else: p = Path()
18 r = fspath(p)
```

Example 7.1 **Motivating Example**

Consider the code from Listing 7.1. It defines a function `fspath`, taken from the standard library, with a parameter `p` as input. If `p` is an object instantiated from a class inheriting from `str` or `bytes`, it is returned. Otherwise, the function searches for an attribute called `fspath` and calls it as a method. If the return type is not an instance of `str` or `bytes`, an exception is raised. Thus, when `fspath` does not raise an error, it takes as input an instance of `str` or `bytes`, or an object having a method `fspath` returning either a string or a bytes-based string. In all cases, the return type of `fspath` is either `str` or `bytes`. This function uses both of Python's type systems: it uses the nominal one when calling the *isinstance* operator, and the structural one with *hasattr*.

We model correct and erroneous calls to `fspath` in lines 12 to 18. In particular, we define a `Path` class, having a method `fspath` returning an integer, hence, a call to function `fspath` on an instance of `Path` would raise a `TypeError`. Our analysis is able to infer that, at the end of line 16, `p` is a string, and that at line 17, `p` is an instance of the `Path` class, which has a field `fspath` that can be called. It finds that, either `r` is a string, or a `TypeError` is raised.

As it is part of the standard library, the `fspath` function is particularly well-behaved: it does not make many implicit assumptions on the argument type (only that `p.fspath` is callable) but instead uses type information to handle different cases in different program branches. Nevertheless, the context is important to infer whether a specific call to `fspath` can raise a `TypeError` or not. A more typical Python programmer might replace lines 5 to 10 with a call to `return p.fspath()`, leaving implicit the fact that `p` should have a method `fspath` returning `str` or `bytes` strings. This is summarized in one of Python motos: "easier to ask for forgiveness than permission". Our analysis would correctly infer that invalid calls to that modified function would raise an `AttributeError` exception.

7.1 Differences with a type system

It is worth comparing our approach with classic typing (the case of gradual typing is covered in Section 7.7). Statically typed languages ensure the absence of type-related errors through static type checking, possibly augmented with automatic type inference. However, while static typing rejects untypable programs, our analysis gives semantics to such programs by propagating type errors as exceptions. This is important in order to support programs that perform runtime type errors and catch them afterward, which is common-place in Python. Indeed, our goal is not to enforce a stricter, easier to check, way to program in Python,[1] but rather to check as-is programs with no uncaught exceptions. Secondly, static type checking is generally flow-insensitive and context-insensitive, trying to associate a unique type to each variable throughout program executions while we use a flow- and context-sensitive analysis. Our approach makes the analysis precise enough to handle introspection operators and dynamic addition of attributes – the latter changing the structural type of objects at runtime. Following classic abstract interpreters [11], our analysis is performed by structural induction on the syntax, starting from the program's entry point. This approach by structural induction has already

[1] In practice, we are nevertheless limited to programs that do not use `eval`.

been used in our introduction to abstract interpretation in Chapter 2. Our analysis is thus unable to analyze functions in isolation. While this kind of modularity is a highlight of typing algorithms, we believe that it is not well suited to Python. Consider, for instance, that a call to a function can alter the value, and so the type, of a global variable, which is difficult to express in a type system. Moreover, even a simple function, such as `def f(a, b): return a + b`, has an unpredictable effect as the `+` operator can be overloaded to an arbitrary method by the programmer (the semantics of `+` has been shown in the previous chapter, Figure 6.23). We view type analysis as an instance of abstract interpretation, one which is slightly more abstract than classic value analyses. This view is not novel, as Cousot [32] reconstructed Hindley-Milner typing rules as an abstract interpretation of the concrete semantics of the lambda calculus. One benefit of this unified view is the possibility to incorporate some amount of value analysis (presented in Chapter 8). For instance, our type analysis currently considers, to be sound, that any division can raise a `ZeroDivisionError`, which could be ruled out by a simple integer analysis. Finally, the correctness proof of our analysis is derived through a soundness theorem linking the concrete and the abstract semantics, in classic abstract interpretation form and not by subject reduction. Both our analysis and type systems are conservative, but we replace the motto "well-typed programs cannot go wrong" with a guaranteed overapproximation of the possible (correct and incorrect) behaviors of the program.

7.2 Non-relational type analysis

This section presents the type analysis. We present abstractions incrementally, starting from addresses to the environment and the heap, as well as additional stateless abstractions. A summary of the full abstract state and an example are provided to close this section.

7.2.1 Abstract addresses

As usual in our endeavor for modular definitions, we do not require addresses to have a specific structure. We only require that other domains can access the type of the object stored at a given address using `type` (we highlighted the need for abstract addresses to depend on types in Section 4.2 for Python analyses), and the mode of the address using `mode` (an address is *weak* if it summarizes multiple addresses in the concrete, and *strong* otherwise). The type is defined in $\mathsf{ObjN}^\sharp$. It only differs from ObjN by forgetting builtin values ($\texttt{int}(i \in \mathbb{Z}) \in \mathsf{ObjN}$ is abstracted by $\texttt{int}^\sharp \in \mathsf{ObjN}^\sharp$) except for booleans, and replacing addresses by their abstract counterparts.

$$\begin{aligned}
&\mathsf{ObjN}^\sharp \stackrel{\text{def}}{=} \texttt{int}^\sharp \cup \texttt{bool}^\sharp(b \in \{\texttt{True}, \texttt{False}, \top\}) \cup \texttt{float}^\sharp \cup \texttt{str}^\sharp \cup \texttt{NoneType}^\sharp \cup \texttt{NotImplType}^\sharp \\
&\quad \cup \texttt{List}^\sharp \cup \texttt{Tuple}^\sharp \cup \texttt{meth}(a, f) \cup \texttt{cls}(c) \cup \texttt{fun}(f) \cup \texttt{inst}(a), a \in \mathsf{Addr}^\sharp \\
&\texttt{type} \in \mathsf{Addr}^\sharp \to \mathsf{ObjN}^\sharp \\
&\texttt{mode} \in \mathsf{Addr}^\sharp \to \{\, \mathit{weak}, \mathit{strong} \,\}
\end{aligned}$$

We define a concretization from abstract nominal objects $\mathsf{ObjN}^\sharp$ to concrete nominal objects relying on abstract addresses $\mathsf{ObjN}_{\mathsf{Addr}^\sharp}$ in Figure 7.1.

7.2.2 Environment abstraction

Domain. The domain of abstract environments $\mathcal{E}^\sharp$ maintains a non-relational map binding variables to a set of abstract addresses. In the concrete, variables are program identifiers (Id). In the abstract, these variables are complemented with auxiliary variables created by other abstractions, such as the attribute abstraction presented in Section 7.2.3. Thus, $\mathsf{Id} \subseteq \mathcal{V}$. To

$$\gamma_{\mathsf{ObjN}} : \begin{cases} \mathsf{ObjN}^\sharp & \to \mathcal{P}(\mathsf{ObjN}^\sharp_{\mathsf{Addr}}) \\ \texttt{int}^\sharp & \mapsto \{\, \texttt{int}(i) \mid i \in \mathbb{Z} \,\} \\ \texttt{bool}^\sharp(b \in \{\texttt{True}, \texttt{False}\}) & \mapsto \{\, \texttt{bool}(b) \,\} \\ \texttt{bool}^\sharp(\top) & \mapsto \{\, \texttt{bool}(b) \mid b \in \texttt{True}, \texttt{False} \,\} \\ \texttt{float}^\sharp & \mapsto \{\, \texttt{float}(f) \mid f \in \mathsf{F}_{64} \,\} \\ \texttt{str}^\sharp & \mapsto \{\, \texttt{str}(s) \mid s \in \mathit{string} \,\} \\ \texttt{NoneType}^\sharp & \mapsto \{\, \texttt{None} \,\} \\ \texttt{NotImplType}^\sharp & \mapsto \{\, \texttt{NotImpl} \,\} \\ \texttt{List}^\sharp & \mapsto \bigcup_{n\in\mathbb{N}} \{\, \mathsf{List}(@_1^\sharp, \ldots, @_n^\sharp) \mid @_i^\sharp \in \mathsf{Addr}^\sharp \,\} \\ \texttt{Tuple}^\sharp & \mapsto \bigcup_{n\in\mathbb{N}} \{\, \mathsf{Tuple}(@_1^\sharp, \ldots, @_n^\sharp) \mid @_i^\sharp \in \mathsf{Addr}^\sharp \,\} \\ \texttt{meth}(a, f) & \mapsto \{\, \mathsf{Method}(a, f) \,\} \\ \texttt{cls}(c) & \mapsto \{\, \mathsf{Class}(c) \,\} \\ \texttt{fun}(f) & \mapsto \{\, \mathsf{Fun}(f) \,\} \\ \texttt{inst}(a) & \mapsto \{\, \mathsf{Inst}(a) \,\} \end{cases}$$

Figure 7.1: Concretization of abstract nominal objects

support Python scoping, variables can also point to `LocalUndef` to represent variables that can be locally undefined.

$$\mathcal{E}^\sharp \stackrel{\text{def}}{=} \mathcal{V} \rightharpoonup \mathcal{P}(\mathsf{Addr}^\sharp \cup \{\, \texttt{LocalUndef} \,\})$$

Transfer functions (Figure 7.2). Following the modular definitions of Mopsa presented in Chapters 3 and 4, the transfer functions are defined as a manipulation of the global abstract state $\sigma^\sharp$. Getting and setting the state of the local abstraction are respectively done using `man.get` $\sigma^\sharp$ and `man.set` $s\,\sigma^\sharp$. To perform an assignment `x = e`, we evaluate `e` in the abstract, and update the local abstract state ϵ for x accordingly. The deletion of a binding updates the local abstract state of the environment in a similar fashion. The evaluation of `x` is disjunctive: if the variable points to multiple abstract addresses, each address is returned in a state where the environment has been refined.

Concretization (Figure 7.3). In the case of the type analysis, the environment abstraction is a leaf domain. We thus define it as a regular concretization, which is lifted to a relation from the unit type using the $\uparrow_{()}$ operator (defined in Remark 2.83). We will show an updated version of the environment and this concretization for the case of the value analysis in Chapter 8. The concretization of this domain creates an environment and a heap. The environment picks for each variable v of the abstract state ϵ an abstract address of the set in $\epsilon(v)$. For each address $@^\sharp$ used in the abstract environment ϵ, the first coordinate of the heap at the same address is one of the nominal objects abstracted by $\texttt{type}(@^\sharp)$ (the second coordinate will be constrained later, by concretizing the heap).

7.2.3 Heap Abstraction

The domain of abstract heaps $\mathcal{H}^\sharp$ defines which fields are defined for each Python objects. We defined in the concrete semantics (Chapter 6) the concept of fields, local to an object. Attributes are fields resolved transitively from an object through its parents' classes. We first describe the lattice of string powerset with underapproximation, which is used to abstract the fields of an object.

$$
\begin{array}{l}
\mathbb{S}^{\sharp}_{env}[\![\, x = e \,]\!]\sigma^{\sharp} \stackrel{\text{def}}{=} \\
\quad \text{letb } \sigma^{\sharp}, @^{\sharp} = \mathbb{E}^{\sharp}[\![\, e \,]\!]\sigma^{\sharp} \text{ in} \\
\quad \text{let } \epsilon = \texttt{man.get}\, \sigma^{\sharp} \text{ in} \\
\quad \text{let } \epsilon = \epsilon[x \mapsto \{\, @^{\sharp} \,\}] \text{ in} \\
\quad \texttt{man.set}\, \epsilon\, \sigma^{\sharp} \\
\mathbb{S}^{\sharp}_{env}[\![\, \texttt{del } x \,]\!]\sigma^{\sharp} \stackrel{\text{def}}{=} \\
\quad \text{let } \epsilon = \texttt{man.get}\, \sigma^{\sharp} \text{ in} \\
\quad \text{if } x \notin \text{dom } \epsilon \text{ then return } \mathbb{S}^{\sharp}[\![\, \texttt{raise NameError} \,]\!]\sigma^{\sharp} \text{ else} \\
\quad \text{if } \epsilon(x) = \texttt{LocalUndef} \text{ then return } \mathbb{S}^{\sharp}[\![\, \texttt{raise UnboundLocalError} \,]\!]\sigma^{\sharp} \text{ else} \\
\quad \text{return } \texttt{man.set}\, (\epsilon \setminus \{\, x \,\})\, \sigma^{\sharp} \\
\mathbb{E}^{\sharp}_{env}[\![\, x \in \mathcal{V} \,]\!]\sigma^{\sharp} \stackrel{\text{def}}{=} \\
\quad \text{let } \epsilon = \texttt{man.get}\, \sigma^{\sharp} \text{ in} \\
\quad \text{if } x \notin \text{dom } \epsilon \text{ then return } \mathbb{S}[\![\, \texttt{raise NameError} \,]\!]\sigma^{\sharp}, \bot \text{ else} \\
\quad \text{if } \epsilon(x) = \texttt{LocalUndef} \text{ then return } \mathbb{S}[\![\, \texttt{raise UnboundLocalError} \,]\!]\sigma^{\sharp}, \bot \text{ else} \\
\quad \text{return } \bigcup_{a \in \epsilon(id)} (\texttt{man.set}\, \epsilon[x \mapsto \{\, a \,\}]\, \sigma^{\sharp}, a)
\end{array}
$$

Figure 7.2: Transfer functions of the environment abstraction

$$
\begin{array}{rl}
\gamma_{env}(\epsilon \in \mathcal{V} \to \mathcal{P}(\mathsf{Addr}^{\sharp})) = \uparrow_{()} \circ \{\, (e, h) \in (\mathcal{V} \rightharpoonup \mathsf{Addr}^{\sharp}) \times (\mathsf{Addr}^{\sharp} \rightharpoonup \mathsf{ObjN} \times \mathsf{ObjS}) \mid & \\
\qquad v \in \text{dom}\epsilon \Leftrightarrow e(v) \in \epsilon(v); & \\
\qquad @^{\sharp} \in \text{codom}\epsilon \Leftrightarrow \text{fst} \circ h(@^{\sharp}) \in \gamma_{\mathsf{ObjN}}(\texttt{type}(@^{\sharp})) \,\} &
\end{array}
$$

Figure 7.3: Concretization of the environment abstraction

Powerset with underapproximation. The structural part $\mathsf{ObjS}^{\sharp}$, defined in Figure 7.4, keeps two sets of fields (i.e., strings). Fields may be added in some execution traces and not in others. Hence, we keep both an underapproximation and an overapproximation of the set of fields that must (respectively may) exist at a given program point for all possible executions. This information is important to avoid raising spurious `AttributeError` exceptions for fields that are definitely present. These properties are formally defined by the concretization γ_{ObjS}, in Figure 7.4. The structural type abstraction may also be approximated as $\top$ by the widening, in order to avoid having an infinite number of fields being added to an instance, which would break the analysis' termination. The special case of 🔒 designates object having a builtin read-only structure, similarly to the concrete. Two powersets with underapproximations (l_1, u_1) and (l_2, u_2) can easily be joined: $(l_1, u_1) \sqcup^{\sharp} (l_2, u_2) = (l_1 \cap l_2, u_1 \cup u_2)$.

Abstract domain. The abstraction is then a mapping from abstract addresses to the powerset we presented:

$$\mathcal{H}^{\sharp} \stackrel{\text{def}}{=} \mathsf{Addr}^{\sharp} \rightharpoonup \mathsf{ObjS}^{\sharp}$$

$$\mathsf{ObjS}^\sharp \stackrel{\text{def}}{=} \{ \text{🔒} \} \cup \{ (l,u) \mid l \in \mathcal{P}(), u \in \mathcal{P}(string) \cup \{ \top \}, l \subseteq u \vee u = \top \}$$

$$\gamma_{\mathsf{Objs}}((l,u)) = \{ x \in \mathcal{P}(string) \mid l \subseteq x \subseteq u \}$$
$$\gamma_{\mathsf{Objs}}(\text{🔒}) = \{ \text{🔒} \}$$

Figure 7.4: Concretization of $\mathsf{ObjS}^\sharp$

Transfer functions (Figure 7.5). The transfer functions handle the auxiliary field operators we introduced in the concrete semantics, and work by delegation through the use of auxiliary address variables. The transfer function of *get_field* first handles a special case for builtins, just as in the concrete (Figure 6.4). It searches for the local state η of the heap abstraction, and queries it at address $@^\sharp$ to get the fields defined through string sets (l,u). If the field is present in the lower, underapproximating part, we can just evaluate the auxiliary address variable $\underline{@^\sharp \cdot attr}$ corresponding to the field. This evaluation will be handled by the environment domain presented previously. Otherwise, if the field is only present in the overapproximating part, we return a case disjunction, where we assume that either the field must exist or that it does not exist at all. In the latter case, we return $\bot$ to signal an erroneous state, but no exception is raised by this low-level operator. This will be performed by the methods performing the attribute access, such as `object.__getattribute__` (described in Figure 6.31 in the concrete). The first case evaluates the same auxiliary variable and updates the local state so that the field is now in the underapproximation. The second case returns an erroneous evaluation where the field has been removed from the local state. *has_field* enjoys a definition extremely close to its concrete counterpart, where the work is delegated to *get_field*. *set_field* fetches the local state η, and the current approximation of fields at $@^\sharp$ is defined as (l,u). The local state is updated to take into account the new field. The assignment is then delegated using the auxiliary address variable $\underline{@^\sharp \cdot attr}$ representing the field $attr$ of $@^\sharp$.

Concretization (Figure 7.6). The concretization of the heap abstraction transforms concrete environments with auxiliary address variables and a concrete heap with only nominal object information into a concrete environment without auxiliary variables and a concrete heap with nominal and structural objects. It keeps the same environment for variables that are not auxiliary address variables. Auxiliary address variables are transformed into binding in the concrete heap. For each abstract address $@^\sharp$, the domain of structural objects of the concrete heap ranges in between the bounds provided by the local abstract state $\eta(@^\sharp)$.

7.2.4 Additional abstractions

The environment and heap domains are the main abstractions of the type analysis. This section briefly describes the other abstract domains on which the analysis rely on.

7.2.4.1 Flow tokens

Flow tokens are used to handle non-local control-flow operators. They are similar to the concrete ones (Section 6.1, Figure 6.2), except that addresses are now abstract. The whole state is lifted to a finite mapping over flow tokens.

$$\mathcal{F}^\sharp \stackrel{\text{def}}{=} \{ \mathit{cur}, \mathit{ret}, \mathit{brk}, \mathit{cont} \} \cup \{ \mathit{exn}\ @^\sharp \mid @^\sharp \in \mathsf{Addr}^\sharp \}$$

$$
\begin{array}{l}
\mathbb{E}^{\sharp}_{heap}[\![\, \mathit{get\_field}(@^\sharp \in \mathsf{Addr}^\sharp, attr \in \mathit{string})\,]\!]\sigma^\sharp \overset{\text{def}}{=} \\
\quad \text{let } b = \mathrm{get\_builtin\_field}(@^\sharp, attr) \text{ in} \\
\quad \text{if } b \neq \bot \text{ then return } \sigma^\sharp, b \text{ else} \\
\quad \text{let } \eta = \texttt{man.get}\ \sigma^\sharp \text{ in} \\
\quad \text{if } \eta(@^\sharp) = \text{🔒} \text{ then return } \sigma^\sharp, \bot \\
\quad \text{let } l, u = \eta(@^\sharp) \text{ in} \\
\quad \text{if } attr \in l \text{ then return } \mathbb{E}^{\sharp}[\![\, \underline{@^\sharp \cdot attr}\,]\!]\sigma^\sharp \\
\quad \text{else if } attr \in u \text{ then} \\
\qquad \text{return } \mathbb{E}^{\sharp}[\![\, \underline{@^\sharp \cdot attr}\,]\!](\texttt{man.set}\ \eta[@^\sharp \mapsto (l \cup \{\, attr \,\}, u)]\ \sigma) \cup \\
\qquad\qquad (\texttt{man.set}\ \eta[@^\sharp \mapsto (l, u \setminus \{\, attr \,\})]\ \sigma, \bot) \\
\quad \text{else return } \sigma^\sharp, \bot \\
\mathbb{E}^{\sharp}_{heap}[\![\, \mathit{has\_field}(@^\sharp \in \mathsf{Addr}^\sharp, attr \in \mathit{string})\,]\!]\sigma^\sharp \overset{\text{def}}{=} \\
\quad \text{letb } \sigma^\sharp, r = \mathbb{E}^{\sharp}[\![\, \mathit{get\_field}(@^\sharp, attr)\,]\!]\sigma^\sharp \text{ in} \\
\quad \text{if } r \neq \bot \text{ then return } \mathbb{E}^{\sharp}[\![\, \texttt{True}\,]\!]\sigma^\sharp \text{ else return } \mathbb{E}^{\sharp}[\![\, \texttt{False}\,]\!]\sigma^\sharp \\
\mathbb{S}^{\sharp}_{heap}[\![\, \mathit{set\_field}(@^\sharp \in \mathsf{Addr}^\sharp, attr \in \mathit{string}, val \in \mathsf{Addr})\,]\!]\sigma^\sharp \overset{\text{def}}{=} \\
\quad \text{let } \eta = \texttt{man.get}\ \sigma^\sharp \text{ in} \\
\quad \text{if } \eta(@^\sharp) = \text{🔒} \text{ then return } \bot \\
\quad \text{let } l, u = \eta(@^\sharp) \text{ in} \\
\quad \text{let } \sigma^\sharp = \texttt{man.set}\ \eta[@^\sharp \mapsto (l \cup \{\, attr \,\}, u \cup \{\, attr \,\})]\ \sigma^\sharp \text{ in} \\
\quad \text{return } \mathbb{S}^{\sharp}[\![\, \underline{@^\sharp \cdot attr} = val\,]\!]\sigma^\sharp
\end{array}
$$

Figure 7.5: Abstract semantics of field operators

$$
\begin{array}{rl}
\gamma_{heap}(\eta) = & \{\, ((e,h),(e',h')) \mid v \in \mathrm{dom} e \setminus \{\, \underline{@^\sharp \cdot \_}\,\} \Leftrightarrow e(v) = e'(v); \\
& @^\sharp \in \mathrm{dom} h \Leftrightarrow (\mathrm{fst} \circ h' = \mathrm{fst} \circ h \wedge \mathrm{dom}\ (\mathrm{snd} \circ h'(@^\sharp)) \in \gamma_{\mathsf{Objs}}(\eta(@^\sharp))); \\
& \underline{@^\sharp \cdot a} \in \mathrm{dom} e \Leftrightarrow (\mathrm{snd} \circ h')(@^\sharp)(a) = e(\underline{@^\sharp \cdot a});\ \}
\end{array}
$$

Figure 7.6: Concretization of the heap abstract domain

7.2.4.2 Containers

Containers such as lists and dictionaries are abstracted according to what we presented in Chapter 5.

7.2.4.3 Stateless abstractions close to the concrete semantics

Most of the concrete semantics of Python is abstracted by stateless domains. We show the example of the attribute operators. Their concrete semantics was defined in Figure 6.10, the abstract version is shown in Figure 7.7. We comment on the definition of attribute access $e.s$ to highlight the differences. The first three lines are similar to the concrete: we evaluate e, search for the `__getattribute__` method in the class of e, to call it on e and the attribute s.

The transfer function differs afterward since the global abstract state is a map from control-flow tokens to substates $\mathcal{F}^\sharp \to (\mathcal{E}^\sharp \times \mathcal{H}^\sharp)$ (while in the concrete, we used a set of states, each tagged by a control-flow token, $\mathcal{P}(\mathcal{F} \times \mathcal{E} \times \mathcal{H})$, which is isomorphic to $\mathcal{F} \times \mathcal{P}(\mathcal{E} \times \mathcal{H})$). We thus inspect each substate st tagged by its flow token τ, and return the corresponding evaluation. If we encounter an erroneous state that was not present in the initial state, and where an `AttributeError` has been raised, we try to use the fallback call to the `__getattr__` method. If we encounter the normal state, we return the computed value. For all other cases, we stop the evaluation by returning $\bot$.

7.2.5 Functions

The analysis of functions is performed in a context-sensitive fashion by inlining: when a function call is reached, we substitute the call by the body of the function and analyze it. This scheme supports easily dynamic dispatch as well as calling anonymous functions defined using `lambda`. The analysis of recursive functions is not currently supported in the implementation.

7.2.6 Full abstraction

In the case of the type analysis defined here, the whole abstract state maps flow tokens to the state of the recency abstraction, the abstract environment and the abstract heap. The abstract state is thus:

$$\mathcal{D}^\sharp = \mathcal{F}^\sharp \rightharpoonup \Big(\mathcal{P}(\mathsf{Addr}^\sharp) \times \Big(\mathcal{V} \rightharpoonup \mathcal{P}(\mathsf{Addr}^\sharp \cup \{\,\texttt{LocalUndef}\,\})\Big) \times (\mathsf{Addr}^\sharp \rightharpoonup \mathsf{ObjS})\Big)$$

We write $\gamma_{\mathcal{D}}$ the concretization of the whole abstract state. It is a simple lift of γ, which concretizes substates without flow tokens.

$$\gamma((\sigma^\sharp_{mem}, \sigma^\sharp_{heap}, \sigma^\sharp_{env})) = {\downarrow} \circ \gamma_{mem}(\sigma^\sharp_{mem}) \circ \gamma_{heap}(\sigma^\sharp_{heap}) \circ \gamma_{env}(\sigma^\sharp_{env})$$

The $\downarrow$ operator corresponds to taking the image of the relation; the composition operator between relations is the one induced by the composition of concretizations shown in Definition 2.82.

Listing 7.2: Python program with mutation and optional attribute addition

```
1  class A:
2    def __init__(self):
3      self.update(0)
4    def update(self, x):
5      self.val = x * 2
6  x = A()
7  y = x.val
8  z = x
9  if rand(0, 1): z.update('a')
10 if rand(0, 1): x.attr= 'b'
```

Example 7.2 **Analysis of Listing 7.2**

We focus on the analysis of Listing 7.2. We show the evolution of the abstract state for the current flow *cur* in Figure 7.8. For the sake of concision, elements (l, u) of the powerset approximation are displayed without repetition, i.e., we display $(l, u \setminus l)$. We also assume that builtin values are allocated to a single weak address for each type (this reduces the number of allocations, and the precision is unchanged as the fields of builtin values are immutable). The declaration of class `A` lines 1-5 creates three bindings in the environment and in the heap. In the environment, the identifier A now maps to the corresponding class

$\mathbb{E}^\sharp_{attrs}[\![\, e.s \,]\!]\sigma^\sharp \overset{\text{def}}{=}$
  letb $\sigma^\sharp, @^\sharp_e = \mathbb{E}^\sharp[\![\, e \,]\!]\sigma^\sharp$ in
  let $\sigma^\sharp, c = \mathbb{E}^\sharp[\![\, \mathit{mro\_search}(\mathit{type}(@^\sharp_e), \texttt{"\_\_getattribute\_\_"}) \,]\!]\sigma^\sharp$ in
  let $\sigma^\sharp_1, @^\sharp_{e.s} = \mathbb{E}^\sharp[\![\, c(@^\sharp_e, s) \,]\!]\sigma^\sharp$ in
  return $\lambda(\tau \in \mathcal{F})$.
    let $st = \sigma^\sharp_1(\tau)$ in
    if $\tau = \mathit{exn}\ @^\sharp \wedge \tau \notin \mathrm{dom}\sigma^\sharp \wedge \mathit{isinstance}(@^\sharp, \texttt{AttributeError})$ then
      let $\sigma^\sharp, @^\sharp_d = \mathbb{E}^\sharp[\![\, \mathit{mro\_search}(\mathit{type}(@^\sharp_e), \texttt{"\_\_getattr\_\_"}) \,]\!](\mathit{cur}, st)$ in
      if $@^\sharp_d \neq \bot$ then return $\mathbb{E}^\sharp[\![\, d(@^\sharp_e, s) \,]\!]\sigma^\sharp$
      else return $\sigma^\sharp, \bot$
    else if $\tau = \mathit{cur}$ then return $\sigma^\sharp, @^\sharp_{e.s}$
    else return $\sigma^\sharp, \bot$
$\mathbb{E}^\sharp[\![\, \mathit{hasattr}(obj, attr) \,]\!]\sigma^\sharp \overset{\text{def}}{=}$
  letb $\sigma^\sharp, @^\sharp_{obj} = \mathbb{E}^\sharp[\![\, obj \,]\!]\sigma^\sharp$ in
  letb $\sigma^\sharp, @^\sharp_{attr} = \mathbb{E}^\sharp[\![\, attr \,]\!]\sigma^\sharp$ in
  if $\neg \mathit{isinstance}(@^\sharp_{attr}, str)$ then return $\mathbb{S}^\sharp[\![\, \texttt{raise TypeError} \,]\!]\sigma^\sharp, \bot$ else
  let $\sigma^\sharp_1, @^\sharp_r = \mathbb{E}^\sharp[\![\, @^\sharp_{obj}.@^\sharp_{attr} \,]\!]\sigma^\sharp$ in
  if $@^\sharp_r = \bot$ then
    return $\lambda(\tau \in \mathcal{F})$.
      let $st = \sigma^\sharp_1(\tau)$ in
      if $\tau = \mathit{exn}\ @^\sharp_a \wedge \tau \notin \mathrm{dom}\sigma^\sharp \wedge \mathit{isinstance}(@^\sharp_a, \texttt{AttributeError})$ then $\mathbb{E}^\sharp[\![\, \texttt{False} \,]\!](\mathit{cur}, st)$
      else $\sigma^\sharp, \bot$
  else return $\mathbb{E}^\sharp[\![\, \texttt{True} \,]\!]\sigma^\sharp$
$\mathbb{S}^\sharp[\![\, x.s = e \,]\!]\sigma^\sharp \overset{\text{def}}{=}$
  letb $\sigma^\sharp, @^\sharp = \mathbb{E}^\sharp[\![\, x \,]\!]\sigma^\sharp$ in
  if $\mathit{hasattr}(\mathit{type}(@^\sharp), \texttt{"\_\_setattr\_\_"})$ then
    return $\mathbb{S}^\sharp[\![\, \mathit{type}(@^\sharp).\texttt{\_\_setattr\_\_}(x, s, e) \,]\!]\sigma^\sharp$
  return $\mathbb{S}^\sharp[\![\, \texttt{raise TypeError} \,]\!]\sigma^\sharp$
$\mathbb{S}^\sharp[\![\, \texttt{del}\ x.s \,]\!]\sigma^\sharp \overset{\text{def}}{=}$
  letb $\sigma^\sharp, @^\sharp = \mathbb{E}^\sharp[\![\, x \,]\!]\sigma^\sharp$ in
  if $\mathit{hasattr}(\mathit{type}(@^\sharp), \texttt{"\_\_delattr\_\_"})$ then
    return $\mathbb{S}^\sharp[\![\, \mathit{type}(@^\sharp).\texttt{\_\_delattr\_\_}(x, s) \,]\!]\sigma^\sharp$
  return $\mathbb{S}^\sharp[\![\, \texttt{raise TypeError} \,]\!]\sigma^\sharp$

Figure 7.7: Abstract semantics of attribute access, assignment and deletion

address. This class address has two attributes corresponding to the `__init__` and `update`

methods. Auxiliary variables for these attributes point to the methods' addresses in the environment. These methods are builtins: they cannot have additional attributes. At line 6, an instance of A is created at address $@^\sharp_{\texttt{inst}A}$. This instance has one field val, pointing to an integer (this address cannot have additional attributes either). After the binding $x = A()$, we thus have $x \mapsto @^\sharp_{\texttt{inst}A}$. The assignments at lines 7 and 8 add new bindings in the environment. During the non-deterministic call to `update` at line 9, the val field of the instance can be changed into a string. At line 10, a field $attr$ bound to another string is added nondeterministically. Thus, this field is only present in the overapproximation of the fields of $@^\sharp_{\texttt{inst}A}$ in the heap.

Line	$\epsilon \in \mathcal{E}^\sharp$	$\eta \in \mathcal{H}^\sharp$
5	$A \mapsto @^\sharp_{\texttt{cls}A}$ $\underline{@^\sharp_{\texttt{cls}A} \cdot \texttt{\_\_init\_\_}} \mapsto @^\sharp_{\texttt{fun init}}$ $\underline{@^\sharp_{\texttt{cls}A} \cdot \texttt{update}} \mapsto @^\sharp_{\texttt{fun update}}$	$@^\sharp_{\texttt{cls}A} \mapsto \{\texttt{\_\_init\_\_}, \texttt{update}\}, \emptyset$ $@^\sharp_{\texttt{fun init}} \mapsto$ 🔒 $@^\sharp_{\texttt{fun update}} \mapsto$ 🔒
6	$\underline{@^\sharp_{\texttt{inst}A} \cdot val} \mapsto @^\sharp_{\texttt{int}^\sharp}$ $x \mapsto @^\sharp_{\texttt{inst}A}$	$@^\sharp_{\texttt{inst}A} \mapsto \{val\}, \emptyset$ $@^\sharp_{\texttt{int}^\sharp} \mapsto$ 🔒
7	$y \mapsto @^\sharp_{\texttt{int}^\sharp}$	
8	$z \mapsto @^\sharp_{\texttt{inst}A}$	
9	$\underline{@^\sharp_{\texttt{inst}A} \cdot val} \mapsto \{@^\sharp_{\texttt{int}^\sharp}, @^\sharp_{\texttt{str}^\sharp}\}$	$@^\sharp_{\texttt{str}^\sharp} \mapsto$ 🔒
10	$\underline{@^\sharp_{\texttt{inst}A} \cdot attr} \mapsto @^\sharp_{\texttt{str}^\sharp}$	$@^\sharp_{\texttt{inst}A} \mapsto \{val\}, \{\boldsymbol{attr}\}$

Figure 7.8: Evolution of the abstract states of the example from Listing 7.2

Example 7.3 **Concretization of the states**

We consider the state reached at the end of the analysis of Listing 7.2, shown in Figure 7.8. We simplify it to remove the description of class A (Line 5 of Figure 7.8). The environment ϵ, the heap η and the recency abstraction $\sigma^\sharp_{mem}$ have the following states:

$$\epsilon = \left(x \mapsto @^\sharp_{\texttt{inst}A}, \underline{@^\sharp_{\texttt{inst}A} \cdot val} \mapsto \{@^\sharp_{\texttt{int}^\sharp}, @^\sharp_{\texttt{str}^\sharp}\}, \underline{@^\sharp_{\texttt{inst}A} \cdot attr} \mapsto @^\sharp_{\texttt{str}^\sharp}\right)$$
$$\eta = \left(@^\sharp_{\texttt{inst}A} \mapsto (\{val\}, \{attr\}), @^\sharp_{\texttt{int}^\sharp} \mapsto 🔒, @^\sharp_{\texttt{str}^\sharp} \mapsto 🔒\right)$$
$$\sigma^\sharp_{mem} = \{@^\sharp_{\texttt{inst}A}, @^\sharp_{\texttt{int}^\sharp}, @^\sharp_{\texttt{str}^\sharp}\}$$

We start by concretizing the environment into two cases, depending on the type of the *val* field of the instance.

$$\begin{aligned}\downarrow \circ \gamma_{env}(\epsilon) = &\left\{ \left(x \mapsto @^\sharp_{\texttt{inst}A}, \underline{@^\sharp_{\texttt{inst}A} \cdot val} \mapsto @^\sharp_{\texttt{int}^\sharp}, \underline{@^\sharp_{\texttt{inst}A} \cdot attr} \mapsto @^\sharp_{\texttt{str}^\sharp}\right), \right. \\ &\left. \left(@^\sharp_{\texttt{inst}A} \mapsto (\mathsf{Inst}(A), \top), @^\sharp_{\texttt{int}^\sharp} \mapsto (\texttt{int}(z), \top)\right) \;\middle|\; z \in \mathbb{Z} \right\} \\ \cup &\left\{ \left(x \mapsto @^\sharp_{\texttt{inst}A}, \underline{@^\sharp_{\texttt{inst}A} \cdot val} \mapsto @^\sharp_{\texttt{str}^\sharp}, \underline{@^\sharp_{\texttt{inst}A} \cdot attr} \mapsto @^\sharp_{\texttt{str}^\sharp}\right), \right. \\ &\left. \left(@^\sharp_{\texttt{inst}A} \mapsto (\mathsf{Inst}(A), \top), @^\sharp_{\texttt{str}^\sharp} \mapsto (\texttt{str}(s), \top)\right) \;\middle|\; s \in \mathit{string} \right\}\end{aligned}$$

These two cases are transformed into four different ones by the heap environment, de-

pending on whether the *attr* field exists or not in the instance.

$$
\begin{aligned}
\downarrow \circ \gamma_{env}(\eta) \circ \gamma_{env}(\epsilon) = \Big\{ & \left(x \mapsto @^\sharp_{\texttt{inst}A}\right), \Big(@^\sharp_{\texttt{inst}A} \mapsto (\textsf{Inst}(A), (val \mapsto @^\sharp_{\texttt{int}^\sharp})), \\
& @^\sharp_{\texttt{int}^\sharp} \mapsto (\texttt{int}(z), \blacksquare)\Big) \;\Big|\; z \in \mathbb{Z} \Big\} \\
\cup \Big\{ & \left(x \mapsto @^\sharp_{\texttt{inst}A}\right), \Big(@^\sharp_{\texttt{inst}A} \mapsto (\textsf{Inst}(A), (val \mapsto @^\sharp_{\texttt{str}^\sharp})), \\
& @^\sharp_{\texttt{str}^\sharp} \mapsto (\texttt{str}(s), \blacksquare)\Big) \;\Big|\; s \in \mathit{string} \Big\} \\
\cup \Big\{ & \left(x \mapsto @^\sharp_{\texttt{inst}A}\right), \Big(@^\sharp_{\texttt{inst}A} \mapsto (\textsf{Inst}(A), (val \mapsto @^\sharp_{\texttt{str}^\sharp}, attr \mapsto @^\sharp_{\texttt{str}^\sharp})), \\
& @^\sharp_{\texttt{str}^\sharp} \mapsto (\texttt{str}(s), \blacksquare)\Big) \;\Big|\; s \in \mathit{string} \Big\} \\
\cup \Big\{ & \left(x \mapsto @^\sharp_{\texttt{inst}A}\right), \Big(@^\sharp_{\texttt{inst}A} \mapsto (\textsf{Inst}(A), (val \mapsto @^\sharp_{\texttt{int}^\sharp}, attr \mapsto @^\sharp_{\texttt{str}^\sharp})), \\
& @^\sharp_{\texttt{int}^\sharp} \mapsto (\texttt{int}(z), \blacksquare), @^\sharp_{\texttt{str}^\sharp} \mapsto (\texttt{str}(s), \blacksquare)\Big) \;\Big|\; z \in \mathbb{Z}, s \in \mathit{string} \Big\}
\end{aligned}
$$

We show how the third case is concretized by the recency abstraction. As we mentioned earlier, the abstract addresses of builtin values are summarized into a single one. The recency then concretizes them into multiple concrete addresses. Even if a single value is picked by $\gamma_{\textsf{ObjN}^\sharp}$ (e.g., the value of the string allocated at $\gamma_{\texttt{str}^\sharp}$), the recency's concretization will then merge coherent states into a single one (Section 4.1.5.1; $\gamma_{\texttt{str}^\sharp}$ may be concretized into two addresses having different string values) constructed from multiple concrete states.

$$
\begin{aligned}
\gamma((\sigma^\sharp_{mem}, \eta, \epsilon)) \supseteq \Big\{ & \left(x \mapsto @_1\right), \Big(@_1 \mapsto (\textsf{Inst}(A), (val \mapsto @_2, attr \mapsto @_2)), \\
& @_2 \mapsto (\texttt{str}(s), \blacksquare)\Big) \;\Big|\; s \in \mathit{string} \Big\} \\
\cup \Big\{ & \left(x \mapsto @_1\right), \Big(@_1 \mapsto (\textsf{Inst}(A), (val \mapsto @_2, attr \mapsto @_3)), \\
& @_2 \mapsto (\texttt{str}(s), \blacksquare), @_3 \mapsto (\texttt{str}(s'), \blacksquare)\Big) \;\Big|\; (s, s') \in \mathit{string}^2 \Big\}
\end{aligned}
$$

We have strived to build a sound analysis, meaning that the abstract states computed by our abstract transfer functions over-approximate the concrete states reachable during any program execution. More formally, for any Python statement s, the following inclusion should hold: $\forall \delta \in \mathcal{D}^\sharp, \mathbb{S}[\![s]\!] \circ \gamma_{\mathcal{D}}(\delta) \subseteq \gamma_{\mathcal{D}} \circ \mathbb{S}^\sharp[\![s]\!](\delta)$. We have not proved the soundness theorem due to the number of cases of the concrete semantics. However, the proof should be simple in most cases since the abstract transfer functions of statements and expressions are close to the concrete ones (as shown in Section 7.2.4.3).

7.3 A relational reduced product bringing polymorphism

The analysis presented previously is polymorphic, as a variable may be abstracted as a set of addresses of different types. However, bounded parametric polymorphism *à la ML*[2] is impossible to express in this abstraction as we cannot infer that two variables pointing to multiple addresses have the same type. From an abstract interpretation point of view, we lack a relational domain.

[2]To express for example that x and y have the same type $t \in \{ \texttt{int}, \texttt{str} \}$.

Listing 7.3: Python program motivating polymorphism

```
1 if *: x, y = 1, 2
2 else: x, y = 'a', 'b'
3 z = x + y
```

Example 7.4 **Motivating polymorphism**

Consider the program in Listing 7.3. Our non-relational analysis can infer that, at the beginning of line 3, both `x` and `y` have type `int` or `str`. However, it cannot show that `x` and `y` are either both `int` or both `str`, and thus it raises a false `TypeError` alarm when evaluating `x + y`.

Definition 7.5 **Type equality domain**

We introduce an abstract domain $\mathcal{Q}^\sharp \overset{\text{def}}{=} \mathcal{V} \rightharpoonup \mathbb{N}$ to track type equalities between variables. It is defined as a partitioning of variables into equivalence classes of equally typed variables. Given $\kappa \in \mathcal{Q}^\sharp$, we ensure that two variables $\mathtt{x}$ and $\mathtt{y}$ verifying $\kappa(\mathtt{x}) = \kappa(\mathtt{y})$ will have the same nominal type. More precisely, we define an abstract equivalence relation $\equiv^\sharp \subseteq \mathsf{ObjN}^\sharp \times \mathsf{ObjN}^\sharp$ between nominal types:

$$\begin{aligned}\equiv^\sharp \overset{\text{def}}{=}\ & \{\,(\mathtt{int}^\sharp, \mathtt{int}^\sharp)\,\} \cup \{\,(\mathtt{str}^\sharp, \mathtt{str}^\sharp)\,\} \cup \{\,(\mathtt{fun}\ f, \mathtt{fun}\ f)\,\} \cup \{\,(\mathtt{cls}\ c, \mathtt{cls}\ c)\,\} \\ & \cup \{\,(\mathtt{meth}(@_1^\sharp, -), \mathtt{meth}(@_2^\sharp, -)) \mid typ(@_1^\sharp) \equiv^\sharp typ(@_2^\sharp)\,\} \\ & \cup \{\,(\mathtt{inst}(@_1^\sharp), \mathtt{inst}(@_2^\sharp)) \mid typ(@_1^\sharp) \equiv^\sharp typ(@_2^\sharp)\,\}\end{aligned}$$

Definition 7.6 **Concretization**

The concretization function $\gamma_{\mathcal{Q}} \in \mathcal{Q}^\sharp \to \mathcal{P}((\mathcal{V} \rightharpoonup \mathsf{Addr}^\sharp) \times (\mathsf{Addr}^\sharp \rightharpoonup (\mathsf{ObjN} \times \mathsf{ObjS})))$ gives the set of concrete states verifying the equality constraints of an abstract element in $\mathcal{Q}^\sharp$:

$$\gamma_{\mathcal{Q}}(\kappa) \overset{\text{def}}{=} \uparrow_{()} \{\,(e, h) \mid \forall \mathtt{x}, \mathtt{y} \in \mathrm{dom}\ \kappa : \kappa(\mathtt{x}) = \kappa(\mathtt{y}) \implies \mathtt{type}(e(x)) \equiv^\sharp \mathtt{type}(e(y))\,\}$$

In order to perform a type analysis with bounded parametric polymorphism, we construct a reduced product of the equality domain $\mathcal{Q}^\sharp$ and the non-relational environment domain $\mathcal{E}^\sharp$.

Two reduction operators $\psi_\uparrow, \psi_\downarrow \in (\mathcal{E}^\sharp \times \mathcal{Q}^\sharp) \to (\mathcal{E}^\sharp \times \mathcal{Q}^\sharp)$ are proposed to refine product states by propagating information between domains.

1. The reduction $\psi_\uparrow$ enriches κ with new type equalities. It searches for variables $\mathtt{x}$ and $\mathtt{y}$ such that both of them point to singleton objects with equivalent nominal types:

$$\begin{aligned}\epsilon(\mathtt{x}) &= \{\,@_\mathtt{x}^\sharp\,\} \\ \epsilon(\mathtt{y}) &= \{\,@_\mathtt{y}^\sharp\,\}\end{aligned} \quad \wedge \quad \mathtt{type}(@_\mathtt{x}^\sharp) \equiv^\sharp \mathtt{type}(@_\mathtt{y}^\sharp)$$

   In such case, we add the type equality $\kappa(\mathtt{x}) = \kappa(\mathtt{y})$.
2. The reduction operator $\psi_\downarrow$ refines the non-relational environment ϵ whenever two variables $\mathtt{x}$ and $\mathtt{y}$ are equally typed in κ and the type of $\mathtt{x}$ is more precise. We do so by pruning away the addresses referenced by $\mathtt{y}$ that are not equivalent to any object pointed by $\mathtt{x}$.

Example 7.7 **Back to the motivating example**

Let us consider again the motivating example in Listing 7.3. After the assignment `x, y = 1, 2`, both `x` and `y` point to singleton integer objects, which allows us to apply $\psi_\uparrow$ in order

Before reduction	After reduction
$\epsilon = x \mapsto @^\sharp_{int} \wedge y \mapsto @^\sharp_{int}$	ϵ
$\kappa = \bot$	$\kappa = x \mapsto 0, y \mapsto 0$

Figure 7.9: Example of $\psi_\uparrow$ reduction

to infer the type equality of `x` and `y` (the state is shown in Figure 7.9). The same reasoning is applied after the assignment `x, y = 'a', 'b'` in the `else` branch. Consequently, equality is preserved after joining the two abstract states at line 3. When evaluating `x` in the addition expression, a disjunction with two cases is created, one for each referenced abstract object. In each case, the reduction operator $\psi_\downarrow$ is applied to refine the type of `y` according to the type of `x`. Therefore, at the end of the program, we infer that no `TypeError` is raised. Moreover, the reduction $\psi_\uparrow$ will find that `x`, `y`, and `z` have the same type.

Remark 7.8 **Bounded parametric polymorphism**

In the motivating example, our analysis intuitively infers that x, y, z have type $\alpha \in \{\texttt{int}, \texttt{str}\}$. We believe this is close to bounded parametric polymorphism. In future work, we want to combine relationality with partial function summaries to deduce that f has type $\alpha \to \alpha$, $\alpha \in \{\texttt{int}, \texttt{str}\}$ in the program below.

```
def f(x, y): return x + y
f(1, 2)
f('a', 'b')
```

Remark 7.9 **Relational domain not supported anymore**

In the evaluation of our ECOOP paper [110], we have not noticed any improvement (in terms of alarms) using the relational domain in the benchmarks (cf. Section 7.6.2) This domain also required further adaptations to get the best results on containers. Let us consider the code below, where we perform an access to a list of integers or strings.

```
if *: l = [1, 2, 3]
else: l = ['a', 'b', 'c']
x = l[0]
```

After the join of the states of the conditional, our abstract state is ($\underline{\text{arr}}(\cdot)$ denotes auxiliary variable used by the summarization abstraction of lists, as defined in Chapter 5):

$$\left(l \mapsto \{ @^\sharp_{\texttt{List}^\sharp,1}, @^\sharp_{\texttt{List}^\sharp,2} \}, \underline{\text{arr}}(@^\sharp_{\texttt{List}^\sharp,1}) \mapsto @^\sharp_{\texttt{int}^\sharp}, \underline{\text{arr}}(@^\sharp_{\texttt{List}^\sharp,2}) \mapsto @^\sharp_{\texttt{str}^\sharp}\right)$$

The relational domain of this section will be unable to infer that x is an integer only when l is a list of integers. One way to alleviate this problem, is to unify the list addresses, so that the abstract state after the join of the abstract states is:

$$\left(l \mapsto \{ @^\sharp_{\texttt{List}^\sharp,(1\vee 2)} \}, \underline{\text{arr}}(@^\sharp_{\texttt{List}^\sharp,(1\vee 2)}) \mapsto \{ @^\sharp_{\texttt{int}^\sharp}, @^\sharp_{\texttt{str}^\sharp} \}\right)$$

In that case, the relational domain would be able to infer a type equality between x and the auxiliary $\underline{\text{arr}}(\cdot)$ variable. However, these unifications made the implementation much more complex, with little benefits in terms of precision in the benchmarks. In the end, we

did not update the implementation of this domain through the API changes of Mopsa, and the domain is not supported anymore.

7.4 Interaction with Python's type annotations

As the Python standard library is huge and partly written in C, we needed a way to support it quickly. We decided to leverage the work from the Typeshed project [152], which offers type annotations for a substantial part of the standard library. This project uses the standard type annotations recently introduced by the PEP 484 into Python [154]. These type annotations are quite powerful (they feature bounded polymorphism using `TypeVar`, structural subtyping support with `Protocol`, disjunctive function signatures with the `@overload` decorator, ...). These annotations can also be handy to avoid analyzing big, pure Python libraries when we want to focus our analysis on client code.

Example 7.10 **Example type annotation**

An example annotation of the function `fspath`, introduced in Listing 7.1, is shown in Listing 7.4. We define a type variable `T`, representing either string or bytes. Lines 2-3 define a class `PathL`, having a generic parent `Protocol[T]`, and a method `fspath` returning objects of type `T`. This class is used to define a structural type through the `Protocol` parent: objects having type `PathL` are objects having any nominal type, with a function `fspath` returning strings or bytes. The type of the `fspath` function is then defined disjunctively using the `@overload` decorator. It can take an object of structural type `PathL[T]`, and return an object of nominal type `T`. It can also take a string and return another string, and similarly for bytes.

Listing 7.4: Type annotations for fspath function

```
1 T = TypeVar('T', str, bytes)
2 class PathL(Protocol[T]):
3   def fspath(self) -> T: ...
4 @overload
5 def fspath(path: PathL[T]) -> T: ...
6 @overload
7 def fspath(path: str) -> str: ...
8 @overload
9 def fspath(path: bytes) -> bytes: ...
```

When a stubbed module is imported, our analyzer parses the corresponding annotated file and stores its functions (similarly for classes and variables). Then, when a stubbed function is called, we check that the arguments match the function signature. In that case, we assume that the function has no side effects and returns an object of the annotated return type, which we convert into an abstract object. We introduced two new expressions in the AST node: one to check that an expression has a given type (used for example to check the arguments in a function signature), and one to assume that an expression has a given type (for the return type of an annotated function). The type checking expression performs rewriting that ends up calling *isinstance*, while the type introduction expression ends up allocating objects of corresponding types (we also assume that returned objects are fresh objects, and not objects passed as arguments).

The example of Listing 7.4 provides multiple different signatures. In that case, our analyzer keeps only the signatures compatible with the passed arguments.

Remark 7.11 **Expressivity of annotations**

These annotations remain in general less expressive than our analysis, as side-effects (such as raised exceptions, aliasing) cannot be expressed. However, a type-and-effect system [101] could be used to alleviate this limitation.

The function annotation, written `Callable[arg_1, ..., arg_n, ret]` more difficult to handle. Our analysis is able to check that an object abides by this annotation. However, we cannot precisely represent functions of this type in our analysis: the transfer function of the type introduction is imprecise in this case.

Remark 7.12 **Soundness assumption**

The use of these annotations changes the soundness of our analyzer: exceptions raised by concrete functions where we used their annotated counterpart will not be reported. Our analyzer supports the declaration of raised exceptions in type annotation stub files, but this feature is specific to Mopsa and not supported by Typeshed[3] or PEP 484.

Remark 7.13 **Detecting wrong annotations**

Our implementation of the type analysis primarily targets uncaught exceptions. However, it can also display as alarms which type annotations of the program do not match with the abstract state we inferred.

Remark 7.14 **Recording typing assumptions**

In the implementation, it is possible to record which annotations have been used and display them as assumptions at the end of the analysis

7.5 Implementation

We have implemented our analysis into Mopsa, presented in Chapter 3. The type analysis consists of 2,100 lines of OCaml code (measured using `cloc`), the container abstraction consists of 1,300 lines of OCaml, and there are 6,300 lines of OCaml code defining the iterators and data model of Python.

7.5.1 Configuration

We show the configuration used by Mopsa in Figure 7.10. The top three lines consist of stateless domains. “Py.program” handles the beginning and the end of a program’s analysis: it initializes global and module variables at the beginning and transforms uncaught exceptions reaching the end of the execution into alarms to be displayed. “Py.desugar” rewrites some Python AST nodes into Universal ones. It handles rewriting of `for` loops (Listing 6.1 in the concrete), `with` statements (Listing 6.2), list comprehensions (Listing 6.9). “Py.flow” handles control-flow specific to Python, i.e., exceptions and generators (described in Section 6.2.8 and in Section 6.3.10 in the concrete). “U.intraproc”, “U.loops”, “U.interproc” are universal domains shared by the C and the Python analyses; they handle intraprocedural constructs, abstract fixpoint computation and interprocedural function inlining. “Py.libraries” implements some library stubs in OCaml, as well as the handling of type annotations shown in Section 7.4. “Py.data_model” implements domains such as the attributes domain shown in Section 7.2.4.3. It corresponds to the core language shown in Section 6.2. “Py.objects” handles builtin objects, described in the concrete in Section 6.3. “Py.environment” and “Py.heap” respectively implement the eponym abstractions shown in Section 7.2.2 and Section 7.2.3. “Py.lists”, “Py.tuples”, “Py.dicts” handle containers

[3] `https://github.com/python/typing/issues/71`

abstractions as described in Chapter 5. Tuples are immutable, and we abstract them by expansion. "U.recency" is the recency abstraction, handling dynamic memory abstraction, and presented in Chapter 4.

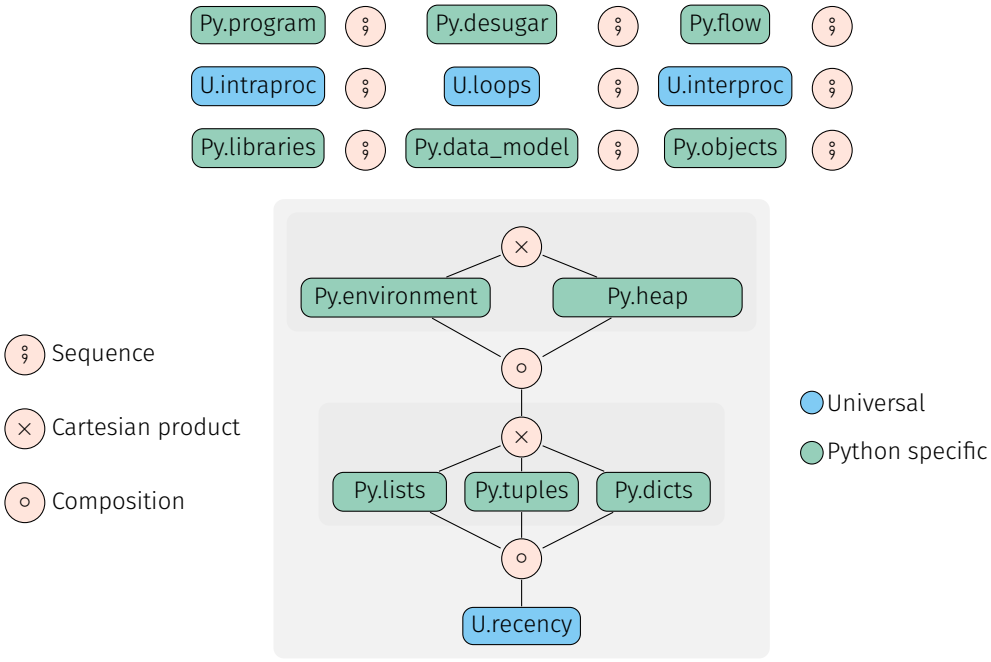

Figure 7.10: Configuration for Python's type analysis

7.5.2 Optimizations & extensions

During our initial testing of this type analysis, we noticed that it was slowed down by two factors:

1. The number of exceptions that were raised. Each new exception creates a new entry in the flow map, which created large abstract states.
2. The analysis of function calls (where the same functions were analyzed many times, with arguments having the same types in most calls).

These observations lead us to two optimizations described below.

7.5.2.1 Exception abstraction

When an exception is raised, we store the current abstract state with the exception flow token for the rest of the analysis, in order to reuse it if this exception is caught later on. However, imprecise analyses may raise exceptions frequently. For example, the smashing abstraction handling the list analysis needs (in order to be sound) to raise a potential `IndexError` at each list access, as the analysis does not keep precisely abstract list sizes. This created many different exceptions stored in the analysis state, but most were never used. To solve the problem, we abstracted sets of such exceptions for which the analysis is deemed *a priori* imprecise into a single abstract exception, joining the corresponding abstract states into one. The set of imprecise exceptions can be parameterized by the user. By default, the exceptions abstracted are `IndexError`, `KeyError` and `ValueError`, corresponding to imprecisions on list accesses, dictionary accesses, and list unpacking.

7.5.2.2 Towards a partially modular function analysis.

We have implemented a partially modular function analysis, which keeps the abstract input state, the abstract output state and the returned expression of function calls in a cache. When analyzing a function call, the cache is checked: if this function has already been analyzed with the *same* abstract input state, the analysis result is taken directly from the cache. Otherwise, the function is inlined, and the analysis result is cached afterward. In particular, using this cache does not reduce the precision of the analysis but greatly improves its running times. The experiments displayed in Figure 7.11 show that this cache, combined with the exception abstraction can provide a 32x speedup over the inlining-based analysis (`regex_v8.py`), while the memory usage increased by only 15%. In some cases, the inlining-based analysis and the cache-based analysis have the same running times: this may be due to a program having less user-defined functions to analyze, or the cache not being hit because the calling contexts are too different. We believe this cache is particularly efficient because we compute types rather than values: while the abstract state would change a lot during a value analysis (e.g., as loop indexes increase), the abstract state in the case of a type analysis is more stable.

We can also reuse the cache when the current input state is included in the input state kept in the cache. This is actually used in our implementation. In the benchmarks below, choosing this relaxed version improves the running times by 40% in one case (`choose.py`), but introduces imprecision in another case (22 out of the 25 alarms detected in `hexiom.py`).

Note that we keep analyzing functions on demand at each call-site knowing their calling context. We believe that performing a sound, context-free function call analysis, as done in most type systems, would not be practical for Python programs, as functions rely on implicit assumptions and may have side effects on their arguments or other variables not defined in the function scope. The cache-based analysis could still be improved to keep only the relevant parts of the whole abstract input and output states, such as the parts that may be read or changed by the function. This extension, which would help reuse more of the analysis results kept in the cache, is left as future work.

7.6 Experimental evaluation

In this part, we report previous evaluations of our implementation on several benchmarks:

1. We compare our analysis with five tools aiming at detecting incorrect programs potentially reaching runtime errors in Section 7.6.2. This part is based on the benchmarks of our ECOOP paper [110].
2. We evaluate the choice of allocation policy and the effect of performing garbage collection on abstract addresses in Section 7.6.3. The notion of allocation policy and abstract garbage collection were presented in Chapter 4. This part is based on the measurements of our SOAP paper [112].

Since the implementation evolved in between the publication of the articles, the measured analysis times are not exactly the same.

7.6.1 Benchmarks

We chose 5 of the biggest benchmarks from Typpete's unit tests (prefixed with Typpete's icon, , in Figure 7.11). Typpete [69] performs a type inference for Python programs. It will be presented in Section 7.6.2. We also took 12 benchmarks from Python's reference interpreter [143] (prefixed with in the table). These benchmarks are used by Python and Pypy developers to detect performance variations of their interpreters during development. Out of the 44 benchmarks currently available, we chose 12 with no external dependencies and few standard library module dependencies so that most tools can analyze them. We argue that

while the benchmarks are not very long, these Python programs are realistic and may call a lot of functions. For example, calling Python profiler `cProfile` on `chaos.py` shows more than 469,000 function calls. We also add three small tests focusing on characteristics we believe are paramount to performing a sound analysis of Python programs: taking into account object mutation and aliasing, as shown in Listing 7.2 (`mutation.py`); being able to precisely analyze introspection operators such as `isinstance` and `hasattr`, in order to analyze precisely a program calling the function `fspath` from Listing 7.1 for example (file `isinstance.py`, Listing 7.5); and analyzing precisely exception-related control-flow operators in order to have a precise analysis and avoid raising type errors later caught by an `except TypeError` statement for example (file `exception.py`, Listing 7.6). Finally, we analyze the two entry points (`processInput.py, choose.py`) of a real-world command-line utility from Facebook, called PathPicker (prefixed with ; the LOC for these files consists in the size of the file and all the PathPicker files imported by this one). These two parts are multifile projects depending on other modules from PathPicker (which are inlined and analyzed by our tool), as well as some standard library modules, including `re, subprocess, json, curses, posixpath, argparse, configparser, os, stat, locale, bz2, lzma` respectively handling regular expressions, external process calls, json files, curses command-line interfaces, file-related functions, argument parsing, configuration file parsing, operating-system and file status, internationalization of output and compression algorithms. As all these modules are at least partially written in C, we used the annotations from Typeshed [152] to support them. All program constructs used in the benchmarks are supported by our tool, meaning our analysis is sound on them.

We describe and quantify some of Python's features used the benchmarks in Table 7.1. "max. loops nested" is the maximum nesting level of loops when analyzing them. Due to function inlining, this number is higher than the maximum nesting level at the syntactic level of the program.

Listing 7.5: Python program isinstance.py

```
1 if isinstance(x, int): y = 4
2 else: y = 'a'
3 z = 2 + y
```

Listing 7.6: Python program exception.py

```
1 try: z = 2 + 'a'
2 except: z = 3.14
3 a = z+1
```

7.6.2 Comparison with other tools

7.6.2.1 Competing tools

We compare our tool with the abstract-interpretation-based value analysis of Python of Fromherz et al. [54], and three other tools having close goals: a tool by Fritz and Hage [53], Typpete [69] and Pytype [26]. We also include the static analysis part of RPython [2] in our comparison, whose goal is to compile a restricted subset of Python into more efficient programs. [53, 69, 26] try to infer a static typing of programs ensuring the absence of dynamic typing errors, while we go further and check whether dynamic typing errors can occur and result in exceptions that stop the program (hence, we can successfully analyze correct programs that are not typable but are nevertheless correct as they recover from dynamic type errors). We consider that dynamic type errors correspond to both the `AttributeError` and the `TypeError` exceptions. Both [54] and our analyzer generate as output the set of exceptions that may escape to the top-level, with detailed exception messages close to those given by Python. While this naturally includes type-related exceptions, we also take into account that even type errors can be caught and handled by the program, in which case they are not reported as errors. Contrary to [53, 69, 26], we also detect other errors (such as out-of-bound list accesses) in order to have a sound analysis (though we are imprecise in most cases).

	Explicit Exceptions	Introspection	Dynamic Attr. Addition	Operator Overloading	Object Inheritance	Lists	Dictionaries	Tuples	Max. Loops Nested	Number of Loops	Number of Functions	Number of Classes
`isinstance.py` (Listing 7.5)		●							0	0	0	0
`exception.py` (Listing 7.6)	●								0	0	0	0
`mutation.py` (Listing 7.2)			●						0	0	2	1
`disjoint_sets.py`						●			1	2	4	1
`functions.py`						●			0	0	7	0
`fannkuch.py`						●			2	4	1	0
`bellman_ford.py`						●			4	6	3	1
`float.py`						●	●		1	4	6	1
`coop_concat.py`					●	●			1	1	10	9
`spectral_norm.py`							●		2	6	6	0
`crafting.py`						●	●		3	9	0	0
`nbody.py`						●	●	●	3	10	6	0
`chaos.py`		●		●		●		●	4	19	24	4
`raytrace.py`	●		●	●	●	●		●	4	11	60	9
`scimark.py`	●	●		●	●	●		●	4	37	34	3
`richards.py`	●	●			●	●			3	4	39	14
`unpack_seq.py`						●		●	1	2	3	0
`go.py`						●	●	●	5	34	39	5
`hexiom.py`	●			●		●		●	5	57	38	5
`regex_v8.py`						●		●	2	67	13	0
`processInput.py`	●		●			●	●	●	2	40	77	9
`choose.py`	●	●				●	●	●	4	81	176	14

Table 7.1: Features used by some benchmarks

The tool developed by Fritz and Hage performs a data-flow analysis, computing the type of each variable. While the original paper [53] experiments various tradeoffs between performance and precision (using different widenings, flow-sensitivity, context-sensitivity, ...), we used the default arguments of the provided artifact. As mentioned in their paper, this tool does not handle exceptions nor generators. Its output is a dump of the data-flow map, associating to each program point the type of each variable. A program is untypable for this tool when the analyzer puts reachable variables to the bottom type.

Typpete encodes type inference of Python programs into a MaxSMT problem and passes it to Z3 [41] to solve it. If Z3 yields unsat, the program is untypable. Otherwise, the output of Typpete is a type annotation of the input program. It comes with around 40 examples on which we were able to test our analyzer. Typpete restricts its input to Python programs where variables have a single type in a program (but it handles subtyping: a variable having both types `int` and `str` will have type `object`) and dynamic attribute addition is not supported. When there is a type error, Z3 finds the inference problem to be unsatisfiable and Typpete shows a line in relationship with the type error. As the structure of the program is lost during the MaxSMT encoding, the line shown by Typpete is not always the line where the error will occur at runtime. Typpete supports the basics of the PEP 484 type annotations, and uses them for its stubs, or to guide the analysis on an input program.

Pytype is a tool developed by Google and actively used to maintain their codebase. Hence, it is more mature than the other tools. It performs an analysis that is not described formally, but it has a wide language and library support (it also uses Typeshed), allowing it to scale to large codebases. It outputs the last type of each variable when the typing is successful and can produce a type annotation of the input program. It also produces clear error messages looking like the exceptions raised by Python when it detects an erroneous program.

We obtained the analyzer developed by Fromherz et al. [54]. It performs a value analysis by abstract interpretation. Its output is a set of potentially uncaught exceptions.

RPython performs a data-flow analysis to check that a program is part of the subset it can efficiently compile. It outputs the control-flow graph with the inferred types.

We compare the analysis of these five tools to two different configurations of our analyzer:

▷ **Configuration 1** using inlining and no exception smashing of the alarms;
▷ **Configuration 2** using partially modular analysis and exception smashing in alarms (see Section 7.5.2).

These configurations use the type allocation policy π^{types} defined in Section 4.2, and no abstract garbage collection.

7.6.2.2 Performance and precision

We test the language support, the performance, and the precision of each tool. An analyzer may crash due to an unsupported construction (🐞) or may timeout after one hour of analysis (🕓). We measured the analysis time five times for each benchmark and tool, and the mean is displayed. The memory usage of our analysis is deduced from the statistics of OCaml's garbage collector. All tools are deterministic. In the evaluation of our tool in its most efficient configuration (Conf. 2), the column ⚠ displays the number of false alarms raised (the precision is identical in Conf. 1), with the smashed exceptions (corresponding to the imprecise exceptions raised by the list and dictionary abstractions) separated. The results are displayed in Figure 7.11.

We notice that our analysis is able to scale to benchmarks a few thousand lines long within a reasonable analysis time. Some benchmarks take longer to analyze: for example `hexiom.py` has a lot of nested loops, and functions are called multiple times, so the analyzer has a lot of fixpoint computations and inlining to perform (it performs 1770 analyses of 5-levels nested loops). It seems that the other type analyzers [53, 69, 26] do not perform fixpoint computations over loops (which, at least for the case of Typpete, seems sound as it infers more abstract types). Similarly, Typpete is able to perform an efficient analysis, although it lacks library support to analyze some Python benchmarks, and is unable to analyze programs where a variable is initialized in a (potentially unexecuted) loop. The tool from Fritz and Hage is quite fast (the running times are measured by running a docker container due to the software dating from 2011), but we will in Section 7.6.2.3 that it is unsound in most cases. It fails on `hexiom.py` due to a parsing error. Pytype is a more mature analyzer, and it does not fail on any of the benchmarks, but times out in the `regex_v8.py` benchmark (after reaching out to the development team, it appears to be a performance bug from Pytype in its analysis of big dictionaries[4]). The value analysis of Fromherz et al. [54] is unable to support the standard library functions needed for most benchmarks (supporting new library functions in the value analysis is more time-consuming, as it requires to include the effect of this function on the abstract values). On the benchmarks it is able to pass, our analysis is in average $8.5\times$ faster than the value analysis; it also scales to benchmarks $5\times$ longer. RPython is able to type 6 out of 22 benchmarks. In the 16 other cases, 5 seem to be due to internal bugs, while the 11 last cases are due to constructs unsupported by RPython. Compared to RPython, our analysis is able to fully analyze statically untypable programs (it will not stop at the first type exception, which can be caught later on).

[4] cf. `https://github.com/google/pytype/issues/484`

Name	LOC	Conf. 1	Conf. 2	⚠	[53]	Pytype	Typpete	[54]	RPython
`isinstance.py`	3	42ms	40ms	0	1.2s	0.78s	0.67s	10ms	4.9s
`exception.py`	3	37ms	34ms	0	1.3s	0.70s	0.57s	9ms	🐞
`mutation.py`	12	34ms	34ms	0	1.3s	0.75s	0.68s	11ms	🐞
`disjoint_sets.py`	45	70ms	59ms	0†	0.92s	0.91s	1.2s	🐞	8.8s
`functions.py`	58	41ms	39ms	0†🔑	1.2s	0.84s	1.1s	🐞	8.0s
`fannkuch.py`	59	76ms	69ms	0†	1.2s	0.80s	🐞	0.31s	🐞
`bellman_ford.py`	61	0.17s	0.24s	0†	1.4s	0.99s	1.4s	2.4m	7.1s
`float.py`	63	0.13s	82ms	0†	1.7s	0.92s	1.3s	0.84s	5.6s
`coop_concat.py`	64	45ms	43ms	0†	1.8s	0.81s	1.3s	20ms	🐞
`spectral_norm.py`	74	0.32s	0.19s	1	1.6s	0.98s	🐞	🐞	🐞
`crafting.py`	132	0.48s	0.41s	0†🔑	1.6s	0.97	1.7s	🐞	🐞
`nbody.py`	157	1.4s	0.80s	1†🔑*	1.7s	1.3s	🐞	🐞	🐞
`chaos.py`	324	8.9s	2.3s	0†*	13s	11s	🐞	🐞	🐞
`raytrace.py`	411	3.5s	1.5s	7*	36s	2.8s	🐞	🐞	🐞
`scimark.py`	416	0.85s	0.55s	2†	8.5s	4.4s	🐞	🐞	🐞
`richards.py`	426	11s	5.0s	2†*	38s	2.4s	🐞	🐞	7.8s
`unpack_seq.py`	458	13s	4.2s	0*	1.1s	7.4s	2.7s	14s	🐞
`go.py`	461	4.0m	15s	32†*	8.5s	3.4s	🐞	🐞	🐞
`hexiom.py`	674	6.9m	22s	25†🔑*	🐞	4.2s	🐞	🐞	🐞
`regex_v8.py`	1792	8.2m	15s	0†	4.9s	⏱	1.7m	🐞	🐞
`processInput.py`	1417	6.1s	4.8s	7†🔑*	2.4s	11s	🐞	🐞	🐞
`choose.py`	2562	8.6m	46s	17🔑†*	1.7s	15s	🐞	🐞	🐞

🐞 unsupported by the analyzer (crash) ⏱ timeout (after 1h)
Smashed Exceptions: KeyError 🔑, IndexError †, ValueError *

Figure 7.11: Analysis of Python benchmarks

Our analysis raises a few alarms. As the programs did not mix types implicitly, our analysis was sufficiently precise to avoid raising false alarms over type and attribute errors. However, the smashing abstraction of the lists and the dictionaries creates some false alarms: dictionary values having different types (and heterogeneously-typed lists) are smashed into content variables, triggering imprecision over the types in the rest of the analysis (cf. Example 5.24). In addition, the smashing abstraction currently does not keep track of the (potential) emptiness of lists: this creates a few false alarms, corresponding to the `UnboundlocalError` in `spectral_norm.py`, `nbody.py`, `bm_raytrace.py`, `bm_scimark.py` and 22 of those in `hexiom.py`. More generally, the absence of information on the length of the list means that each list access should raise a potential `IndexError` (this precision issue is addressed by the value analysis presented in Chapter 8). Similarly, `KeyError` exceptions are raised upon each dictionary access, and `ValueError` may be raised during list unpacking. The spurious `IndexError` are not raised by the value analysis of Fromherz et al. [54], which is able to track the length of lists. The alarms are not raised by the three other analyzers, as they focus on type errors only, and not on finding which exceptions may be raised. As each analyzer has its own output (and type system), we were unable to compare their precision in all cases, and only study the precision on the first three small examples. In the `isinstance.py` example, both our tool, the value analysis of Fromherz et al. [54] and Pytype are precise, but the others are imprecise (Fritz and Hage [53]'s tool yields an unsound result, Typpete declares the program incorrect). For `exception.py`, the tool of Fritz and Hage [53] does not support exceptions; while the value analysis of Fromherz et al. [54], Pytype and Typpete declare the program incorrect; our tool does not raise any alarm. Concerning `mutation.py`, both Pytype, the value analysis of Fromherz et al. [54] and our tool

are precise. Typpete is imprecise (it declares some integers and strings to be of object type), and the tool of Fritz and Hage [53] infers a variable holding a string as an integer.

7.6.2.3 Soundness evaluation

We experimentally check the soundness of the analyzers. We believe soundness is important in order to detect all potential errors. As each benchmark file was a correct Python program, we created erroneous variants having one type error (by introducing a string into an integer variable), in order to check the soundness of each analyzer (similarly to the evaluation of Typpete [69, pages 5-6]). We then ran each analyzer on those files (the correct and the erroneous one each time), and checked whether the inferred types and alarms were matching the behavior of the program: either the analysis seemed sound as the types and alarms were correctly raised, or the analysis was unsound (no error was detected in the erroneous variant). The injection of type errors to evaluate the soundness is simplistic, as our goal was to test the soundness of the other tools quickly. Our analyzer is sound by construction but may include implementation bugs, and it would be interesting to automate error injection to check the soundness more thoroughly experimentally.

We find that our analysis catches the errors in all erroneous variants and is thus sound – as expected – in these cases. Typpete is sound over the programs it can analyze. The tool from Fromherz et al. should be sound by construction, but an implementation error yields an unsound case in `unpack_seq.py`. The artifact from [53] is unable to detect errors in all cases except `fannkuch.py`. Pytype is not sound in a few cases (`bellman_ford.py`, `crafting_challenge.py`, `float.py`, `richards.py`, `spectral_norm.py`, `unpack_seq.py`).

7.6.2.4 Summary of the comparison

Our analysis is sound and reports a few false alarms. Preliminary results indicate that our analyzer is able to scale, at least on programs a few thousand lines long. The soundness evaluation showed that even simple errors such as replacing an integer with a string may go unnoticed for unsound analyzers, while our analysis was able to systematically detect these errors.

Comparatively, Pytype is the most advanced tool: it is able to scale and seems to support most of the standard library. However, it is unsound in some cases. Both Typpete and [54] perform a sound analysis, but they lack some language or library support in the bigger benchmarks. The tool from Fritz and Hage is able to analyze programs very quickly, and supports most benchmarks, but it is unsound in most cases. RPython has a different goal: it focuses on compiling a more static subset of Python efficiently. Most of the benchmarks use constructs that are too dynamic to be inside of RPython's scope.

7.6.3 Impact of the allocation policy and of the abstract garbage collector

In this section, we use the more complex benchmarks from Python's performance repository [143] and the PathPicker files.

We compare the analysis time, the memory used, and the number of alarms raised depending on the allocation policy used in Figure 7.12, and the best results with noticeable performance improvements in bold font. The policies used were defined in Section 4.2. The first policy π^{types} does not split abstract addresses by their allocation site (except for containers such as lists). The second policy π_{loc} splits all abstract addresses by their allocation site. In the case of a type analysis, changing sensitivities does not make much of a difference. There are some specific cases (`chaos.py`, `richards.py`, `regex_v8.py`) where having no location sensitivity vastly improves analysis time and memory usage. In these cases, the location-sensitive analysis is more costly because accesses to containers holding objects of the same type with different

Name	No loc. sensitivity: p^{types}			Loc. sensitivity: p_{loc}		
	Time	Mem.	⚠	Time	Mem.	⚠
`fannkuch.py`	0.34s	3MB	1	0.33s	3MB	1
`float.py`	0.19s	3MB	2	0.24s	3MB	2
`spectral_norm.py`	0.70s	6MB	2	0.77s	6MB	2
`nbody.py`	1.6s	3MB	5	1.4s	4MB	5
`chaos.py`	**7.4s**	**42MB**	3	56s	300MB	3
`raytrace.py`	14s	74MB	8	**2.2s**	**32MB**	8
`scimark.py`	1.4s	12MB	3	1.3s	13MB	3
`richards.py`	**13s**	**112MB**	4	9.4m	2444MB	4
`unpack_seq.py`	8.3s	7MB	1	7.9s	9MB	1
`go.py`	27s	345MB	35	27s	345MB	35
`hexiom.py`	1.1m	525MB	6	1.0m	525MB	6
`regex_v8.py`	**23s**	**18MB**	1	1.3m	36MB	1
`processInput.py`	10s	64MB	13	9.5s	64MB	**11**
`choose.py`	1.1m	1.6GB	23	1.2m	1.6GB	**20**
Total	240s	2.8GB	107	883s	5.4GB	102

Figure 7.12: Comparing allocation sensitivities (type analysis, with AGC)

allocation sites create disjunctions. In the case of `raytrace.py`, the location-sensitive analysis is quicker: the analysis without location sensitivity performs more weak updates (when allocating an address that already exists through the recency abstraction), which are costlier than having multiple recent addresses in this case. Lastly, the location-sensitive analysis of `processInput.py` and `choose.py` is a bit more precise, allowing to rule out five alarms in total.

We show the effects of the abstract garbage collector (AGC) in Figure 7.13. The AGC does not change the number of exceptions detected in the benchmarks. We believe the precision improvements, as illustrated in Example 4.13, do not happen enough to affect the number of exceptions detected. We show the time spent and the memory used during each analysis. The last column gives the relative change in time spent and memory used when the AGC is enabled. The activation of the AGC is extremely beneficial: it almost halves the global memory usage and brings a 38% analysis time improvement. In addition, the most significant speedups are observed on the largest files, showing the scalability of the approach. We measured that the AGC represents less than 6% of the analysis time for all benchmarks except the last, where it takes 30% of the analysis time. In the results, the AGC is called after each assignment, where the right-hand side is a (function, method, or object) call. We have tested running the AGC at only a fraction of those assignments, but the results were not as satisfying.

7.7 Related work

A middle-end between dynamic and static type analysis is gradual typing [136, 61]. In that case, the programmer annotates parts of the program, which can then be typechecked. The unannotated parts of the program have an unknown type called top, from which any static type can be cast to and from. The soundness theorem of gradual typing then guarantees that if a program gradually typechecks, the only type errors that may occur at runtime are casts concerning variables having type top. Gradual typecheckers for Python include Mypy [91] and Pyre [147]. Neither tool support dynamic attribute addition in programs. By contrast, our type analysis is more expressive (for example, it handles dynamic attribute addition), and completely automatic (it does not require any annotation to run).

Name	Without AGC		With AGC		Rel. Impr.	
	Time	Mem.	Time	Mem.	Time	Mem.
fannkuch.py	0.32s	3MB	0.34s	3MB	0%	0%
float.py	0.22s	3MB	0.19s	3MB	16%	0%
spectral_norm.py	0.72s	6MB	0.70s	6MB	3%	0%
nbody.py	1.7s	4MB	1.6s	3MB	10%	25%
chaos.py	10s	64MB	7.4s	42MB	28%	34%
raytrace.py	17s	74MB	14s	74MB	16%	0%
scimark.py	1.5s	13MB	1.4s	12MB	5%	8%
richards.py	16s	227MB	13s	112MB	21%	51%
unpack_seq.py	10s	9MB	8.3s	7MB	19%	22%
go.py	38s	604MB	27s	345MB	31%	43%
hexiom.py	2.2m	1.1GB	1.1m	525MB	49%	50%
regex_v8.py	30s	24MB	23s	18MB	23%	25%
processInput.py	14s	85MB	10s	64MB	28%	25%
choose.py	2.0m	3.2GB	1.1m	1.6GB	43%	50%
Total	6.5m	5.4GB	4.0m	2.8GB	38%	47%

Figure 7.13: Measurement of the AGC's effect

The closest approaches to our work [53, 26, 69, 2] have been described in the experimental evaluation (Section 7.6.2). It should be noted that Fritz & Hage [53] test many different parameter instantiations of their data-flow analysis. We believe that in the context of formal verification, a precise, context-sensitive, sound type analysis is useful. The flow-sensitivity is needed to precisely analyze exception catching statements, but neither have we tested this hypothesis on a larger scale nor have we tried selective flow-sensitivity, contrary to [53]. [157] presents a predictive analysis based on symbolic execution for Python. It consistently finds bugs and scales to projects of thousands of lines of codes, but it does not cover all executions, and is thus not sound. Fromherz et al. [54] performs a static value analysis by abstract interpretation. They use a separation of values similar to the ones presented in TAJS [74]. Their analysis focuses on values, which is more expressive than our type analysis, if we exclude the type equality domain of Section 7.3. Our type analysis is more scalable in its implementation, as supporting new constructs consists in providing a type signature (and knowing the side effects of this function, including the potentially raised exceptions). To scale more quickly, we can also reuse Typeshed and its type annotations to support most of the standard library (though we will lose any side effect of the annotated function in that case). The type analysis also uses less memory and is quicker: we store type information rather than abstract values, and the fixpoint computations during the analysis of loops converge more quickly (types vary less than values, for example during loop iterations). The experiments of this value analysis consisted in some of Python's unit tests and some of Python's benchmarks. As the unit tests consist mostly in equality assertions over values, our type analyzer is unable to verify these. However, the running times for the type analysis on those tests are similar to the ones described in [54]. The benchmarks were shown in Figure 7.11.

7.8 Conclusion

We have defined a type analysis, which is able to detect uncaught type-related exceptions that may be raised during a program execution. This analysis is sound, and its modular implementation scales to benchmarks a few thousand lines long. In addition, we found that compared

to other type analyses, we uniquely take into account dynamic Python features such as object mutability, introspection operators, and exception-based control-flow statements. We show in the next chapter how the type analysis can be refined into a value analysis similar to the one of Fromherz et al. [54], and compare both our type and value analyses.

8

Value Analysis

The type analysis was only precise on type and attribute errors. In order to be sound, it raised exceptions for other categories of exceptions, but it was imprecise in these cases. We consider the example of Listing 8.1 (already shown in Section 4.2), computing the average weight of four tasks defined in a list, to compare informally the type and the value analyses.

Listing 8.1: Python program computing average of tasks

```
1  class Task:
2    def __init__(self, weight):
3      if weight < 0: raise ValueError
4      self.weight = weight
5
6  l = [Task(2), Task(1), Task(3), Task(5)]
7  m = 0
8  for i in range(len(l)):
9    m = m + l[i].weight
10 m = m  (i + 1)
```

Example 8.1 **Type analysis of Listing 8.1**

The type analysis of the previous chapter infers that `l` is a list of instances of the `Task` class. It infers that `weight` is an integer, but without a numerical domain, it has to assume it can take any possible non-positive integer values. It will thus raise false `ValueError` alarms during each instantiation of `Task`. It then infers that `m` is an integer at line 7. The type analysis does not maintain numerical length information for lists, and thus creates an out-of-bound access alarm (an `IndexError` exception in Python) at each list access to ensure that all reachable exceptions handled are included in the analysis. As `l` is a list of `Task` instances and provided that `l[i]` is a valid list access, the attribute `weight` exists for `l[i]` and is an integer. Knowing that both `m` and `l[i].weight` are integers, the type analysis infers that the `+` operator is resolved as a call to the `__add__` method of `m`, which returns an integer. The type analysis is unable to know for sure that loop body is executed, and in particular that variable `i` is defined at line 10. It will thus create a `NameError` false alarm, and the division also raises a `ZeroDivisionError`. In the end, the analysis inferred precise type information for `l` and `m`. It infers that if `i` exists, then it is an integer, and that seven false alarms can be raised: four `ValueErrors`, one `IndexError`, one `NameError` and one `ZeroDivisionError`.

Example 8.2 **Value analysis of Listing 8.1**

During the analysis of line 3, the type analysis inferred that `weight` is an integer, so that the comparison calls `int.__lt__(weight, 0)`. The type analysis inferred that this call returns a boolean, and the value analysis refines the result: the return will be `True`, as the weight is positive. The value analysis infers that `l` is a list of size four and that the weight of each `Task` is between one and five. It finds that the `for` loop is executed and that all list accesses are valid, avoiding all seven false alarms generated by the type analysis.

We explain in Section 8.1 how the value analysis is defined, compared to the domains used in the type analysis. An experimental evaluation of this analysis, comparing it with the type analysis and evaluating the impact of allocation policies and abstract garbage collection is performed in Section 8.2

8.1 Value analysis as a refinement of the type analysis

In the type analysis, the abstract environment maps variable to abstract addresses, which have a type. The domains' evaluations used addresses as a means to communicate with each other.

Example 8.3 **Type analysis, communication through addresses**

We show communication through abstract address on a simple example in Figure 8.1. The execution of `x = 1`, handled by the type environment first delegates the evaluation of the constant 1. This evaluation is handled by the domain in charge of Python's `int` object and returns a weak integer address. The state of the environment domain is then updated so that x maps to this address.

In order to obtain a value analysis, the main modification consists in upgrading the environment domain and adding a numerical abstract domain. The environment domain will delegate operations on builtin values to the corresponding numerical domains. Since builtin value objects were summarized into a single abstract address, we abstract numerical objects through variables rather than addresses. In particular, we use $\underline{\text{int}}(x)$ (resp. $\underline{\text{float}}(x)$, $\underline{\text{str}}(x)$) to denote the integer (resp. floating-point, string) value of the builtin stored by variable x. In the case of the example assignment, this means that the environment of the type analysis is complemented with $\underline{\text{int}}(x) \mapsto [1, 1]$ in our example. The domains' evaluations now use an address, along with an optional expression denoting the builtin value. In the rest of this thesis, these evaluations are written $\langle @^\sharp, e \rangle$, where $@^\sharp$ is an abstract address, and e is either a universal expression or $\bot$.

Example 8.4 **Value analysis**

We show how the value analysis handles the assignment `x = 1` in Figure 8.2. The evaluation of the constant 1 is delegated similarly to the domain in charge of Python's `int` object. In addition to the `int` address already returned by the type analysis, the `int` domain also returns the constant expression 1. While evaluated 1 (in black) represented the Python constant, this 1 represents the mathematical integer (natively handled by the numerical domain). The local state of the environment domain is updated just as in the type analysis. The environment then delegates the value assignment to the numerical abstract domain. In the end, we know that x is an integer whose builtin value is $\underline{\text{int}}(x)$, which is 1 in the final state $\sigma_3^\sharp$.

We show the updated transfer functions of the environment domain in Figure 8.3. These transfer functions start just like the ones from the type environment do (Figure 7.2). The added operations correspond to the delegation to numerical domains. In the case of the assignment `x`

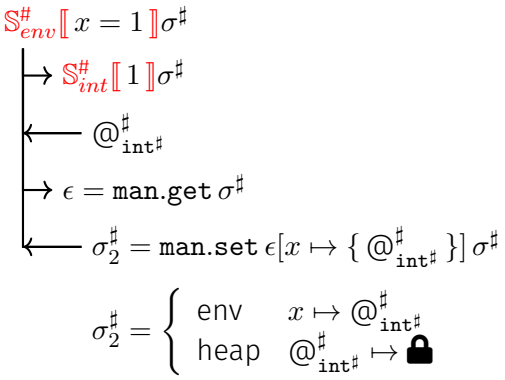

Figure 8.1: Type analysis example

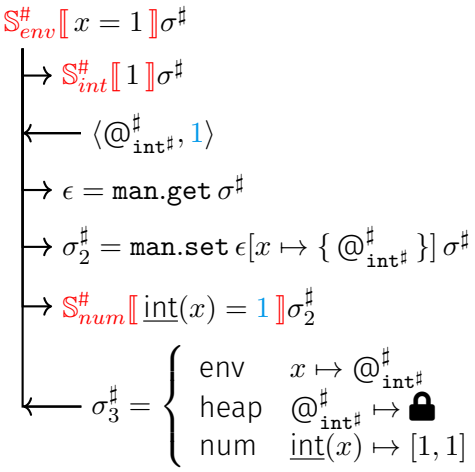

Figure 8.2: Value analysis example

`= e`, we start by evaluating e into an abstract address, and an optional expression denoting the builtin value of e. If this evaluation returns a disjunction of expressions, the monadic operator handles the disjunction (cf. Definition 6.4). We can thus define the rest of the transfer function on a single expression. We then update the environment in the global abstraction. Then, if the type of the abstract address corresponds to a builtin value (i.e., integers, floating-points, or strings), a numerical assignment is delegated. The suppression of a variable x consists in suppressing it in the local state, as well as in the underlying numerical domains if needed. The evaluation of a variable is the same disjunction where the environment's state is refined. In the case of builtin values, the returned address is accompanied by the corresponding numerical auxiliary variable, where auxiliary variables for other numerical types have been removed.

Listing 8.2: Python program with type disjunction

```
1 if *: x = 3
2 else: x = 'a'
3 y = x * 2
```

Remark 8.5 **Removing auxiliary variables of other types**

During the evaluation of a variable x by the environment domain, the case disjunction removes auxiliary variables of other types (e.g., the case where x is an integer removes $\underline{\text{float}}(x)$ and $\underline{\text{str}}(x)$ from the value domains). We show why these removals are necessary to ensure a consistent state. We consider the program in Listing 8.2. At the beginning of line 3, x is either the integer 3, or the string "a" (the heap abstraction has been omitted):

$$\sigma^\sharp = \begin{cases} \text{env} & x \mapsto \{\, @^\sharp_{\texttt{int}^\sharp}, @^\sharp_{\texttt{str}^\sharp} \,\} \\ \text{num} & \underline{\text{int}}(x) \mapsto [3, 3] \\ \text{str} & \underline{\text{str}}(x) \mapsto \{\, \text{"}a\text{"} \,\} \end{cases}$$

If we evaluate `y = x * 2` and *do not perform the removals*, we obtain two different states (depending on whether x has been evaluated into an `int` or a `str`)

$$\sigma^\sharp_{\texttt{int}} = \begin{cases} \text{env} & x \mapsto \{\, @^\sharp_{\texttt{int}^\sharp} \,\} \\ \text{num} & \underline{\text{int}}(x) \mapsto [6, 6] \\ \text{str} & \underline{\text{str}}(x) \mapsto \{\, \text{"}a\text{"} \,\} \end{cases} \qquad \sigma^\sharp_{\texttt{str}} = \begin{cases} \text{env} & x \mapsto \{\, @^\sharp_{\texttt{str}^\sharp} \,\} \\ \text{num} & \underline{\text{int}}(x) \mapsto [3, 3] \\ \text{str} & \underline{\text{str}}(x) \mapsto \{\, \text{"}aa\text{"} \,\} \end{cases}$$

$\mathbb{S}^\sharp_{env}[\![x = e]\!]\sigma^\sharp \stackrel{\text{def}}{=}$
  letb $\sigma^\sharp, \langle @^\sharp, e_v \rangle = \mathbb{E}^\sharp[\![e]\!]\sigma^\sharp$ in
  let $\epsilon = \texttt{man.get}\ \sigma^\sharp$ in
  let $\epsilon = \epsilon[x \mapsto \{ @^\sharp \}]$ in
  let $\sigma^\sharp = \texttt{man.set}\ \epsilon\ \sigma^\sharp$ in
  if $\texttt{type}(@^\sharp) = \texttt{int}^\sharp$ then $\mathbb{S}^\sharp[\![\underline{\text{int}}(x) = e_v]\!]\sigma^\sharp$
  else if $\texttt{type}(@^\sharp) = \texttt{float}^\sharp$ then $\mathbb{S}^\sharp[\![\underline{\text{float}}(x) = e_v]\!]\sigma^\sharp$
  else if $\texttt{type}(@^\sharp) = \texttt{str}^\sharp$ then $\mathbb{S}^\sharp[\![\underline{\text{str}}(x) = e_v]\!]\sigma^\sharp$
  else $\sigma^\sharp$
$\mathbb{S}^\sharp_{env}[\![\texttt{del}\ x]\!]\sigma^\sharp \stackrel{\text{def}}{=}$
  let $\epsilon = \texttt{man.get}\ \sigma^\sharp$ in
  if $x \notin \text{dom}\ \epsilon$ then return $\mathbb{S}^\sharp[\![\texttt{raise NameError}]\!]\sigma^\sharp$ else
  if $\epsilon(x) = \texttt{LocalUndef}$ then return $\mathbb{S}^\sharp[\![\texttt{raise UnboundLocalError}]\!]\sigma^\sharp$ else
  letb $\sigma^\sharp, \langle @^\sharp, e_v \rangle = \mathbb{E}^\sharp[\![e]\!]\sigma^\sharp$ in
  let $\sigma^\sharp = \texttt{man.set}\ (\epsilon \setminus \{ x \})\ \sigma^\sharp$ in
  if $\texttt{type}(@^\sharp) = \texttt{int}^\sharp$ then $\mathbb{S}^\sharp[\![\texttt{del}\ \underline{\text{int}}(x)]\!]\sigma^\sharp$
  else if $\texttt{type}(@^\sharp) = \texttt{float}^\sharp$ then $\mathbb{S}^\sharp[\![\texttt{del}\ \underline{\text{float}}(x)]\!]\sigma^\sharp$
  else if $\texttt{type}(@^\sharp) = \texttt{str}^\sharp$ then $\mathbb{S}^\sharp[\![\texttt{del}\ \underline{\text{str}}(x)]\!]\sigma^\sharp$
  else $\sigma^\sharp$
$\mathbb{E}^\sharp_{env}[\![x \in \mathcal{V}]\!]\sigma^\sharp \stackrel{\text{def}}{=}$
  let $\epsilon = \texttt{man.get}\ \sigma^\sharp$ in
  if $x \notin \text{dom}\ \epsilon$ then return $\mathbb{S}[\![\texttt{raise NameError}]\!]\sigma^\sharp, \bot$ else
  if $\epsilon(x) = \texttt{LocalUndef}$ then return $\mathbb{S}[\![\texttt{raise UnboundLocalError}]\!]\sigma^\sharp, \bot$ else return
  $\cup_{@^\sharp \in \epsilon(id)} \Big($let $\sigma^\sharp = \texttt{man.set}\ \epsilon[x \mapsto \{ a \}]\ \sigma^\sharp$ in
    if $\texttt{type}(@^\sharp)) = \texttt{int}^\sharp$ then $\mathbb{S}^\sharp[\![\texttt{del}\ \underline{\text{float}}(x)]\!] \circ \mathbb{S}^\sharp[\![\texttt{del}\ \underline{\text{str}}(x)]\!]\sigma^\sharp, \langle @^\sharp, \underline{\text{int}}(x) \rangle$
    else if $\texttt{type}(@^\sharp) = \texttt{float}^\sharp$ then $\mathbb{S}^\sharp[\![\texttt{del}\ \underline{\text{int}}(x)]\!] \circ \mathbb{S}^\sharp[\![\texttt{del}\ \underline{\text{str}}(x)]\!]\sigma^\sharp, \langle @^\sharp, \underline{\text{float}}(x) \rangle$
    else if $\texttt{type}(@^\sharp) = \texttt{str}^\sharp$ then $\mathbb{S}^\sharp[\![\texttt{del}\ \underline{\text{int}}(x)]\!] \circ \mathbb{S}^\sharp[\![\texttt{del}\ \underline{\text{float}}(x)]\!]\sigma^\sharp, \langle @^\sharp, \underline{\text{str}}(x) \rangle$
    else $\sigma^\sharp, \langle @^\sharp, \bot \rangle \Big)$

Figure 8.3: Transfer functions of the environment abstraction

Both states are confusing: $\sigma^\sharp_{\texttt{int}}$ still has a binding for $\underline{\text{str}}(x)$, although x cannot point to a string anymore, according to the environment domain. Using the stacked domains (cf. Remark 3.34), it would be possible to let the environment domain unify those two states (by removing $\underline{\text{str}}(x)$ in $\sigma^\sharp_{\texttt{int}}$, and $\underline{\text{int}}(x)$ in $\sigma^\sharp_{\texttt{str}}$) before joining them. Instead, we ensure the states are consistent through the removals performed during the case disjunction over the

evaluation of x. With the removals, we get:

$$\left\{\begin{array}{lll}\text{env} & x \mapsto \{\, @^\sharp_{\texttt{int}^\sharp} \,\} \\ \text{num} & \underline{\text{int}}(x) \mapsto [6,6]\end{array}\right. \sqcup^\sharp \left\{\begin{array}{lll}\text{env} & x \mapsto \{\, @^\sharp_{\texttt{str}^\sharp} \,\} \\ \text{str} & \underline{\text{str}}(x) \mapsto \{\,\text{"}aa\text{"}\,\}\end{array}\right. = \left\{\begin{array}{lll}\text{env} & x \mapsto \{\, @^\sharp_{\texttt{str}^\sharp}, @^\sharp_{\texttt{str}^\sharp} \,\} \\ \text{num} & \underline{\text{int}}(x) \mapsto [6,6] \\ \text{str} & \underline{\text{str}}(x) \mapsto \{\, aa\text{"}\,\}\end{array}\right.$$

Remark 8.6 **Auxiliary variables in the implementation**

In Mopsa, variables are records containing a field defining the `type` (Definition 3.1). Given a Python variable `v` (having a `type` `py`), we use `{v with vtyp=int}` to represent the auxiliary variable $\underline{\text{int}}(v)$.

Example 8.7 **Execution of a statement in the value analysis**

We consider the motivating example of this chapter, Listing 8.1. To avoid case disjunctions, we assume that the list `l` has been instantiated with `Task(2)` only. We analyze the statement `m = m + l[i].weight`, corresponding to the loop's body, in a state $\sigma^\sharp$ reached at the loop's entry. We show a simplified analysis of this statement in Figure 8.4.

- The transfer function of the assignment in the environment domain (presented in Figure 8.3) starts by evaluating `m + l[i].weight`.
  - The domain describing the semantics of binary operators handles this evaluation. Its concrete semantics was described in Figure 6.23. It start by evaluating both sides of the addition.
    - The evaluation of variable m is handled by the environment domain (Figure 8.3). The local state of the environment is accessed to return that m points to an integer object, whose value is described by an auxiliary variable: $\langle @^\sharp_{\texttt{int}^\sharp}, \underline{\text{int}}(m)\rangle$.
    - The evaluation of `l[i].weight` is handled by the attribute domain (presented in the concrete in Figure 6.10, and in the abstract in Figure 7.7). It starts by evaluating `l[i]`.
      - The expression `l[i]` is handled by the index domain, implementing the semantics shown in Figure 6.11. It starts by evaluating both sides of the index access. Both sides are handled by the environment domain.
        - The evaluation of l returns a list address $@^\sharp_{list,\mathsf{r}}$, with no associated value expression.
        - The evaluation of i returns an integer address $@^\sharp_{\texttt{int}^\sharp}$, with an auxiliary variable $\underline{\text{int}}(i)$ as the associated value expression.
        - The index domain ultimately calls the method `list.__getitem__`. The arguments of this method should actually be the evaluated versions of l and i (i.e., $\langle @^\sharp_{list,\mathsf{r}}, \bot\rangle$ and $\langle @^\sharp_{\texttt{int}^\sharp}, \underline{\text{int}}(i)\rangle$), but we stick to the initial expressions in this description for the sake of readability. We first check that i is a valid index. The list summarization domain returns the auxiliary array content variable $\underline{\text{arr}}(@^\sharp_{list,\mathsf{r}})$ (Figure 5.3). This variable is evaluated by the environment domain into $@^\sharp_{Task,\mathsf{r}}$, with no value expression.
    - The evaluation of *mro_search* for the `__getattribute__` method is omitted, its result is the function `object.__getattribute__`.
    - The semantics of `object.__getattribute__` is shown in Figure 6.31. We omit its checks and calls, apart from the last one, which evaluates *get_field*$(l[i], \text{"}weight\text{"})$. This evaluation is handled by the heap abstraction (Figure 7.5), creating an auxil-

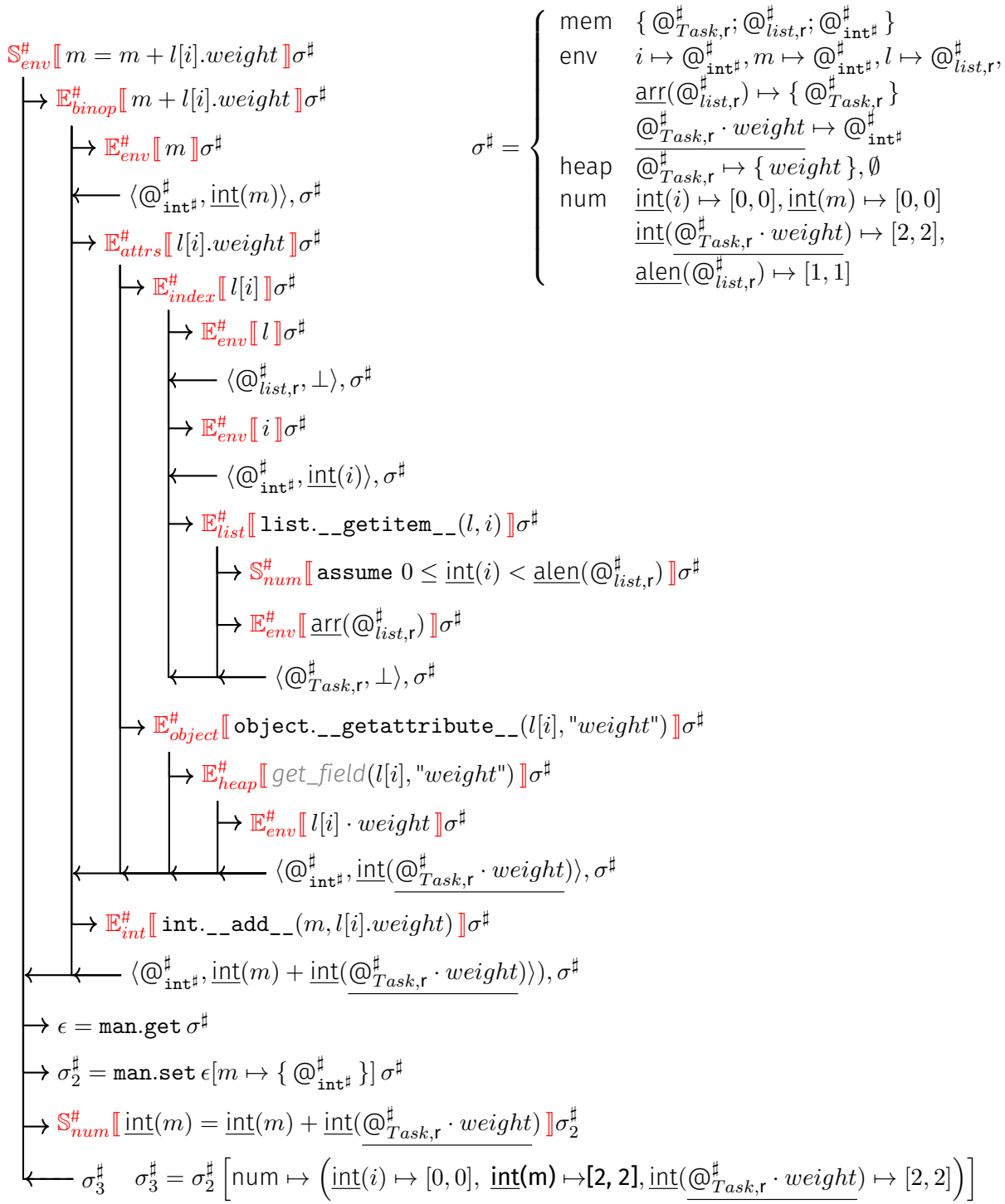

Figure 8.4: Analysis of first unrolling at line 8 of Listing 8.1, assuming – to simplify – that the list `l` has been instantiated with `Task(2)` only

iary address variable $@^\sharp_{Task,\mathsf{r}} \cdot weight$. The evaluation of this variable is then delegated to the environment abstraction. The environment abstraction returns an object, having an integer address $@^\sharp_{\texttt{int}^\sharp}$, and for value the expression $\underline{\text{int}}(@^\sharp_{Task,\mathsf{r}} \cdot weight)$.

- The domain of binary operators ends up calling `int.__add__`. This call is handled by

the domain of `int` objects. It returns the integer address, and the value expression is the addition of the auxiliary integer variables of m and $\underline{@^\sharp_{Task,\mathsf{r}} \cdot weight}$. The $+$ operator corresponds to the addition of mathematical integers and not Python's addition operator.

- The environment domain updates its local binding to map variable m to the integer address.
- It then delegates the assignment $\underline{\text{int}}(m) = \underline{\text{int}}(m) + \underline{\text{int}}(\underline{@^\sharp_{Task,\mathsf{r}} \cdot weight})$ over values to the numerical domain. The result of this statement updates the binding of $\underline{\text{int}}(m)$ to the interval $[2, 2]$.

We encounter an issue to define the concretization of the environment. We have assumed that builtin values are allocated to a single weak address for each type in the previous chapter. In the value analysis, builtin values are tracked through variables to be precise. However, these two approaches conflict in the concretization: we illustrate this issue in the example below.

Example 8.8 **Concretization difficulty with builtin values**

Let us consider the abstract state at the end of the program `x = 1; y = 2`:

$$\sigma^\sharp = \begin{cases} \text{env} & x \mapsto @^\sharp_{\texttt{int}^\sharp}, y \mapsto @^\sharp_{\texttt{int}^\sharp} \\ \text{num} & \underline{\text{int}}(x) \mapsto [1,1], \underline{\text{int}}(y) \mapsto [2,2] \end{cases}$$

If the concretization of the environment puts builtin values in the the heap, all integer values will be merged into the same state, since there is only one weak address for builtin integers. The same applies to floating-point and string values. In that case, we would get:

$$\bigcup_{v \in \{\,1,2\,\}} \left\{ (e,h) \;\middle|\; e(x) = @^\sharp_{\texttt{int}^\sharp}, e(y) = @^\sharp_{\texttt{int}^\sharp}, h(@^\sharp_{\texttt{int}^\sharp}) = (\texttt{int}(v), \text{🔒}) \right\}$$

Clearly, this approach does not work. The recency abstraction will concretize $@^\sharp_{\texttt{int}^\sharp}$ into multiple addresses and provide a sound set of concrete states. This set of states would however be really imprecise (we would get that x may be 2 for example). Our solution consists in forgetting the object structure of builtins for now, and let the environment domain perform the following concretization in this case:

$$\{\, (e, \emptyset) \mid e(x) = 1, e(y) = 2 \,\}$$

Once the concretization of the recency abstraction has been applied, we can always concretize these builtin values into builtin objects to get:

$$\left\{\, (e,h) \;\middle|\; \exists (a,b) \in \mathbb{N}^2, e(x) = @_a, e(y) = @_b, h(@_a) = (\texttt{int}(1), \text{🔒}), h(@_b) = (\texttt{int}(2), \text{🔒}) \,\right\}$$

We show the updated concretization of the environment abstraction in Figure 8.5. This concretization is a relation between a concretized environment over auxiliary value variables (i.e., **Value** $\stackrel{\text{def}}{=} \mathbb{Z} \cup \mathbb{F}_{64} \cup$ *string*) and a concrete environment and a heap. For non-builtin values, the concretization is the same as the one of the type analysis shown in Figure 7.3. Builtin values are picked from the input environment ρ, using the corresponding auxiliary variables.

$$\begin{aligned}\gamma_{env}(\epsilon) = \{\, (\rho \in \mathcal{V} \to \mathsf{Value}), (e \in (\mathcal{V} \rightharpoonup \mathsf{Addr}^\sharp \cup \mathsf{Value}), h \in (\mathsf{Addr}^\sharp \rightharpoonup \mathsf{ObjN} \times \mathsf{ObjS})) \mid \\ v \in \mathsf{dom}\epsilon \Leftrightarrow \Big(e(v) \in \epsilon(v) \setminus \{\, @^\sharp_{\mathtt{int}^\sharp}, @^\sharp_{\mathtt{float}^\sharp}, @^\sharp_{\mathtt{str}^\sharp} \,\} \vee \\ \big(\underline{\mathsf{int}}(v) \in \mathsf{dom}\rho \implies e(v) \in \rho(\underline{\mathsf{int}}(v))\big) \vee \\ \big(\underline{\mathsf{float}}(v) \in \mathsf{dom}\rho \implies e(v) \in \rho(\underline{\mathsf{float}}(v))\big) \vee \\ \big(\underline{\mathsf{str}}(v) \in \mathsf{dom}\rho \implies e(v) \in \rho(\underline{\mathsf{str}}(v))\big)\Big); \\ @^\sharp \in \mathsf{codom}\epsilon \Leftrightarrow (\mathsf{fst} \circ h(@^\sharp) \in \gamma_{\mathsf{ObjN}}(\mathtt{type}(@^\sharp)) \wedge \mathsf{snd} \circ h(@^\sharp) = \top) \,\}\end{aligned}$$

Figure 8.5: Concretization of the environment abstraction

Example 8.9 **Full concretization of the value analysis**

We consider the abstract state $\sigma^\sharp$. It corresponds to the tasks allocated at line 6 of Listing 8.1, along with a variable x pointing to one of the task instances.

$$\sigma^\sharp = \begin{cases} \mathsf{mem} & \{\, @^\sharp_{Task,\mathsf{r}}; @^\sharp_{Task,\mathsf{o}} \,\} \\ \mathsf{env} & x \mapsto \{\, @^\sharp_{Task,\mathsf{r}}; @^\sharp_{Task,\mathsf{o}} \,\}, \\ & \underline{@^\sharp_{Task,\mathsf{r}} \cdot weight} \mapsto @^\sharp_{\mathtt{int}^\sharp}, \quad \underline{@^\sharp_{Task,\mathsf{o}} \cdot weight} \mapsto @^\sharp_{\mathtt{int}^\sharp} \\ \mathsf{heap} & @^\sharp_{Task,\mathsf{r}} \mapsto (\{\, weight \,\}, \emptyset), \quad @^\sharp_{Task,\mathsf{o}} \mapsto (\{\, weight \,\}, \emptyset) \\ \mathsf{num} & \underline{\mathsf{int}}(\underline{@^\sharp_{Task,\mathsf{r}} \cdot weight}) \mapsto [5,5], \quad \underline{\mathsf{int}}(\underline{@^\sharp_{Task,\mathsf{o}} \cdot weight}) \mapsto [1,3] \end{cases}$$

The concretization of the numerical domain yields three different states:

$$\bigcup_{v \in \{\,1,2,3\,\}} \left\{ \underline{\mathsf{int}}(\underline{@^\sharp_{Task,\mathsf{o}} \cdot weight}) \mapsto v, \underline{\mathsf{int}}(\underline{@^\sharp_{Task,\mathsf{r}} \cdot weight}) \mapsto 5 \right\}$$

The concretization of the environment chooses what x points to. It defines the nominal object structure of the objects allocated at $@^\sharp_{Task,\mathsf{m}}$. For each auxiliary integer variable $\int(v)$, a binding of $e(v)$ is created with the corresponding numerical value (now considered as a builtin value of Python).

$$\bigcup_{\substack{v \in \{\,1,2,3\,\}, \\ m \in \{\,r,o\,\}}} \left\{ (e,h) \;\middle|\; \begin{array}{l} e(x) = @^\sharp_{Task,\mathsf{m}} \quad h(@^\sharp_{Task,\mathsf{r}}) = (Task, \top) \quad h(@^\sharp_{Task,\mathsf{o}}) = (Task, \top) \\ e(\underline{@^\sharp_{Task,\mathsf{o}} \cdot weight}) = v \quad e(\underline{@^\sharp_{Task,\mathsf{r}} \cdot weight}) = 5 \end{array} \right\}$$

The concretization of the heap transforms the auxiliary attribute addresses on weight, that are always defined, into structural information on the objects in the heap.

$$\bigcup_{\substack{v \in \{\,1,2,3\,\}, \\ m \in \{\,r,o\,\}}} \left\{ (e,h) \;\middle|\; \begin{array}{ll} e(x) = @^\sharp_{Task,\mathsf{m}} & h(@^\sharp_{Task,\mathsf{r}}) = (Task, \{\, weight \mapsto 5 \,\}) \\ & h(@^\sharp_{Task,\mathsf{o}}) = (Task, \{\, weight \mapsto v \,\}) \end{array} \right\}$$

The recency abstraction concretizes the old address in a finite number of addresses and the recent address into a single address allocated after the old addresses.

$$\bigcup_{n\geq 1}\Big\{\,(e,h)\;\Big|\;\exists j\in[1,n], e(x)=@_j;\qquad h(@_n)=(Task,\{\,weight\mapsto 5\,\});\\ \forall i\in[1,n-1],\exists v_i\in\{\,1,2,3\,\},h(@_i)=(Task,\{\,weight\mapsto v_i\,\})\Big\}$$

As we have mentioned, the concrete state above does not exactly match the definition of concrete states of Chapter 6. The builtin values (5 and the v_i in the state above) have to be transformed into Python objects, giving:

$$\bigcup_{n\geq 1}\Big\{\,(e,h)\;\Big|\;\exists j\in[1,n], e(x)=@_j;\\ h(@_n)=(Task,\{\,weight\mapsto @^a_{\texttt{int}}\,\});\exists a\in\mathbb{N}, h(@^a_{\texttt{int}})=(\texttt{int}(5),\blacksquare);\\ \forall i\in[1,n-1],\exists v_i\in,\exists b_i\in\mathbb{N},\{\,1,2,3\,\},h(@_i)=(Task,\{\,weight\mapsto @^{b_i}_{\texttt{int}}\,\})\wedge\\ h(@^{b_i}_{int})\mapsto(\texttt{int}(v_i),\blacksquare)\Big\}$$

Remark 8.10 **List analysis**

In the current implementation, the list domain handles both summarized content and list length at once. It thus does not change whether a type or a value analysis is used. This differs from the presentation we made in Chapter 5, where summarization and length domains are combined in a reduced product (see also Remark 5.17 in that chapter). The approach based on a reduced product would, in theory, be better: the length domain would not be needed in the case of a type analysis. In addition, the implementation would be smaller and more modular. We leave this modular implementation as future work.

8.2 Experimental evaluation

This section mainly presents the results obtained in the publication of our SOAP article [112]. We compare the type analysis and the value analysis in Section 8.2.1, the choice of allocation policy in Section 8.2.2, and the impact of the abstract garbage collection (AGC) in Section 8.2.3. We finish by showing new results about the selectivity of the value analysis.

The configuration of the value analysis we used relies on an interval domain for integers and floating-point number abstractions, and on a string powerset domain for the string abstraction. It is shown in Figure 8.6. All the analyses shown in section use the function cache defined in Section 7.5.2.2.

8.2.1 Value-sensitivity

The results of the type and value analyses are displayed in Figure 8.7. Both analyses were performed with no allocation sensitivity, and the AGC active. The memory is measured through OCaml's garbage collector statistics, using the maximum size reached by the major heap. We print the reductions in the number of detected exceptions in bold. The exceptions detected are split into different categories:

- Type errors: `TypeError, AttributeError` exceptions.
- Index errors: out-of-bound list accesses.
- Key errors: key not found during dictionary access.
- Math errors: overflows, divisions by zero.
- Value errors: when an iterable is unpacked in too many values.

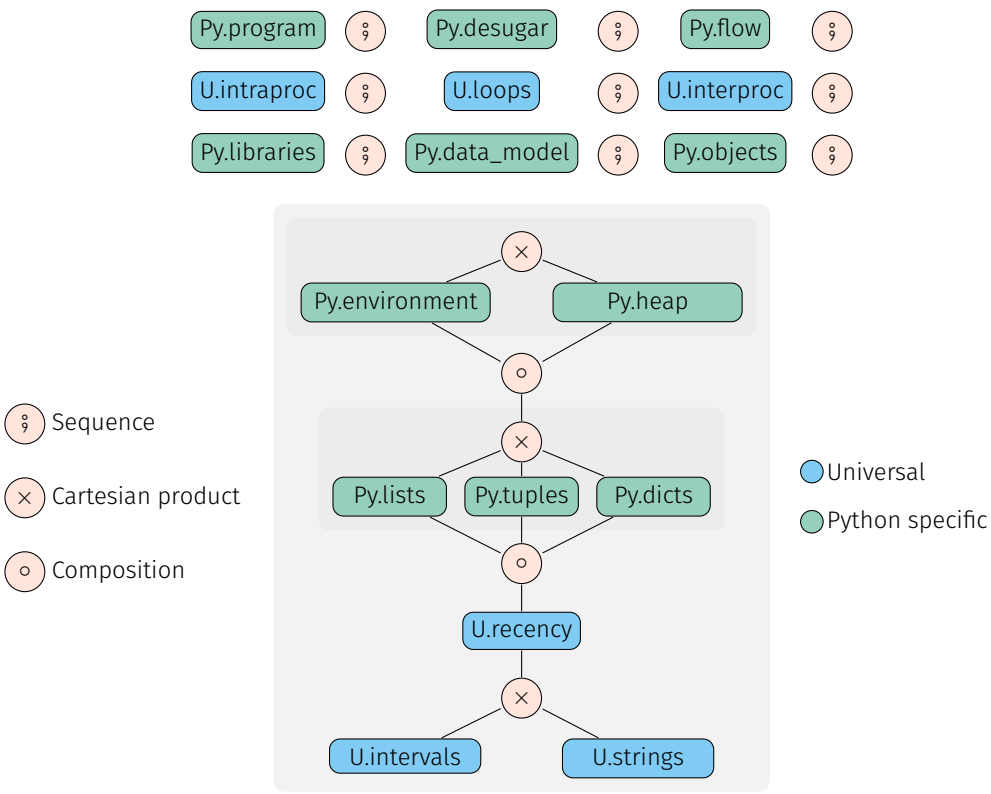

Figure 8.6: Configuration of the value analysis (to be compared with Figure 7.10)

- All other errors, including user-defined exceptions.

Some exceptions are systematically raised by the type analysis in order to be sound (such as index errors during list accesses). We included these as alarms in the table to show how many potential errors are eliminated by the value analysis. We notice that the precision gained with the value analysis does not remove any type-related error in the programs we analyze. We find that the value analysis is in average 3.25 times slower than the type analysis and similarly, it needs 3 times more memory. In some rare cases, the value analysis is able to detect dead code that the type analysis considers reachable. The value sensitivity does not reduce the key errors (the smashing abstraction of dictionaries is too coarse). For all other exception categories, an improvement in precision is witnessed. In the case of the index errors, the number of raised alarms is divided by 10.

8.2.2 Allocation-site policy choice

We perform an allocation-site sensitivity comparison in Figure 8.8. We compare having no allocation-site sensitivity for all objects (excepting containers) with having allocation-site sensitivity for all objects. The abstract garbage collector is activated in both cases. For each allocation policy, the time spent, the memory used, and the number of exceptions detected are indicated. The best results for each policy are printed in bold. We find that not performing an allocation-site partitioning is more efficient, although it may raise a few more alarms – less than 9% more in total. Cases such as `chaos.py` and `regex_v8.py` illustrate this observation. The analysis of `richards.py` reveals that the addition of the location sensitivity may cost an order of magnitude more in resources. This cost increase is a consequence of objects of the same type being allocated at different program locations, resulting in more cases to be analyzed. In some cases (`go.py`, `hexiom.py`), the location sensitivity yields quicker analyses, because the analysis without location sensitivity performs more weak updates (when allocat-

Name	LOC	Type Analysis								Value Analysis							
		Time	Mem.	Exceptions detected						Time	Mem.	Exceptions detected					
				Type	Index	Key	Math	Value	Other			Type	Index	Key	Math	Value	Other
fannkuch.py	59	0.32s	3MB	0	9	0	3	0	0	0.63s	3MB	0	**4**	0	**0**	0	0
float.py	63	0.19s	3MB	0	2	0	8	0	0	0.32s	3MB	0	**0**	0	**3**	0	0
spectral_norm.py	74	0.70s	6MB	0	0	0	9	0	1	1.7s	15MB	0	0	0	**3**	0	**0**
nbody.py	157	1.5s	3MB	0	22	1	11	5	1	5.7s	9MB	0	**1**	1	**1**	**0**	**0**
chaos.py	324	7.4s	42MB	0	28	0	54	10	0	30s	197MB	0	**18**	0	**4**	**8**	0
raytrace.py	411	14s	74MB	5	0	0	43	1	1	27s	171MB	5	0	0	**22**	1	**0**
scimark.py	416	1.4s	12MB	1	1	0	23	0	0	3.4s	27MB	1	**0**	0	**3**	0	0
richards.py	426	13s	112MB	1	4	0	2	1	1	17s	149MB	1	**2**	0	**0**	1	1
unpack_seq.py	458	8.3s	7MB	0	0	0	0	400	0	9.4s	6MB	0	0	0	0	**0**	0
go.py	461	27s	345MB	33	20	0	11	0	0	2.0m	1.4GB	33	20	0	**4**	0	0
hexiom.py	674	1.1m	525MB	0	46	3	0	2	3	4.7m	3.2GB	0	**21**	3	0	**1**	**2**
regex_v8.py	1792	23s	18MB	0	2053	0	0	0	0	1.3m	56MB	0	**145**	0	0	0	0
processInput.py	1417	10s	64MB	7	7	1	2	1	2	12s	85MB	7	**4**	1	**0**	1	2
choose.py	2562	1.1m	1.6GB	12	22	7	19	18	7	2.9m	3.7GB	12	**13**	7	**11**	18	7
Total	9294	4.0m	2.8GB	59	2214	12	185	438	16	13m	9.1GB	59	228	12	51	30	12

Figure 8.7: Value-sensitivity Comparison (no allocation sensitivity, with AGC)

Name	No loc. sensitivity			Loc. sensitivity		
	Time	Mem.	⚠	Time	Mem.	⚠
`fannkuch.py`	0.63s	3MB	4	0.63s	3MB	4
`float.py`	**0.32s**	3MB	3	0.39s	3MB	3
`spectral_norm.py`	1.7s	15MB	3	1.7s	15MB	3
`nbody.py`	5.7s	9MB	3	**5.0s**	**9MB**	3
`chaos.py`	**30s**	**197MB**	30	2.4m	1.2GB	**15**
`raytrace.py`	27s	171MB	28	**4.5s**	**74MB**	**7**
`scimark.py`	3.4s	27MB	4	**3.0s**	27MB	**3**
`richards.py`	**17s**	**149MB**	5	69m	15GB	5
`unpack_seq.py`	**9.4s**	6MB	0	9.6s	6MB	0
`go.py`	2.0m	1.4GB	57	**1.7m**	**1.2GB**	57
`hexiom.py`	4.7m	3.2GB	27	**4.2m**	3.2GB	27
`regex_v8.py`	**1.3m**	**56MB**	145	3.6m	85MB	145
`processInput.py`	12s	85MB	15	**11s**	**74MB**	**13**
`choose.py`	**2.9m**	**3.7GB**	68	3.1m	4.3GB	**63**
Total	13m	9.1GB	392	87m	25GB	359

Figure 8.8: Allocation-site Comparison (value analysis & AGC)

Name	Without AGC		With AGC		Rel. Impr.	
	Time	Mem.	Time	Mem.	Time	Mem.
`fannkuch.py`	0.65s	3MB	0.63s	3MB	4%	0%
`float.py`	0.37s	3MB	0.39s	3MB	-4%	0%
`spectral_norm.py`	1.7s	18MB	1.7s	15MB	0%	17%
`nbody.py`	5.5s	10MB	5.0s	9MB	9%	10%
`chaos.py`	2.7m	1.4GB	2.4m	1.2GB	9%	13%
`raytrace.py`	6.5s	112MB	4.5s	74MB	31%	34%
`scimark.py`	3.1s	32MB	3.0s	27MB	5%	16%
`richards.py`	75m	20GB	69m	15GB	8%	24%
`unpack_seq.py`	13s	7MB	9.6s	6MB	24%	14%
`go.py`	2.3m	1.8GB	1.7m	1.2GB	25%	34%
`hexiom.py`	6.5m	7.5GB	4.2m	3.2GB	36%	57%
`regex_v8.py`	8.4m	345MB	3.6m	85MB	58%	75%
`processInput.py`	12s	74MB	11s	74MB	7%	0%
`choose.py`	4.0m	6.5GB	3.1m	4.3GB	24%	34%
Total	99m	38GB	85m	25GB	15%	34%

Figure 8.9: AGC Comparison (value analysis, with location sensitivity)

ing an address that already exists through the recency abstraction), which are costlier than having multiple recent addresses in these cases. We tested adding the callstack-sensitivity to the location-sensitivity. In our benchmarks, it achieved the same results in precision and similar analysis times.

8.2.3 Abstract garbage collector

We show the impact of the abstract garbage collector in Figure 8.9. We only show time and memory usage, as the precision is unchanged in those cases. If we remove the pathological case of `richards.py`, whose analysis time is significantly higher than the others, the AGC yields

a 33% relative improvement for the analysis time and 44% for the memory usage. These results are similar to the ones obtained with the type analysis in Section 7.6.3.

8.2.4 Selectivity of the analysis

We show the selectivity of the non-relational value analysis, with no location sensitivity and the abstract garbage collector, on our benchmarks in Figure 8.10. Given a check (in our case, a class of Python exception), the selectivity is the number of operations proved safe by our analysis on this check, divided by the total number checks performed by the analysis. Mopsa provides utilities to tag expressions as safe or unsafe with respect to a check, and stores the results in the abstract state (Remark 3.14). This addition is rather recent, so the selectivity was not presented in our SOAP or in our ECOOP papers [110, 112]. We consider the selectivity with respect to the following Python exceptions:

- `AttributeError` (invalid attribute accesses),
- `TypeError` (invalid type operations),
- `IndexError` (out-of-bound list accesses),
- `KeyError` (invalid dictionary accesses),
- `ValueError` (invalid tuple unpacking),
- `OverflowError` (during casts from integers to float for example), and
- `DivisionByZeroError`.

We highlight in bold font cases where the selectivity is not 100%. Empty cells correspond to a program where a kind of exception cannot happen: for example, `fannkuch.py` does not manipulate dictionaries, so its cell for key errors is left empty. We can see that the selectivity of the analysis is really high for attributes and types. They are also a high number of checks in this category, since Python is a dynamic language, and we have to infer and check the types during the analysis. The smashing abstraction of lists and dictionaries are not really precise and have a low selectivity. The selectivity is good for the last three categories, except for some cases. For example, `raytrace.py` has a reduced selectivity with respect to overflow exceptions. It performs computations on a class of vectors. In that case, the absence of location-sensitivity means that different vectors will be summarized in the same old address, creating imprecise values in their coordinates and triggering the false alarms.

8.3 Scaling relational analyses using packing

The experimental evaluation of Section 8.2 relies on the non-relational interval numerical domain. A relational analysis using vanilla abstract domains of Apron [73] does not scale well however: even a small program of around 150 lines of code such as `nbody.py` takes 53 minutes to be analyzed using the octagons abstract domain (and did not terminate in 8 hours with the polyehdra domain). This is due to the high number of variables – both program variables and the auxiliary variables introduced: at the end of the execution, the octagon has 75 dimensions. As we mentioned in Figure 2.15, the computational cost of operations is cubic in the number of variables for octagons and exponential for polyhedra. One way to alleviate this problem is to use packing techniques [11]. The idea of packing is to split a relational domain over a high number of variables – having a high computational cost – into a set of relational domains working on small groups of related variables. The domain will infer relationships between variables inside the same group, but not between variables in different groups. This approach thus reduces the computational complexity of the numerical operations.

We have implemented a packing heuristic for the value analysis. This packing is static, i.e., it is only defined by syntactic information on the program. It keeps all global variables in a same pack. One pack is created for each function defined in the analyzed program. These function packs keep the corresponding parameters, local variables, and return variable. Auxiliary

address variables related to lists are packed within the function allocating the list. The same applies to auxiliary variables of `range` objects, used a lot to iterate in loops. Other auxiliary variables are not kept in the packs for now.

The relational analysis with packing is more scalable. In the case of `nbody.py`, we go from an analysis taking 53 minutes with octagons (having 75 dimensions at the end of the program), to an analysis completing in 37 seconds. In that case, there are five different packs at the end of the program, corresponding to octagons having 14, 3, 8, 3 and 8 dimensions. The packing also improves the precision a lot in the case of `regex_v8.py`: it is able to rule out all 145 index errors found by the value analysis (Figure 8.7). The analysis is 2.5 times slower that the non-relational one. In our preliminary experiments, we did not notice significant precision improvements compared to the non-relational value analysis for the other benchmarks.

Remark 8.11 **The balancing act of static packing**

The static packing provides a first heuristic, which comes with a few downsides. On the one hand, it can be imprecise: it is not possible to expression relationships between variables in different packs[1]. This may reduce the precision of the analysis. On the other hand, some packs may be unnecessarily big, and thus computationally costly. They could be split into smaller packs, preferably by using a dependency analysis as a guide to create the packs.

Remark 8.12 **Static packing and Mopsa's loosely-coupled domains**

The current approach of our static packing does not respect Mopsa's approach of designing loosely-coupled domains: it needs to know which auxiliary variables are created by each abstract domain used in the analysis to perform the packing. Creating more adaptative approaches to variable packing is left as future work.

Name	Attributes	Types	Indexes	Keys	Values	Overflows	Divisions
`fannkuch.py`	57/57	90/90	7/11		3/3		3/3
`float.py`	88/88	104/104	3/3		1/1	9/9	**4**/7
`spectral_norm.py`	63/63	84/84			3/3	**7**/8	2/2
`nbody.py`	184/184	285/285	**22**/23	**0**/1	3/3	6/6	**5**/6
`chaos.py`	622/622	674/674	**22**/40		**10**/14	**40**/41	**15**/26
`raytrace.py`	573/573	**638**/643	2/2		**5**/6	**24**/44	**7**/9
`scimark.py`	746/746	844/844	**2**/5		**29**/30	**21**/43	**20**/21
`richards.py`	**352**/353	389/389	**2**/4		**2**/3		2/2
`unpack_seq.py`	807/807	1210/1210			1/1		
`go.py`	**664**/697	728/728	**2**/20		7/7	**6**/12	**4**/6
`hexiom.py`	598/598	672/672	**10**/32	**0**/3	**23**/24		
`regex_v8.py`	7357/7357	8349/8349	**1913**/2057		63/63		
`processInput.py`	**617**/619	**790**/792	12/12	**0**/1	**0**/1	2/2	
`choose.py`	**2519**/2521	**2997**/2999	**28**/39	**4**/8	**9**/24	**7**/17	

Figure 8.10: Selectivity of the analysis on some classes of exceptions
Selectivity = Number of proved safe operations / Total number of checks
An empty cell denotes a program where the kind of exception cannot happen

[1] In practice, some variables can be added in different packs, and their values reduced by communicating their intervals. For example, the parameters of functions can be added in the packs of the caller and the callee, to keep relations between the two.

8.4 Conclusion

This concludes the part on the analysis of pure-Python programs. This chapter defined a value analysis, acting as a refinement of the type analysis. We have shown in our experimental evaluation that this analysis is more costly than the type analysis, but it is also more precise, and raises less false alarms. The analysis can use an underlying relational numerical abstract domain, which needs to be packed into smaller parts to allow scaling. We have proposed a simple static packing, and shown it can reduce further the number of alarms in one of our benchmarks. To improve the precision of our analysis, we believe the most important point is the development of more precise abstractions for the analysis of dictionaries. Having a better packing heuristic, or leveraging recent work on efficient relational abstract domains [137, 138, 139, 60], is left as future work. The two most pressing points to improve the scalability of the analysis are:

- to reduce the cost of analyzing functions called in multiple different contexts, through the use of function summaries,
- to support the vast standard library of Python, and other popular third-party libraries.

The last point is the object of the next part: a significant number of third-party libraries are written in C, and we define an analysis that is able to multilanguage programs consisting of Python and C code.

Part IV

Mixing Python and C

9

Interoperability Mechanisms between Python and C

Modern programs are increasingly multilanguage. This allows developers to combine the strengths of different languages and reuse libraries written in other languages. A *host* language may call a *guest* language through an interface; this interface is also called a *boundary*. The guest language is frequently C and is usually referred to as native code or native C. In this thesis, the host language is Python, and the guest language is C. This work supports the Python/C API [153] as the interoperability mechanism between Python and C. Using native C modules in Python is frequent as it allows writing high-level Python code, itself calling efficient C code. As a matter of fact, one in five of the 200 most downloaded Python libraries available on GitHub contains C code.

We start by showing a self-contained motivating example in Section 9.1, giving insights on how native C modules are defined and how they work with Python. We compare the Python/C API with other available interoperability mechanisms in Section 9.2.

9.1 A toy example using Python's API

This section provides an in-depth motivating example. We show how to define a native C extension module and how it can be used by a Python client code. We end the section by discussing which errors may happen.

When developers want to run native C code in Python, they can define native C extension modules using the Python API. These modules may contain attributes, methods, and classes, just as any other Python module. However, these methods and classes are now written in C. API functions are denoted by the `Py` prefix (and written in magenta in the listings). The semantics of some of these functions are described formally in Chapter 10.

9.1.1 Counter module, viewed from Python

Our example is a C module defining a `Counter` class, alongside some client code in Python. This example is self-contained and shown in Listings 9.1 and 9.2. From a high-level point of view, the `counter` module defines a `Counter` class. Instances of `Counter` can be created (`count.py`, line 4); their internal counter can be incremented using the `incr` method, which takes an optional integer argument being the increment (lines 6-7); they also have a read-only attribute `counter` returning their current value (line 8).

Listing 9.1: Contents of file `count.py`

```
import counter
import random

c = counter.Counter()
p = random.randrange(128)
c.incr(2**p-1)
c.incr()
r = c.counter
```

Listing 9.2: Contents of file `counter.c`

```
#include <Python.h>
#include "structmember.h"

typedef struct {
  PyObject ob_base;
  int count;
} CounterO;

static PyObject*
CounterIncr(CounterO *self, PyObject *args)
{
  int i = 1;
  if(!PyArg_ParseTuple(args, "|i", &i))
    return NULL;
  self->count += i;
  Py_RETURN_NONE;
}

static int
CounterInit(CounterO *self, PyObject *args,
            PyObject *kwds)
{
  self->count = 0;
  return 0;
}

static PyMethodDef CounterMethods[] = {
 {"incr", (PyCFunction) CounterIncr,
  METH_VARARGS, ""}, {NULL}
};
static PyMemberDef CounterMembers[] = {
 {"counter", T_INT, offsetof(CounterO,
   count), READONLY, ""}, {NULL}
};

static PyType_Slot CounterTSlots[] = {
  {Py_tp_new, PyType_GenericNew},
  {Py_tp_init, CounterInit},
  {Py_tp_methods, CounterMethods},
  {Py_tp_members, CounterMembers}, {0, 0}
};

static PyType_Spec CounterTSpec = {
  .name = "counter.Counter",
  .basicsize = sizeof(CounterO),
  .itemsize = 0,
  .flags = Py_TPFLAGS_DEFAULT
         | Py_TPFLAGS_BASETYPE,
  .slots = CounterTSlots
};

static struct PyModuleDef countermod = {
 PyModuleDef_HEAD_INIT, .m_name = "counter",
  .m_methods = NULL, .m_size = -1
};

PyMODINIT_FUNC
PyInit_counter(void)
{
  PyObject *m =PyModule_Create(&countermod);
  if(m == NULL) return NULL;
  PyObject* CounterT =
    PyType_FromSpec(&CounterTSpec);
  if(CounterT == NULL || PyModule_AddObject(
       m, "Counter", CounterT) < 0) {
    Py_DECREF(m);
    return NULL;
  }
  return m;
}
```

Listing 9.3: Some Python's API headers

```
typedef struct PyObject {
  Py_ssize_t ob_refcnt;
  struct PyTypeObject *ob_type;
} PyObject;

typedef PyObject *(*PyCFunction)
  (PyObject *, PyObject *);
typedef int (*initproc)
  (PyObject *, PyObject *, PyObject *);

typedef struct PyMethodDef {
 const char  *ml_name; PyCFunction ml_meth;
 int  ml_flags; const char  *ml_doc;
} PyMethodDef;

typedef struct PyMemberDef {
 const char *name; int type;
 Py_ssize_t offset; int flags;
 const char *doc;
} PyMemberDef;

typedef struct PyTypeObject {
  PyObject ob_base;
  const char *tp_name;
  Py_ssize_t tp_basicsize;
  Py_ssize_t tp_itemsize;
  unsigned long tp_flags;
  struct PyMethodDef *tp_methods;
  struct PyMemberDef *tp_members;
  struct PyTypeObject *tp_base;
  PyObject *tp_dict;
  initproc tp_init;
  newfunc tp_new;
} PyTypeObject;
```

9.1.2 Counter, viewed from C

In C, instances of `Counter` will be stored using the `CounterO` `struct`. This `struct` starts with a `PyObject ob_base` field. All Python objects are represented as `PyObject`s in C. Putting the `PyObject` as the first field in the `Counter` structure allows casting to and from Python objects.[1] The `PyObject` definition is part of the API and shown in Listing 9.3. Its fields are a reference counter for the garbage collector and a pointer to the class to which it belongs. `PyTypeObject` is Python's `type` object, from which all classes derive. The second field of `CounterO` is the instance's data: an integer `count`, not directly exposed to Python.

The `Counter` class' specification is defined lines 51-58. It has three methods stored in the `Py_tp_new, Py_tp_init`, and `Py_tp_methods` fields. It also defines a special attribute member `counter` lines 40-41. The `PyTypeObject` structure is synthesized from the specification by `PyType_FromSpec` (line 71). These methods and members are lifted to become Python attributes and methods when the class is initialized (in `PyType_FromSpec`, cf. Section 9.1.4). Other fields are defined in the `PyTypeObject` structure. `tp_basicsize, tp_itemsize` define the size (in bytes) of instances. `tp_flags` is used to perform fast instance checks for builtin classes. `tp_base` points to the parent of the class. `tp_dict` is the class' dictionary used by Python to resolve attribute accesses (created during class initialization).

9.1.3 Module import

When executing the `import counter` statement, the C function `PyInit_counter` is called. This function starts by creating a module whose name (line 61) is `counter` with no methods attached to it (line 62). Then, the `CounterT` class is created (lines 71-72, cf. Section 9.1.4), and the class is bound to the module (lines 72-73). Python uses a reference-counting-based garbage collector, which has to be manually handled using the `Py_INCREF` and `Py_DECREF` macros in C. If no errors have been encountered, the module object is returned at the end of the function, and `counter` will now point to this module in Python.

9.1.4 Class initialization

The call to `PyType_FromSpec` creates the `Counter` class from its specification. The function `PyType_FromSpec` starts by creating the `PyTypeObject` and fills its fields according to the specification. This structure is then passed to `PyType_Ready` which populates the `tp_dict` field of the class. This field is the class' dictionary used by Python to resolve attribute accesses. Before this call, the attribute `counter` and the methods `__new__`, `__init__`, and `incr` do not exist on the Python side.

We explain how these C functions are encapsulated into Python objects by `PyType_Ready`.

- ▷ **`__new__` method.** The `__new__` field points to a `builtin_function` object. When this object is called, it calls `tp_new_wrapper`, which is a C function defined by CPython. This function in turn will call `PyType_GenericNew` to allocate the instance.
- ▷ **`__init__` method.** The prototype of C functions for Python is `PyCFunction` (Listing 9.3, line 6). Some signature adaptations may be needed for specific kinds of functions. For example, initialization methods (such as `CounterInit`) return a C `int` by convention. Thus, `CounterInit` will be wrapped into a function called `wrap_init`, which behaves as a `PyCFunction`. It is then encapsulated into a builtin Python descriptor object. Upon a call to this object, this descriptor performs pre- and post-call checks (described in Chapter 10). Continuing our example, `wrap_init` will be stored into an instance of the builtin `wrapper_descriptor` object. This descriptor is then added to the class' dictionary.

[1] According to the ISO C reference, "a pointer to a structure object, suitably converted, points to its initial member, and vice versa".

▷ **`incr` method.** The C function `CounterIncr` has a signature abiding by the `PyCFunction` type. There is no need for a specific wrapper similar to what `CounterInit` required. However, `incr` is a method, and its assumes that its first argument is a `Counter` instance. This behavior is obtained by wrapping the C function `CounterIncr` into a `method_descriptor` object, performing the instance check on the first argument when it is called.

▷ **`counter` attribute.** The member declaration at lines 40-41 of the running example (Listing 9.2) is used to create a `member_descriptor` object, which is added to the class' dictionary at the `counter` field. This object will behave to make `counter` appear to be an attribute. However, access to the attribute will call the `__get__` method of this member descriptor, and setting the attribute's value will call the `__set__` method (cf. Figure 6.31 and Example 6.31). `__get__` is method of the `member_descriptor` object which synthesizes the Python integer from the C integer at `offsetof(CounterO, count)` (Listing 9.2, lines 40-41). `__set__` raises an exception when it called, since the member declaration specifies that `counter` is `READONLY`.

Figure 9.1 shows a summary of these encapsulation. The next chapter includes a schematic representation of the state in Figure 10.5 and Remarks 10.9 to 10.11 discuss in details the different kind of descriptors.

Attribute	Encapsulating Object	Underlying Wrapper	Underlying C definition
`__new__`	`builtin_function`	`tp_new_wrapper`	`PyType_GenericNew`
`__init__`	`wrapper_descriptor`	`wrap_init`	`CounterInit`
`incr`	`method_descriptor`	∅	`CounterIncr`
`counter`	`member_descriptor`	∅	`CounterMembers[0]`

Figure 9.1: Python `Counter` structure summary

9.1.5 Counter creation

When a new instance of `Counter` is created (line 4), Python starts by allocating it by calling `Counter.__new__`. This call will eventually be resolved into `PyType_GenericNew` (from `tp_new`), allocating the object and initializing the necessary fields (such as `ob_refcnt` and `ob_type` of `PyObject`). Then, `Counter.__init__` is called and the C function `CounterInit` (lines 27-33) ends up being called. It initializes the `count` field of the `CounterO` `struct` to 0. We show how this counter creation is evaluated in the abstract in Example 11.3.

9.1.6 Counter increment

When the `incr` function is called, it is resolved through Python's attribute accesses into `Counter Incr`. `CounterIncr` uses the standard Python function prototype, corresponding to `PyCFunction` (Listing 9.3). Its first argument is the object instance (here, the instance stored in variable `c`), and the second argument is a tuple of the arguments passed to the function (for the call at line 6 it is a tuple of length one containing `2**p-1`, and an empty one for the second call at line 7). `PyArg_ParseTuple` is a helper C function from the Python API converting the Python arguments wrapped in the tuple into C values.[2] It uses a format string to describe the conversion. The `|` character separates mandatory arguments from the optional ones, while `i` signals a conversion from a Python integer to a C `int`. Internally, the conversion is done from a Python integer to a `long` (which may fail with an exception since Python integers are unbounded), which is then cast to an `int` if size permits (otherwise, another exception is set). In the first call to `CounterIncr` (line 6), `i` will be assigned `2**p-1` if the conversion is successful. In the

[2] `Py_BuildValue` is the converse function translating C values to Python ones.

second call, `i` will keep its default value, `1`. The internal value of the counter is then incremented by `i`, and then Python's `None` value is returned.

9.1.7 Counter access

Thanks to the complex semantics of Python, attribute accesses can actually hide calls to custom getter and setter functions through data descriptors (cf. Figure 6.31 and Example 6.31). In our case, `PyType_Ready` takes the member declaration lines 39-42, and creates those custom getters and setters through a `member_descriptor` builtin object. The access to attribute `counter` at line 8 calls the getter (i.e., the `__get__` method) of this `member_descriptor` object. This getter accesses the `count` field of the `CounterO` `struct` and converts the C integer into a Python integer. The `READONLY` flag in the declaration ensures that any call to the setter function raises an exception. These member descriptors are supported by our multilanguage semantics (cf. Remark 10.11) and our multilanguage analysis (cf. Example 11.4).

9.1.8 Building the module

In order for a native extension module to be accessible in Python, it has to be compiled into a dynamically linked shared object library. The easiest approach is to use the `setuptools` library to perform the build, using a `setup.py` file shown in Listing 9.4. This script will call GCC to create a shared library for the counter module.

Listing 9.4: Example `setup.py` build file

```
1 from setuptools import setup, Extension
2
3 module = Extension('counter', sources=['counter.c'])
4
5 setup(name="counter", ext_modules=[module])
```

9.1.9 What can go wrong?

Depending on the chosen value of `p` the result `r` will range from

(i) the expected value ($\mathtt{r} = 2^{\mathtt{p}}$ when $0 \leq \mathtt{p} \leq 30$),
(ii) conversion errors from a Python integer to a C integer, raised as `OverflowError` exceptions. The error message depends on whether $p \geq 64$. As we have mentioned in Section 9.1.6, the integer conversion is done in two steps: the Python integer is first converted to a C `long`. This conversion fails when $p \geq 64$, and the error message is then "Python int too large to convert to C long" Otherwise, the conversion to a C `long` works, but the interpreter then checks that the cast is also possible. When $31 < p < 64$, the cast is not possible and the error message is "signed integer is greater than maximum".
(iii) a silent integer overflow on the C side, causing a wrap-around which is an unexpected behavior for Python developers ($\mathtt{r} = -2^{31}$ for $\mathtt{p} = 31$).

All these errors are due to different representations between the builtin values of the language. The C integer overflow does not interrupt the execution. This motivates the creation of an analysis targeting all kinds of runtime errors in both host and guest languages as well as at the boundary. The analysis presented in Chapter 11 infers all reachable C and Python values, as well as raised exceptions in order to detect these runtime errors. In this example, our analyzer is able to detect all these cases. Our analyzer is also able to prove that the program is safe when `p` ranges between 0 and 30.

9.1.10 Common bugs at the boundary

We refer the reader to the work of Hu and Zhang [70] for an empirical evaluation of bugs at the boundary between Python and C. The most frequent bugs happening at the boundary between the languages are:

- mismatches between a returned `NULL` and the exception being set in C (`NULL` should be returned by Python C functions if and only if an exception has been set – cf. Figure 10.8),
- mismatches between the C and Python datatypes during conversion (in calls to `PyArg_ParseTuple`, `PyLong_FromLong`),
- integer overflows during conversions from arbitrary-precision Python integers to C,
- reference-counting errors.

The multilanguage analysis of Chapter 11 supports all classes of bugs, apart from the reference-counting errors.

9.2 Other Python/C interoperability mechanisms

This section describes other interoperability mechanisms that can be used in Python. They are higher-level than the Python API, and all are based on it one way or another. Thus, a static analysis handling the Python/C API interoperability should be able to handle programs using the mechanisms described in this section with little modifications.

For the sake of simplicity, we consider the case where we want to call a C function from Python. It is also possible to handle structures, as we have done in Section 9.1. We assume we have a file `syracuse.c` (Listing 9.5), defining a function `syrac_nth`. Given an initial element u_0, `syrac_nth(`u_0`, n)` computes the n-th term of the $3n+1$ sequence (similar to the code shown in Listing 2.1).

Listing 9.5: Contents of `syracuse.c` file

```c
int syrac_nth(int u0, int n)
{
    int u = u0;
    int i = 0;
    while(i < n) {
        if(u % 2 == 0) { u = u / 2; }
        else { u = 3 * u + 1; }
        i = i + 1;
    }
    return u;
}
```

9.2.1 Ctypes

`ctypes` [149] is a module from Python's standard library. It allows to call functions from libraries and manipulate C datatypes within Python. For example, let us assume that our file `syracuse.c` has been compiled into the library `libsyracuse.so`. The functions of the library can be dynamically loaded and called, as shown in Listing 9.6. The `ctypes` module is loaded, and the `CDLL` class is used to load the library. We can then call the function `syrac_nth`. By default, `ctypes` assumes that functions return a C integer, and converts it as a Python integer. In our case, `r` is thus a Python integer having value 827370449. As the computation is performed in C, overflows can silently occur. For example, calling `_syracuse.syrac_nth(113383, 120)` in Python returns -1812855948 instead of $827370449 * 3 + 1 = 2482111348$.

Listing 9.6: Ctypes example

```
1 import ctypes
2
3 _syracuse = ctypes.CDLL("libsyracuse.so")
4
5 r = _syracuse.syrac_nth(113383, 119)
```

By default, `ctypes` does not have any information on the types of the functions called. It is thus possible to call `_syracuse.syrac_nth("abc", 120)` for example. The user can declare the types of functions exposed by the library, and `ctypes` will then check that types are correct. For example, if we declare that both arguments of `syrac_nth` have a C integer type through:

$$\texttt{\_syracuse.syrac\_nth.argtypes} = [\texttt{ctypes.c\_int}, \texttt{ctypes.c\_int}]$$

The previous call with the chain "abc" would raise a `ctypes.ArgumentError` exception.

Programs using `ctypes` are very dynamic compared to the other interoperability mechanisms: `ctypes` uses a library that is already compiled and loads it directly during the execution. `ctypes` is implemented using Python's API, and relies on `libffi`. This would be the most difficult framework to support for this reason.

9.2.2 Cffi

The `cffi` [129] library was created by Pypy developers. Compared to `ctypes`, `cffi` is able to read function declarations and automatically generate wrappers around these functions, ensuring safe type conversions. Similarly to `swig` and `cython`, it requires a build pass invoking a C compiler to work. The code needed to use our `syrac_nth` function is shown in Listing 9.7. The execution of this script will create a C file `_syracuse_cffi.c`, containing the definition of the Python C function `syrac_nth` (shown in Listing 9.8), which wraps around the call to the initial function. This script will then combine this C file with `syracuse.c` using `gcc`, to create a shared library that Python can open as a module.

Listing 9.7: Cffi example

```
1  from cffi import FFI
2
3  ffi = FFI()
4
5  ffi.set_source("_syracuse_cffi",
6                 """
7                 #include "syracuse.h"
8                 """, sources=["syracuse.c"])
9
10 ffi.cdef("int syrac_nth(int, int);")
11
12 if __name__ == "__main__":
13     ffi.compile(verbose=True)
```

It is then possible to import the module using:

```
from _syracuse_cffi import lib
```

Calls such as `lib.syrac_nth(113383, 120)` can then be performed. In that particular case, we have seen that the C computation overflows and returns -1812855948. `cffi` is well-behaved concerning types. It checks that the arguments have the correct type. In our example, the function `_cffi_to_c_int` called at line 14 of Listing 9.8 will raise a `TypeError` when `lib.syrac_nth ('a', 0)` is called. The wrappers also check that those argument conversions will fit in their equivalent C type. As such, if we call `lib.syrac_nth(2**32, 0)`, an `OverflowError` will be raised by `_cffi_to_c_int`, as 2^{32} does not fit into a 32-bit C int.

Listing 9.8: Cffi generated code for syracuse example

```
1  static PyObject *
2  _cffi_f_syrac_nth(PyObject *self, PyObject *args)
3  {
4    int x0;
5    int x1;
6    int result;
7    PyObject *pyresult;
8    PyObject *arg0;
9    PyObject *arg1;
10
11   if (!PyArg_UnpackTuple(args, "syrac_nth", 2, 2, &arg0, &arg1))
12     return NULL;
13
14   x0 = _cffi_to_c_int(arg0, int);
15   if (x0 == (int)-1 && PyErr_Occurred())
16     return NULL;
17
18   x1 = _cffi_to_c_int(arg1, int);
19   if (x1 == (int)-1 && PyErr_Occurred())
20     return NULL;
21
22   Py_BEGIN_ALLOW_THREADS
23   _cffi_restore_errno();
24   { result = syrac_nth(x0, x1); }
25   _cffi_save_errno();
26   Py_END_ALLOW_THREADS
27
28   (void)self; /* unused */
29   pyresult = _cffi_from_c_int(result, int);
30   return pyresult;
31 }
```

The analysis of Python code calling the `cffi` generated code should essentially come for free using the Python/C API analysis of Chapter 11, as it generates static C code similar to what we presented in section 9.1, that we already handle. We would have to support the `cffi` functions, but since their C source code is available, we could analyze it.

Remark 9.1 **`cffi` works on shared libraries**

`cffi` can also work with compiled shared libraries (such as `libsyracuse.so`). Since functions are declared using `cffi.cdef` (Listing 9.7, line 10), it is still able to generate wrapper code in this case.

9.2.3 Swig

`swig` [9] is able to generate interfaces between C code and multiple languages, such as Python, JavaScript or OCaml. Swig requires the definition of an interface file (Listing 9.9), defining the module's name, which headers files should be included in the module's generated code, and which C functions should be made available.

Listing 9.9: Contents of file `syracuse.i`

```
1 %module syracuse
2
3 %{
4 #define SWIG_FILE_WITH_INIT
5 #include "syracuse.h"
6 %}
7
8 int syrac_nth(int, int);
```

Then, running `swig -python syracuse.i` creates two files: `syracuse.py` and `syracuse_wrap.c`. We can define a `setup.py` using `setuptools`, defining how to compile `syracuse_wrap.c` into a Python module, similarly to what we have shown in Section 9.1.8. It is then possible to call `syracuse.syrac_nth` once the module `syracuse` has been imported. Similarly to the `cffi`, `swig` checks that objects passed have correct types and can be represented in C. We show an excerpt of the wrapper function in Listing 9.10. Similarly to `cffi`, we believe that analyzing multilanguage code using `swig` should be easy using the analysis of Chapter 11.

Listing 9.10: Generated `swig` wrapper

```
1  SWIGINTERN PyObject *_wrap_syrac_nth(PyObject *SWIGUNUSEDPARM(self), PyObject *args) {
2    PyObject *resultobj = 0;
3    int arg1 ;
4    int arg2 ;
5    int val1 ;
6    int ecode1 = 0 ;
7    int val2 ;
8    int ecode2 = 0 ;
9    PyObject *swig_obj[2] ;
10   int result;
11
12   if (!SWIG_Python_UnpackTuple(args, "syrac_nth", 2, 2, swig_obj)) SWIG_fail;
13   ecode1 = SWIG_AsVal_int(swig_obj[0], &val1);
14   if (!SWIG_IsOK(ecode1)) {
15     SWIG_exception_fail(SWIG_ArgError(ecode1), "in method '" "syrac_nth" "', argument "
       ↪  "1"" of type '" "int""'");
16   }
17   arg1 = (int)(val1);
18   ecode2 = SWIG_AsVal_int(swig_obj[1], &val2);
19   if (!SWIG_IsOK(ecode2)) {
20     SWIG_exception_fail(SWIG_ArgError(ecode2), "in method '" "syrac_nth" "', argument "
       ↪  "2"" of type '" "int""'");
21   }
22   arg2 = (int)(val2);
23   result = (int)syrac_nth(arg1,arg2);
24   resultobj = SWIG_From_int((int)(result));
25   return resultobj;
26 fail:
27   return NULL;
28 }
```

9.2.4 Cython

`cython` [50] has a different goal: it aims at generating high-performance Python code by letting developers annotate their Python programs with C datatypes. `cython` then compiles these annotated Python programs to C programs. These C programs may use the Python/C API (to handle the Python objects provided to or returned by the functions, as well as calls back to Python when the simplification to C datatypes was not possible). Our running example is shown in its Cython version in Listing 9.11. The first two lines are the only ones different from generic Python code: both the arguments and variables `u` and `i` are declared as C integers, with the `int` and `cdef int` keywords respectively.

The code generated by `cython` is shown in Listing 9.12. `__PyInt_From_int` (at line 29) eventually calls a Python/C API function, such as `PyInt_FromLong`. `cython` also performs type and value checks to ensure conversions to C datatypes will be correct. Since `cython` generates code using the Python/C API, our analysis of Chapter 11 should be easily adapted.

Listing 9.11: Cython code for $3n + 1$ sequence computation

```
def syrac_nth(int u0, int n):
    cdef int u, i

    u = u0
    i = 0
    while(i < n):
        if(u % 2 == 0): u = u / 2
        else: u = 3 * u + 1
        i = i + 1
    return u
```

Listing 9.12: Abbreviated code generated by `cython` for Listing 9.11

```
static PyObject *__pyx_pf_8syracuse_syrac_nth(CYTHON_UNUSED PyObject *__pyx_self, int
↪  __pyx_v_u0, int __pyx_v_n) {
  int __pyx_v_u;
  int __pyx_v_i;
  PyObject *__pyx_r = NULL;
  int __pyx_t_1;
  PyObject *__pyx_t_2 = NULL;

  __pyx_v_u = __pyx_v_u0;

  __pyx_v_i = 0;

  while (1) {
    __pyx_t_1 = ((__pyx_v_i < __pyx_v_n) != 0);
    if (!__pyx_t_1) break;

    if (__pyx_t_1) {
      __pyx_v_u = __Pyx_div_long(__pyx_v_u, 2);
      goto __pyx_L5;
    }

    /*else*/ {
      __pyx_v_u = ((3 * __pyx_v_u) + 1);
    }

    __pyx_L5:;
    __pyx_v_i = (__pyx_v_i + 1);
  }

  __pyx_t_2 = __Pyx_PyInt_From_int(__pyx_v_u); if (unlikely(!__pyx_t_2)) __PYX_ERR(0, 10,
  ↪  __pyx_L1_error)
  __pyx_r = __pyx_t_2;
  __pyx_t_2 = 0;

  return __pyx_r;
}
```

9.3 Conclusion

This chapter defined an example of native module defining a simple `Counter` class. We have seen that discrepancies between the C and Python values created behaviors (integer wrap-arounds) or conversion errors that a Python developer may not be used to. This example will be reused in the next chapters.

We surveyed other interoperability mechanisms available between Python and C. All rely on the Python/C API in which our running example is written.

The next two chapters respectively define a multilanguage semantics for programs using the Python/C API, and an analysis based on this semantics. Since the code generated by the `cffi`, `swig` and `cython` rely on the Python/C API, we believe these interoperability mechanisms would not require much effort to be supported by our analyzer.

10

Concrete Multilanguage Semantics

This chapter defines the semantics of the interface between Python and C, corresponding to the API presented in the previous chapter. We start by explaining informally the choices made to model the semantics.

A low-level, pure-C semantics. A simple approach to define the multilanguage semantics is to notice that the reference Python interpreter, CPython, is written in C. Thus, we could define the semantics of a multilanguage program as the C semantics of CPython's source code, itself called with the program as input. This approach relies on the fact that the Python interpreter is implemented in C, so we can confuse the interoperability mechanism with CPython's execution. This view would however be too low-level to analyze programs, as the analysis would have to analyze CPython's implementation details in addition to the input program. In addition, we have already developed Python analyses based on a higher-level Python semantics, that we would like to reuse.

Building upon the Python and C semantics. We opt instead for a semantics that builds upon the semantics of each language. These concrete semantics will be used in a black-box fashion. Our goal is to delegate most of the work to the semantics of each language (in particular, the semantics of pure-Python or pure-C parts will be delegated to these semantics). This high-level, delegation-based approach will also simplify our analysis (in Chapter 11) as we will reuse the analyses of Python and C in a similar black-box fashion.

We first need to figure out how to make the semantics of each language interoperate. We can notice that only dynamically allocated Python objects can be passed from one language to another. In our running example, a tuple containing the integer object `2**p-1` is passed (Listing 9.1, line 6) from Python to the `CounterIncr` method (Listing 9.2, lines 17-25). In the other way, the counter member specified lines 39-42 creates a Python integer object during the attribute access at line 8. Thus, semantics can communicate through the addresses of objects passed from one language to the other.

Following our black-box approach, we keep the state of each language's semantics. This means in particular that the machine's heap will be represented using two *views*, since each language's state has its heap. We have previously stated that the addresses are used to communicate between the semantics. Thus, addresses will be shared between the heap's domains. In addition, objects allocated in C can be referenced in the Python heap and vice-versa. In most cases, an object is only described in one view of the heap: C or Python. For example, the `Counter` instance created at line 4 is allocated on the C side during the call to `PyType_GenericNew`. The `count` value is stored in the C heap. The address of this instance

is still referenced in Python's environment (since `c` references this instance). In some cases, though, a single object is present in both views of the heap. For example, the `Counter` class is a `PyTypeObject` in C, having all fields described in Listing 9.3. In Python, the `Counter` class has three methods `__new__`, `__init__`, `incr` and one attribute, `counter`. The detailed state is illustrated in Figure 10.5 and will be commented in the next section.

Of course, having two different representations of the heap creates two issues relating to them being coherent.

1. When one side dynamically allocates a new Python object, the heap description of the other side has to be updated. For example, when the `Counter` instance is allocated by `PyType_GenericNew` in C, the Python heap has to define the nominal and structural object's structure. This issue could be fixed by instrumenting the Python and C allocators in the semantics to update the state of the other language. However, this solution would break our black-box approach.
2. If one language can have side effects on the other's state, their states need to be synchronized afterward. For example, we could assume that a string defined in Python is passed to a C function. This C function could access the internal structure of the string structure and could alter its content. This modification would be done in pure C, without any call to the Python/C API. Without any synchronization into the Python state, the state in the Python semantics would not know that the string has changed.

We could remediate both issue by using a reduced product over the two semantics, and performing a reduction after any statement to keep the heap descriptions coherent. Applying a state reduction after each statement would however be costly. It turns out we can avoid reductions altogether, using a different, lighter way to handle each issue:

1. We do not automatically update the other heap at each dynamic allocation. Instead, we use *boundary functions*, ensuring that when an object switches from one language to the other, it is correctly represented in the incoming language's heap. These functions are purely here to ensure correct representations of each object in both heaps. They do not perform new allocations, nor conversions from a Python object to a C datatype. These conversions will be handled by API functions, which create new objects.
2. Through careful choices, we are able to make the heap representations complementary and disjoint, thus avoiding the second issue. Each language is responsible for disjoint parts of Python objects. In the case of the `Counter` instance, the C integer value is directly available through C only (detailed in Example 10.1). Our semantics assumes that builtin Python objects are opaque in C[1]. Thus, it is not possible to directly modify them using dereferences. They can only be accessed through the API in C. This assumption will be discussed in more details in Section 10.6. It is reasonable for our target use case, which is the analysis of code using the Python API. After all, the implementation details are not supposed to be exposed to third-party modules. In addition, our analysis checks that this assumption is satisfied (it is so far the case in the programs we analyzed). These API accesses to builtins will work by calling back the Python semantics. Although CPython ultimately implements these builtins in C, we abstract away from this behavior and rely on our Python semantics. We illustrate this approach on a tuple access in Example 10.2.

Example 10.1 **Accessing C data from Python**

In our running example (Listings 9.1 and 9.2), the `int count` field from a `Counter` instance is only exposed from the C. It is possible to read the counter's value from Python. This

[1]This approach is similar to the opaque representations of objects such as file descriptors in the work of Ouadjaout and Miné [121].

attribute access will call a C function performing the conversion of the C integer into a Python integer (the detailed execution will be shown in the abstract in Example 11.4). It allocates a new Python integer which is independent from the C integer. Hence, only C code directly dereferences the field value from the object's memory.

Example 10.2 **Accessing Python data from C**

Conversely, accessing a tuple at a given index is not directly possible in C. Of course, C variables may point to a Python object such as a tuple. However, tuple objects are opaque in C, as our semantics does not capture their internal representation. Accessing them through the C language using the API will ultimately be evaluated as a Python tuple access (described later in Section 10.5.2) in our semantics. This kind of access is used at line 21 of our running example, during the call to `PyArg_ParseTuple(args, "|i", &i)`. This call iterates through the elements of the tuple pointed by `args`. To perform the iteration, it uses `PyTuple_Size` to determine the length of the tuple, and `PyTuple_GetItem` to access elements. In our semantics, these builtins will in turn call the Python methods `tuple.__len__` and `tuple.__getitem__`.

Outline. We start by defining the multilanguage state in Section 10.1. The boundary functions are then defined in Section 10.2. We define how Python may call C functions (Section 10.3), and how C may perform callbacks to Python objects (Section 10.4). We also define conversions provided by the API in Section 10.5, such as conversions between Python integers and C `long`s, and tuples. API functions working on other builtin datatypes (such as floats, strings, lists, ...) exist and are supported by our analysis. They are similar to the cases presented here but not described.

10.1 Multilanguage state

We define the state on which each semantics operates. In the following, Python-related states and expressions will be written in green. C-related states and expressions will be in orange. We reuse the concrete state defined in Chapter 6 (Figure 6.2) for Python, and the one defined by Ouadjaout and Miné [121] for C. A set of heap addresses **Addr** (potentially infinite) is common to the states.

10.1.1 Python state

Figure 10.1 defines the concrete Python state used in this chapter. This definition was already provided during the definition of the concrete semantics, in Figure 6.2. Python objects are split into a nominal part (for the type and a potential builtin value) and a structural part (for the attributes). The nominal part **ObjN** can be some builtin, or an instance, defined by the address of the class from which it is instantiated. The structural part **ObjS** maps attribute names to their contents' addresses. The environment $\mathcal{E}_p$ maps variable identifiers Id_p to addresses (or `LocalUndef` for local variables with an undefined value – `LocalUndef` is not a Python object but an artefact of our semantics to handle that case). The heap $\mathcal{H}_p$ maps address to objects. The state is tagged by a flow token $f \in \mathcal{F}_p$ to handle non-local control-flow (created by exceptions for example). Given a state $\sigma_p \in \Sigma_p$, we write as $\sigma.\epsilon_p$ its environment and $\sigma.\eta_p$ its heap.

10.1.2 C state

The concrete C state is shown in Figure 10.2. The memory is decomposed into blocks **Base**, which are either variables Id_c or heap addresses **Addr**. Each block is decomposed into scalar

$$\mathsf{ObjN} \stackrel{\text{def}}{=} \mathtt{int}(i \in \mathbb{Z}) \cup \mathtt{bool}(b \in \{\mathtt{True}, \mathtt{False}\}) \cup \mathtt{float}(f \in \mathbb{F}_{64})$$
$$\cup\, \mathtt{str}(s \in \mathit{string}) \cup \mathsf{None} \cup \mathtt{NotImpl} \cup \mathsf{List}(ls \in \mathsf{Addr}^*) \cup \mathsf{Tuple}(t \in \mathsf{Addr}^*)$$
$$\cup\, \mathsf{Method}(a \in \mathsf{Addr}, f) \cup \mathsf{Fun}(f) \cup \mathsf{Class}(c) \cup \mathsf{Inst}(a \in \mathsf{Addr})$$
$$\mathsf{ObjS} \stackrel{\text{def}}{=} \blacksquare \cup (\mathit{string} \rightharpoonup \mathsf{Addr})$$
$$\mathsf{Value}_p \stackrel{\text{def}}{=} \mathsf{Addr}$$
$$\mathcal{F}_p \stackrel{\text{def}}{=} \{\, \mathit{cur}, \mathit{ret}, \mathit{brk}, \mathit{cont}, \mathit{exn}\ a \in \mathsf{Addr} \,\}$$
$$\mathcal{E}_p \stackrel{\text{def}}{=} \mathsf{Id}_p \rightharpoonup \mathsf{Addr} \cup \mathtt{LocalUndef}$$
$$\mathcal{H}_p \stackrel{\text{def}}{=} \mathsf{Addr} \rightharpoonup \mathsf{ObjN} \times \mathsf{ObjS}$$
$$\Sigma_p \stackrel{\text{def}}{=} \mathcal{F}_p \times \mathcal{E}_p \times \mathcal{H}_p$$
$$\mathbb{E}_p[\![\, \mathit{expr} \,]\!] : \mathcal{P}(\Sigma_p) \to \mathcal{P}(\mathsf{Value}_p{}^{\perp} \times \Sigma_p)$$
$$\mathbb{S}_p[\![\, \mathit{stmt} \,]\!] : \mathcal{P}(\Sigma_p) \to \mathcal{P}(\Sigma_p)$$

Figure 10.1: Concrete Python State

elements (machine integers, floats, pointers). $\wr b, o, \tau \wr$ denotes the memory region in block b, starting at offset o and having type τ, abstracted as cells [107] in our next chapter. C values Value_c are either machine numbers *MNum* (including integer and float datatypes, see Example 10.3 for example), or pointers Ptr defined by their block and offset. Additionally, pointers can be `NULL` or `invalid`. The state Σ_c consists of an environment and a heap. The environment $\mathcal{E}_c$ maps scalar elements to values. The heap $\mathcal{H}_c$ maps address to the type of allocated resource and their size. The type of allocated resources is `Malloc` when the standard C library `malloc` is used[2]. The Python allocator (called by `PyType_GenericNew`) will create resources of type `PyAlloc`, ensuring that:

(i) Python objects are well constructed by the correct allocator
(ii) the C code cannot access directly, using pointer dereferences, these "opaque" objects and needs to use the API,
(iii) the C code cannot `free` a block allocated by Python.

Listing 10.1: Example C program to illustrate the concrete C state

```c
#include <stdlib.h>

typedef struct {int length; float *data;} ftab;

int main()
{
    ftab* f = malloc(sizeof(ftab));
    f->length = 2;
    f->data = malloc(f->length*sizeof(float));
    f->data[0] = 0;
    f->data[1] = 2;
}
```

Example 10.3 **Concrete C state of Listing 10.1**

We consider the program in Listing 10.1, defining a structure `ftab` containing an array of

[2]Other resources (such as file descriptors) can also be defined [121].

$$\mathcal{C}ells \stackrel{\text{def}}{=} \{\, \wr b, o, t \int \mid b \in \textsf{Base}, t\text{: scalartype}, 0 \leq o \leq \text{sizeof}(b) - \text{sizeof}(t) \,\}$$
$$Ptr \stackrel{\text{def}}{=} (\textsf{Base} \times \mathbb{Z}) \cup \{\, \texttt{NULL}, \texttt{invalid} \,\}$$
$$\textsf{Base} \stackrel{\text{def}}{=} \mathsf{Id}_c \cup \textsf{Addr}$$
$$\mathsf{Value}_c \stackrel{\text{def}}{=} \mathit{MNum} \cup Ptr$$
$$\mathcal{E}_c \stackrel{\text{def}}{=} \mathcal{C}ells \rightharpoonup \mathsf{Value}_c$$
$$\mathcal{H}_c \stackrel{\text{def}}{=} \textsf{Addr} \rightharpoonup \mathit{ident} \times \mathbb{N}$$
$$\Sigma_c \stackrel{\text{def}}{=} \mathcal{E}_c \times \mathcal{H}_c$$
$$\mathbb{E}_c[\![\, \mathit{expr} \,]\!] : \mathcal{P}(\Sigma_c) \to \mathcal{P}(\mathsf{Value}_c{}^{\perp} \times \Sigma_c)$$
$$\mathbb{S}_c[\![\, \mathit{stmt} \,]\!] : \mathcal{P}(\Sigma_c) \to \mathcal{P}(\Sigma_c)$$

Figure 10.2: Concrete C state

floating-point numbers alongside its length. We assume we have a 64-bits architecture. We allocate an array of size 2, initialize its first cell to 0 and the second to 2.

The concrete state is shown in Figure 10.3.

Environment	Heap
$\wr f, 0, \texttt{ptr} \int \mapsto (@_7, 0)$	
$\wr @_7, 0, \texttt{int} \int \mapsto 2$	$@_7 \mapsto \texttt{Malloc}, 16$
$\wr @_7, 8, \texttt{ptr} \int \mapsto 2$	
$\wr @_9, 0, \texttt{float} \int \mapsto 0$	$@_9 \mapsto \texttt{Malloc}, 8$
$\wr @_9, 4, \texttt{float} \int \mapsto 2$	

Figure 10.3: Concrete state obtained at the end of Listing 10.1

10.1.3 Combined state

Two new kinds of nominal objects are added to Python: CFun f for Python functions defined in C, CClass c for Python classes defined in C (where f and c denote the name of the underlying C function or class declaration). The combined state used for the multilanguage semantics is the product of the Python and C states, written Σ, and shown in Figure 10.4.

Remark 10.4 **Addresses are shared**

Each state may reference addresses originally allocated by the other language. In the running example (Listings 9.1 and 9.2), the Python variable `c` points to an instance of the `Counter` class. This instance has been allocated on the C side by `PyType_GenericNew` (cf. Section 9.1.5).

The multilanguage semantics $\mathbb{E}_{p \times c}[\![\cdot]\!]$ is defined over Python and C expressions. It operates over the whole state Σ and its return value matches the language of the input expression. We define the semantics of some builtins working at the boundary between Python and C in Sections 10.3 to 10.5. These builtins operate on the whole state. For expressions not working

$$\Sigma = \Sigma_p \times \Sigma_c$$
$$\mathbb{E}_{p\times c}[\![\, expr_p \,]\!] : \mathcal{P}(\Sigma) \to \mathcal{P}(\mathsf{Value}_p{}^{\perp} \times \Sigma)$$
$$\mathbb{E}_{p\times c}[\![\, expr_c \,]\!] : \mathcal{P}(\Sigma) \to \mathcal{P}(\mathsf{Value}_c{}^{\perp} \times \Sigma)$$

Figure 10.4: Combined State

at the boundary, the multilanguage semantics defaults to the usual Python or C semantics:

$$\mathbb{E}_{p\times c}[\![\, expr_p \,]\!]\Sigma = \cup_{\sigma_p,\sigma_c\in\Sigma}\{\, v_p, \sigma'_p, \sigma_c \mid (\sigma'_p, v_p) \in \mathbb{E}_p[\![\, expr_p \,]\!](\sigma_p) \,\}$$
$$\mathbb{E}_{p\times c}[\![\, expr_c \,]\!]\Sigma = \cup_{\sigma_p,\sigma_c\in\Sigma}\{\, v_c, \sigma_p, \sigma'_c \mid (\sigma'_c, v_c) \in \mathbb{E}_c[\![\, expr_c \,]\!](\sigma_c) \,\}$$

Example 10.5 Diagram of the multilanguage state on the running example

We show in Figure 10.5 a schematic representation of the program state after the execution of the first four lines of `count.py` (Listing 9.1). The six nodes with a header text starting in green are Python objects, and the nominal object is described in the header. The three remaining nodes with a header text in orange are resources allocated by malloc, and we give their type.

The instance node in the top right corner represents the `Counter` instance created at line 4 of `count.py`. The header in the node represents the nominal object, which is a Python instance here. This instance points to its parent class, CClass $(Counter)$. Two fields are defined in the corresponding C structure `CounterO`. The mandatory Python object header `ob_base` has a field `ob_type` pointing to the class as well. The field `count` of the struct `CounterO` has been initialized to 0. The Python side of this object contains nothing and is thus not represented.

The class' node is the most interesting, as its state is shared among the two languages. The field `__new__` of the class points to a Python function which will call the builtin tp_new_wrapper function. This function will in turn call the C function stored in the `tp_new` field of the class (on the C side, the class is a `PyTypeObject`, cf. Listing 9.3). In our case, this is `PyType_GenericNew`. The other cases are similar. The Python fields are automatically synthesized from the `PyTypeObject` structure during the call to `PyType_Ready`. All descriptors are Python classes defined in C; hence they have a C state. Descriptors are used to wrap checks and adaptations around C functions. They will be described in more detail in Remarks 10.9 to 10.11. We have already mentioned this duality of field definitions in the previous chapter and provided a summary in Figure 9.1.

10.1.4 Handling Python exceptions in C

Python exceptions may be raised from the C code using the `PyErr_SetNone` builtin. In the Python interpreter, this sets a flag in a structure describing the interpreter's state. We model this by setting a global variable `exc` with the Python object passed as an argument. Additional functions such as `PyErr_Occurred` checking if an exception has been raised and `PyErr_Clear` removing raised exceptions are modeled by accessing and setting this same global variable.

10.2 Boundary functions

Boundary functions ensure that Python objects are correctly represented in the heap of each language. Given a Python object allocated at an address, these boundaries only update the

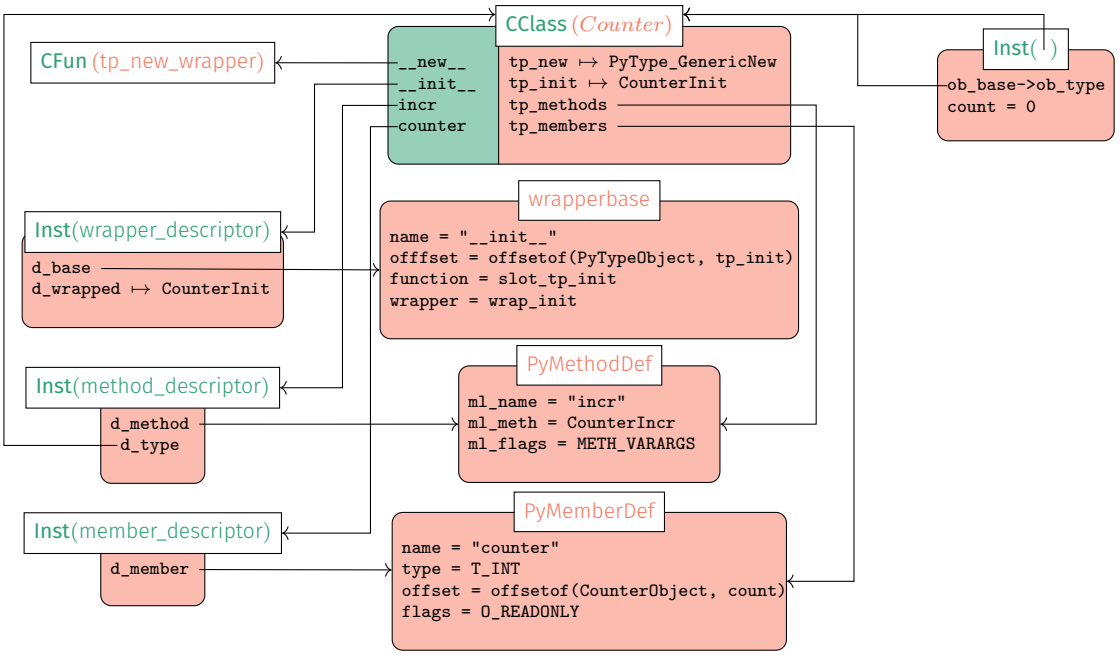

Figure 10.5: Schematic representation of the concrete state reached at line 5 in Listing 9.1

heap of the target language at this address. They do not perform new dynamic allocations. The C to Python boundary also ensures that only Python objects are passed back to Python. Contrary to the other boundary, it may fail (if something that is not a Python object is passed and return $\bot$). We define two boundary functions converting a Python value (in the semantical term, Value_p of Figure 10.1) into a C value (Value_c, Figure 10.2) and conversely (written ${}^{p}\!\hookrightarrow_{c}$ and ${}^{c}\!\hookrightarrow_{p}$). We define the boundary functions as operating on a single value and state at a time, but they are trivially lifted to the powerset.

$$
\begin{aligned}
{}^{p}\!\hookrightarrow_{c} &: \text{Value}_p \times \Sigma \to \text{Value}_c \times \Sigma \\
{}^{c}\!\hookrightarrow_{p} &: \text{Value}_c \times \Sigma \to \text{Value}_p{}^{\bot} \times \Sigma
\end{aligned}
$$

Example 10.6 **Boundary effect**

Going back to the example mentioned in Remark 10.4, the variable `c` of our running example points to an address @ of a `Counter` instance which has been allocated on the C side. When this variable becomes reachable in the Python state, the boundary ensures that Python's heap knows that @ points to a `Counter` instance and has no fields currently defined.

Remark 10.7 **Value conversion**

These functions do not convert values from one language to another. This kind of conversion is handled by builtin conversion functions such as `PyLong_AsLong`, `PyLong_FromLong` for Python integer to C long conversion. These conversions are formalized in Section 10.5.1.

These boundary functions are shallow and lazy:

(i) only objects switching languages are passed through those boundaries,
(ii) an object that has already been converted does not need to be converted again (i.e., when its address is already in the other language's heap). This is because the conversion only propagates immutable data, such as the type of an object.

10.2.1 Python to C boundary

The boundary from Python to C is described in Figure 10.6. As we have mentioned, the boundary is lazy: if the object is already defined in the C heap, there is nothing to do. If the object is not represented in the C heap, the boundary is first applied recursively to the class of the object (using the `type` operator of Python). Then, we update the heap: the object has been allocated by Python, and has the size of `PyObject`. If the object is a class, it has the size of `PyTypeObject`, and we call the class initializer `PyType_Ready` afterward. The last step performed is to initialize the `ob_type` field of the object to point to its class.

$$
\begin{array}{l}
{}^{p}\hookrightarrow_{c}(v_p, \sigma_p, \sigma_c) = \\
\quad \text{if } v_p \in \sigma.\eta_c \text{ then } (v_p, 0), \sigma_p, \sigma_c \text{ else} \\
\quad \text{letb } ty_p, \sigma_p = \mathbb{E}_p[\![\, type(v_p) \,]\!]\sigma_p \text{ in} \\
\quad \text{letb } (ty_c, 0), \sigma_p, \sigma_c = {}^{p}\hookrightarrow_{c}(ty_p, \sigma_p, \sigma_c) \text{ in} \\
\quad \text{let } \sigma_p, \sigma_c = \\
\qquad \text{if } \sigma_p(v_p) = \mathsf{Class}(c) \text{ then} \\
\qquad\quad \text{let } \sigma_c = \sigma.\epsilon_c, \sigma.\eta_c[v_p \mapsto \texttt{PyAlloc}, \text{sizeof}(\texttt{PyTypeObject})] \text{ in} \\
\qquad\quad \mathbb{E}_{p\times c}[\![\, \text{PyType\_Ready}(v_p) \,]\!](\sigma_p, \sigma_c) \\
\qquad \text{else } \sigma_p, (\sigma.\epsilon_c, \sigma.\eta_c[v_p \mapsto \texttt{PyAlloc}, \text{sizeof}(\texttt{PyObject})]) \\
\quad \text{in} \\
\quad \text{let } \sigma_c = \mathbb{S}_c[\![\, v_p\texttt{->ob\_type} = ty_c \,]\!]\sigma_c \text{ in} \\
\quad (v_p, 0), \sigma_p, \sigma_c
\end{array}
$$

Figure 10.6: Python to C value boundary

10.2.2 C to Python boundary

The converse boundary (Figure 10.7) starts by checking that the value is a heap allocated Python object, allocated with resource type `PyAlloc`. It calls itself recursively on the class of the object (using the `ob_type` field in C). The Python heap is updated with the converted object.

$$
\begin{array}{l}
{}^{c}\hookrightarrow_{p}(v_c, \sigma_p, \sigma_c) = \\
\quad \text{if } v_c \notin \mathsf{Addr} \times \{\, 0 \,\} \;||\; \sigma.\eta_c(\text{fst } v_c) \neq (\texttt{PyAlloc}, \_) \text{ then } \bot, \sigma_p, \sigma_c \text{ else} \\
\quad \text{let } v = \text{fst } v_c \text{ in} \\
\quad \text{if } v \in \sigma.\eta_p \text{ then } v, \sigma_p, \sigma_c \text{ else} \\
\quad \text{letb } ty_c, \sigma_c = \mathbb{E}_c[\![\, (\texttt{(PyObject*)}v)\texttt{->ob\_type} \,]\!]\sigma_c \text{ in} \\
\quad \text{letb } ty_p, \sigma_p, \sigma_c = {}^{c}\hookrightarrow_{p}(ty_c, \sigma_p, \sigma_c) \text{ in} \\
\quad \text{let } \sigma_p = \sigma.\epsilon_p, \sigma.\eta_p[v \mapsto \mathsf{Inst}(ty_p), \emptyset] \text{ in } v, \sigma_p, \sigma_c
\end{array}
$$

Figure 10.7: C to Python value boundary

10.3 C call from Python

The semantics of C function calls from Python is shown in Figure 10.8. It corresponds to the function `PyCFunction_Call` in the interpreter's implementation.[3] C functions callable from Python can only have two arguments (cf. the type of **`PyCFunction`**, Listing 9.3, line 6). Thus, the Python arguments are split into the first one e_1 and the other ones, bundled in a tuple. The boundary from Python to C is applied to e_1, and to the tuple containing the other arguments. Then, the C function is evaluated using the standard C semantics. Afterward, **`out_check`** ensures that the function returned **`NULL`** if and only if an exception has been set in the interpreter state. Otherwise, a **`SystemError`** exception is raised. Finally, the C value is passed through the boundary function.

$$
\begin{array}{l}
\mathbb{E}_{p\times c}[\![\,(\mathsf{CFun}\ f)(e_1,e_2,\ldots,e_n)\,]\!](\sigma_p,\sigma_c) = \qquad \texttt{tp\_field PyCFunction\_Call}\\
\quad \mathsf{letb}\ c_1,\sigma_p,\sigma_c = {}^{p}\!\hookrightarrow_{c}(e_1,\sigma_p,\sigma_c)\ \mathsf{in}\\
\quad \mathsf{letb}\ p_2,\sigma_p = \mathbb{E}_{p}[\![\,tuple(e_2,\ldots,e_n)\,]\!]\sigma_p\ \mathsf{in}\\
\quad \mathsf{letb}\ c_2,\sigma_p,\sigma_c = {}^{p}\!\hookrightarrow_{c}(p_2,\sigma_p,\sigma_c)\ \mathsf{in}\\
\quad \mathsf{letb}\ c_f,\sigma_c = \mathbb{E}_{c}[\![\,f(c_1,c_2)\,]\!]\sigma_c\ \mathsf{in}\\
\quad \mathsf{letb}\ c_f,\sigma_c = \texttt{out\_check}(c_f,\sigma_c)\ \mathsf{in}\\
\quad {}^{c}\!\hookrightarrow_{p}(c_f,\sigma_p,\sigma_c)
\end{array}
$$

Figure 10.8: C call from Python

Remark 10.8 **Output check in CPython**

The function **`out_check`** models the effect of function `_Py_CheckFunctionResult`, defined by the Python interpreter. This function is called at the end of the execution of **`PyCFunction_Call`**.

We have shown the simple case of C calls from Python in Figure 10.8. There are three variations, acting as descriptor objects, shown in the following remarks.

Remark 10.9 **Method descriptors**

Method descriptors act as wrappers around C functions, checking that the first argument of the call is an instance of the targeted class (otherwise, a **`TypeError`** exception is raised). The function to call is stored in the **`d_method`** field of the descriptor, and the type in the **`d_type`** field (example in Figure 10.5). In the semantics, this adds an input check. It corresponds to the function `method_check_args` in Python's source code.

Remark 10.10 **Wrapper descriptors**

Wrapper descriptors are used to call C functions that do not respect the signature of **`PyCFunction`** (Listing 9.3, lines 6-7). For example, initialization functions such as **`CounterInit`** (having type **`initproc`**, cf. Listing 9.3) also take a dictionary of arguments as third argument and return an integer as a result. Upon a call to the wrapper descriptor instance, `wrapperdescr_raw_call` will call the wrapper at **`d_base->wrapper`**. In our case, this field (Figure 10.5) will call **`wrap_init`**. In the semantics and our analysis, we can directly use the source code of **`wrap_init`** (shown in Listing 10.2). This function signals an error by return-

[3] We reuse the convention of Chapter 6: Python functions written in `gray and typewriter font` are clickable links to their source code on Github.

ing `NULL` if the initialization function returns a negative integer, and returns the `None` value otherwise.

Listing 10.2: Code of `wrap_init`

```
static PyObject *
wrap_init(PyObject *self, PyObject *args, void *wrapped, PyObject *kwds)
{
    initproc func = (initproc)wrapped;

    if (func(self, args, kwds) < 0)
        return NULL;
    Py_RETURN_NONE;
}
```

Remark 10.11 **Member descriptors**

An example of member descriptor has been already presented in Section 9.1.7. The `member_descriptor` class defines `__get__` and `__set__` methods that are automatically called should an attribute be accessed or set (semantics shown in Figures 6.31 and 6.32). These methods respectively call `member_get` and `member_set`. These functions in turn rely on the specification provided in a `d_member` field (Figure 10.5) to perform the corresponding getter or setter action (with conversions from Python to C datatypes or conversely, and checks that the member is not read-only).

10.4 Python call from C

$$
\begin{array}{l}
\mathbb{E}_{p\times c}[\![\,\mathsf{PyObject\_CallObject}(f,a)\,]\!](\sigma_p,\sigma_c) = \\
\quad \text{letb } f_p,\sigma_p,\sigma_c = {}^{c}\!\hookrightarrow_{p}(f,\sigma_p,\sigma_c) \text{ in} \\
\quad \text{letb } a_p,\sigma_p,\sigma_c = {}^{c}\!\hookrightarrow_{p}(a,\sigma_p,\sigma_c) \text{ in} \\
\quad \text{let } r_p,\sigma_p = \mathbb{E}_p[\![\, f_p(*a_p)\,]\!]\sigma_p \text{ in} \\
\quad \text{if } \sigma_p = (\mathit{cur},\_,\_) \text{ then } {}^{p}\!\hookrightarrow_{c}(r_p,\sigma_p,\sigma_c) \\
\quad \text{else } \mathsf{convert\_err}(\sigma_p,\sigma_c) \\
\\
\mathsf{convert\_err}(\sigma_p,\sigma_c) = \\
\quad \text{let } \mathit{exn}\ e,\epsilon_p,\eta_p = \sigma_p \text{ in} \\
\quad \text{letb } e_c,\sigma_p,\sigma_c = {}^{p}\!\hookrightarrow_{c}(e,(\mathit{cur},\epsilon_p,\eta_p),\sigma_c) \text{ in} \\
\quad \mathtt{NULL},\sigma_p,\mathbb{S}_c[\![\,\mathsf{PyErr\_SetNone}(e_c)\,]\!]\sigma_c
\end{array}
$$

Figure 10.9: Python call from C

Calls back to Python from the C code are possible using the `PyObject_CallObject` function, formalized in Figure 10.9. The first argument is the object being called. The second argument is a tuple containing all the parameters. These two arguments are first passed through the C to Python boundary. Then, we use the Python semantics to evaluate the call (the $*$ operator in Python unpacks the tuple into the arguments of the variadic function). If the call is successful (i.e., the execution is normal, shown by flow token *cur*), the converse boundary function is applied. If an exception has been raised during the evaluation of the Python call, we revert to

the *cur* flow token and pass the exception object e through the boundary. The result of the call will be `NULL`, and the exception will be set on the C side by calling `PyErr_SetNone`.

Remark 10.12 **No references to the source code**

We do not provide references to the source code of CPython for the Python call from C, as well as the integer and tuple conversions of the next section. Indeed, the corresponding CPython code accesses the internal structures of these builtins. We model these accesses instead with the Python semantics.

10.5 Builtins of the API

We show how integers are converted in Section 10.5.1. The case of Python container accesses from the C side is shown in Section 10.5.2. We explain how generic converters using format strings work in Section 10.5.3, and how the fields of a C class are lifted from C to Python in Section 10.5.4.

10.5.1 Integer conversions

We show conversion functions from C `long` to Python integers and back in Figure 10.10. Converting a C `long` of value v_c to a Python integer is done by calling the integer constructor in the Python semantics, where the C value is cast as a literal, mathematical integer (written in blue), and applying the boundary afterward. To perform the conversion back, we apply the boundary function to the C value. Then, we check if the corresponding Python value v_p is an integer by looking into the Python heap. If that is the case, we check that this integer fits in a C `long` (Python integers are unbounded). Otherwise we raise an `OverflowError` and the function returns -1. A `TypeError` exception is raised and the function returns -1 if the object is not an integer.

$$
\begin{array}{l}
\mathbb{E}_{p\times c}[\![\,\mathrm{PyLong\_FromLong}(v_c)\,]\!](\sigma_p,\sigma_c) = {}^{p}\!\hookrightarrow_{c}(\mathbb{E}_p[\![\,int(v_c)\,]\!](\sigma_p),\sigma_c) \\
\mathbb{E}_{p\times c}[\![\,\mathrm{PyLong\_AsLong}(v_c)\,]\!](\sigma_p,\sigma_c) = \\
\quad \text{let } v_p,\sigma_p,\sigma_c = {}^{c}\!\hookrightarrow_{p}^{\sharp}(v_c,\sigma_p,\sigma_c) \text{ in} \\
\quad \text{if } \sigma.\eta_p = int(i) \text{ then} \\
\qquad \text{if } i \in [-2^{63}, 2^{63}-1] \text{ then } i,\sigma_p,\sigma_c \\
\qquad \text{else } -1,\sigma_p,\mathbb{S}_c[\![\,\mathrm{PyErr\_SetNone(PyExc\_OverflowError)}\,]\!]\sigma_c \\
\quad \text{else } -1,\sigma_p,\mathbb{S}_c[\![\,\mathrm{PyErr\_SetNone(PyExc\_TypeError)}\,]\!]\sigma_c
\end{array}
$$

Figure 10.10: Conversion from Python builtin integers to C long

10.5.2 Operations on containers

We show how container operations are handled, using the case of tuples as an example in Figure 10.11. The same approach is used for other operations and other containers. In each case, the goal is to rewrite it as a Python evaluation, and handle conversions and errors before and after.

$\mathrm{PyTuple\_GetItem}(o,i)$ returns the object located at index i of the tuple o. The index is provided as a C integer of type `Py_ssize_t`, which corresponds to the C type `ssize_t`. o is passed as a pointer to a `PyObject`, and the same applies for the result. The function starts by passing o through the boundary. i is converted into a Python integer, and passed through the

boundary too. Then, the Python versions of these objects are passed to the `tuple.__getitem__` function. If the call is successful, the result is passed through the converse boundary and returned. Otherwise, we use the error conversion function defined in Figure 10.9 to translate the exceptional Python flow into an exception in the C state (cf. Section 10.1.4).

PyTuple_Size(o) returns the size of the tuple o as a C integer of type `Py_ssize_t`. It passes o through the boundary, so it can then be used in the Python call to `tuple.__len__`. If the call is successful, the result is passed through a boundary, and converted into a C integer. Otherwise, we convert the exceptional Python flow into an exception in the C state.

$$
\begin{array}{l}
\mathbb{E}_{p\times c}[\![\,\mathsf{PyTuple\_GetItem}(o,i)\,]\!](\sigma_p,\sigma_c) = \\
\quad \mathsf{letb}\ o_p,\sigma_p,\sigma_c = {}^{c}\!\hookrightarrow_p(o,\sigma_p,\sigma_c)\ \mathsf{in} \\
\quad \mathsf{letb}\ i_p,\sigma_p,\sigma_c = {}^{c}\!\hookrightarrow_p \circ\, \mathbb{E}_{p\times c}[\![\,\mathsf{PyLong\_FromSsize\_t}(i)\,]\!](\sigma_p,\sigma_c)\ \mathsf{in} \\
\quad \mathsf{letb}\ r_p,\sigma_p,\sigma_c = \mathbb{E}_p[\![\,\texttt{tuple.\_\_getitem\_\_}(o_p,i_p)\,]\!](\sigma_p,\sigma_c)\ \mathsf{in} \\
\quad \mathsf{if}\ \sigma_p = (\mathit{cur},\_,\_)\ \mathsf{then}\ {}^{p}\!\hookrightarrow_c(r_p,\sigma_p,\sigma_c) \\
\quad \mathsf{else}\ \mathsf{convert\_err}(\sigma_p,\sigma_c)
\end{array}
$$

$$
\begin{array}{l}
\mathbb{E}_{p\times c}[\![\,\mathsf{PyTuple\_Size}(o)\,]\!](\sigma_p,\sigma_c) = \\
\quad \mathsf{letb}\ o_p,\sigma_p,\sigma_c = {}^{c}\!\hookrightarrow_p(o,\sigma_p,\sigma_c)\ \mathsf{in} \\
\quad \mathsf{letb}\ l_p,\sigma_p,\sigma_c = \mathbb{E}_p[\![\,\texttt{tuple.\_\_len\_\_}(o_p)\,]\!](\sigma_p,\sigma_c)\ \mathsf{in} \\
\quad \mathsf{if}\ \sigma_p = (\mathit{cur},\_,\_)\ \mathsf{then} \\
\qquad \mathsf{letb}\ l_c,\sigma_p,\sigma_c = {}^{p}\!\hookrightarrow_c(l_p,\sigma_p,\sigma_c)\ \mathsf{in}\ \mathbb{E}_{p\times c}[\![\,\mathsf{PyLong\_AsSsize\_t}(l_c)\,]\!](\sigma_p,\sigma_c) \\
\quad \mathsf{else}\ \mathsf{convert\_err}(\sigma_p,\sigma_c)
\end{array}
$$

Figure 10.11: Tuple access and length semantics, from C

10.5.3 Generic converters: `PyArg_ParseTuple`, `Py_BuildValue`

`PyArg_ParseTuple` matches a tuple into different C values using a format string. These format strings are similar to the ones used by C's `printf`. They are described in CPython's documentation [148].! The concrete semantics of this function is its implementation in CPython. It iterates on the format string and the tuple elements to perform the conversions, which end up calling builtin conversion functions such as the ones shown in Section 10.5.1. For example, when `PyArg_ParseTuple` encounters an `'i'` `char` in its conversion string, it executes the code shown in Listing 10.3.[4] As explained in our example from Section 9.1.6, it first calls `PyLong_AsLong` and converts the `long` to `int` checking for additional overflows. Our analyzer is able to analyze the code of Listing 10.3 directly, without having to rely on a builtin stub.

`Py_BuildValue` performs the inverse operation: it creates Python objects from C values and a format string. The same approach is used in this case.

Listing 10.3: Python to C int conversion done by `convertsimple`, called by `PyArg_ParseTuple`

```
long ival = PyLong_AsLong(obj);
if(ival == -1 && PyErr_Occurred()) {
  return 0;
```

[4] In the abstract, we use a stub, since the CPython implementation is a bit too complex to be precisely analyzed: it uses loops, and recursive functions manipulating variable arguments. `PyArg_ParseTuple` is defined as a stub (just as `PyLong_AsLong` is), but the case of integers is delegated to the interpreter's implementation shown in Listing 10.3.

```
4 }
5 else if (ival > INT_MAX) {
6   PyErr_SetString(PyExc_OverflowError,
7                   "signed integer is greater than maximum");
8   return 0;
9 }
10 else if (ival < INT_MIN) {
11   PyErr_SetString(PyExc_OverflowError,
12                   "signed integer is less than minimum");
13   return 0;
14 } else {
15   *result = ival;
16   return 1;
17 }
```

10.5.4 Class initialization: PyType_Ready

We have shown in Listing 9.3 (at lines 22-34) a simplified definition of the **PyTypeObject** structure used to define Python classes in C. PyType_Ready takes this structure, having different fields (such as **tp_new**, **tp_init**, but also the methods of **tp_methods** and members of **tp_members**) and binds the corresponding attributes to objects in Python (i.e., it creates the left hand-side of the class in Figure 10.5). We do not show the formal semantics of this function. It just performs the lifting from C fields to Python ones we mentioned.

10.6 Threats to validity

This concrete semantics is already high-level. Our goal is to analyze Python programs with native C modules and detect all runtime errors that may happen. Assuming that those C modules use Python's API rather than directly modify the internal representation of builtins seems reasonable when analyzing third-party modules. This is the recommended approach for developers, as it eases maintenance of the codebase since API changes are not frequent and documented. On the contrary, internal representations may be changed in incompatible ways without notice. Our analysis, presented in the next chapter, is able to detect if a program does not use the Python API and tries to modify a builtin Python object directly. In addition, a lower-level semantics where implementation details of builtins are exposed would be much more complex and not benefit our analysis. Such a lower-level semantics may need to perform whole Python (resp. C) state conversion at the boundary when switching languages.

We have established this semantics by reading the code of the reference Python interpreter. Proving that our semantics is a sound abstraction of such lower-level semantics is left as future work.

The garbage collection based on reference counting is not supported by our semantics. Thus, we cannot detect deallocations that are performed too soon or that are not performed at all.

10.7 Related work

The seminal work of Matthews and Findler [102] defines the first semantics of multilanguage systems, using the notion of boundaries to model conversion between languages. They start from a statically typed call-by-value lambda-calculus (with natural numbers) and an untyped version of the lambda-calculus, with natural numbers and a notion of exceptions. They introduce the notion of boundary to convert the value of one language into the other. They study different embeddings: the lump embedding, where values from one language are opaque to the other, a natural embedding with type-directed conversion of values, values can be converted

using type information. They then prove different results on these combined languages, such as type-safety. Buro and Mastroeni [23] provide a general framework of language interoperability that is independent of the combined languages.

We study a real-world case where the interoperability mechanism was defined by the Python developers. Thus we do not search for the development of better interoperability mechanisms that could automatically perform conversions and prevent type errors by construction. Contrary to the cases studied in these works, the interoperability between Python and C does not mix languages or values at the syntactic level. Our notion of boundary only ensures the coherence of states. Conversions are performed by builtins from the API, with dynamic type checks.

10.8 Conclusion

This chapter defined a concrete semantics for multilanguage programs using the Python/C API as interoperability mechanism. It relied on the concrete semantics of the Python and C programming languages, to which the semantics of pure-Python and pure-C parts are delegated. We defined a careful separation of the Python and C views of the same heap, that does not require systematic reductions between the two states. We used instead lazy boundary functions to ensure objects are well-represented when they cross from one language to another. We assumed that the Python builtins were opaque structure to the C side. This ensures a disjoint separation between the views of the heap. API functions operating on those builtins in C are defined by calling back the Python semantics. The current limitation of this semantics is the absence of support of the garbage collector, which is based on reference counting.

11

Multilanguage Value Analysis

Although useful, multilanguage programs generate additional sources of bugs. Indeed, developers need to take into account different safety mechanisms and memory representations. Python is safe to the extent that runtime errors in pure Python programs are encapsulated into exceptions, which can be caught later on. This safety property breaks when C modules are used since a runtime error in C may irremediably terminate the program or create an inconsistent state. Python and C also have different representations. For example, Python integer objects use at least 24 bytes of memory and have unlimited precision, while C integers have fixed lengths (generally ranging from 8 to 64 bits) and can suffer from overflows.

Static analyzers tend to focus on analyzing one language at a time. They may use stubs to model the behavior of calls to other languages. These stubs may be time-consuming to implement if written by hand. They can undermine the soundness of the analyses since the actual code is not analyzed, and the stubs may be imprecise or wrong. For example, we have seen in Section 7.4 that our Python analyses can leverage official Python type annotations (defined by PEP 484 [154]) as stubs. While these analyses track uncaught Python exceptions, these type annotations do not declare which exceptions may be raised, thus adding an unchecked assumption to the soundness property (Remark 7.12).

We aim at analyzing both the native C code and the Python code (including callbacks to Python code from the native side) within the same analyzer. This analysis is based upon existing Python analysis (Chapter 8) and C analysis Ouadjaout and Miné [121], implemented in Mopsa. Our analysis detects runtime errors in the native C code (invalid pointer operations, invalid memory accesses, integer overflows), in the Python code (raised exceptions), and at the boundary between the languages. The underlying address allocation and numerical abstractions are shared.

We believe the approach described in this paper is general enough to be extended to other multilanguage settings, such as the analysis of Java and C through the JNI.

Outline. We define the abstract domain in Section 11.1, its transfer functions in Section 11.2. We show detailed examples of the analysis in Section 11.3. Section 11.4 defines the concretization of the domain and states the soundness theorem of the analysis. We discuss implementation details in Section 11.5 and evaluate our analysis in Section 11.6. This chapter finishes with a survey of related work (Section 11.7) and a conclusion (Section 11.8).

11.1 Abstract domain

We show a generic construction of the abstract multilanguage state. The multilanguage domain combining the underlying C and Python analyses is stateless. We describe in this section how the existing abstract state for C and Python analyses, developed independently, can be combined without further information to provide a multilanguage abstraction. We assume the abstract semantics of Python and C are provided through $\mathbb{E}^{\sharp}_{p}[\![\cdot]\!], \mathbb{E}^{\sharp}_{c}[\![\cdot]\!]$.

These are instantiated in practice using previous works: the value analysis of Python presented in Chapter 8, and the C analysis developed by Ouadjaout and Miné [121]. We assume that each language's abstract state relies on an address allocation abstraction (such as the callsite abstraction or the recency abstraction [4]) and a numerical abstraction (such as intervals, octagons, ...). We write $\Sigma^{\sharp}_{u}$ the cartesian product of these two abstractions. The abstract Python (resp. C) state can then be decomposed as a product $\Sigma^{\sharp}_{p} = \tilde{\Sigma}^{\sharp}_{p} \times \Sigma^{\sharp}_{u}$ (resp. $\Sigma^{\sharp}_{c} = \tilde{\Sigma}^{\sharp}_{c} \times \Sigma^{\sharp}_{u}$).[1] As we have mentioned before, the mutlilanguage domain is stateless. The state of the multilanguage analysis consists of the cartesian product of the Python and C abstract states, where the address allocation and numerical states are shared: $\Sigma^{\sharp}_{p\times c} = \tilde{\Sigma}^{\sharp}_{p} \times \tilde{\Sigma}^{\sharp}_{c} \times \Sigma^{\sharp}_{u}$.

Listing 11.1: Reminder from Listing 10.1 – example C program

```c
#include <stdlib.h>

typedef struct {int length; float *data;} ftab;

int main()
{
    ftab* f = malloc(sizeof(ftab));
    f->length = 2;
    f->data = malloc(f->length*sizeof(float));
    f->data[0] = 0;
    f->data[1] = 2;
}
```

Example 11.1 **Abstract C state of Listing 11.1**

We show how the C analysis abstracts a state. We reuse the example C program of the previous chapter, whose code is recalled in Listing 11.1. We assume the analysis targets a 64-bits architecture. It defines a structure `ftab` containing a dynamically allocated array of floating-point numbers alongside its length. We showed the concrete state in Example 10.3.

The recency abstraction $\sigma^{\sharp}_{\text{mem}}$ contains two addresses (both are recent), corresponding to the allocations performed at lines 7 and 9. Addresses are written in red to show that they have been allocated in C by `malloc`.

$$\sigma^{\sharp}_{\text{mem}} = \left\{ @^{\sharp}_{7}; @^{\sharp}_{9} \right\}$$

The cells domain [107] contains the set $\sigma^{\sharp}_{\text{cells}}$ of cells showed below. These cells are auxiliary variables, tracking the actual access patterns of variables and addresses. Each cell is defined by a base (a variable or an address), an offset, and the type accessed.

$$\sigma^{\sharp}_{\text{cells}} = \left\{ \wr f, 0, \texttt{ptr}\int; \wr @^{\sharp}_{7}, 0, \texttt{s32}\int; \wr @^{\sharp}_{7}, 8, \texttt{ptr}\int; \wr @^{\sharp}_{9}, 0, \texttt{float}\int; \wr @^{\sharp}_{9}, 4, \texttt{float}\int \right\}$$

[1] The construction could be adapted if the state is partitioned.

The pointer domain $\sigma^\sharp_{\text{ptr}}$ has two bindings:

$$\sigma^\sharp_{\text{ptr}} = \left(f \mapsto @^\sharp_7, \wr @^\sharp_7, 8, \texttt{ptr} \wr \mapsto @^\sharp_9\right)$$

The numerical domain (here, a constant or an interval domain is sufficient) has the state $\sigma^\sharp_{num}$. The first three lines are auxiliary variables representing the mathematical numbers corresponding to each cell with numerical type. The abstract state introduces auxiliary variables to track the size of dynamically-allocated memory blocks, and the pointer's offsets, shown in the last two lines.

$$\begin{aligned}\sigma^\sharp_{num} = \Big(&\underline{\text{int}}(\wr @^\sharp_7, 0, \texttt{s32} \wr) = 2,\\ &\underline{\text{float}}(\wr @^\sharp_9, 0, \texttt{float} \wr) = 0,\\ &\underline{\text{float}}(\wr @^\sharp_9, 4, \texttt{float} \wr) = 2,\\ \\ &\underline{\text{bytes}}(@^\sharp_7) = 8, \underline{\text{bytes}}(@^\sharp_7) = 16,\\ &\underline{\text{offset}}(\wr @^\sharp_7, 8, \texttt{ptr} \wr) = 0, \underline{\text{offset}}(\wr f, 0, \texttt{ptr} \wr) = 0\Big)\end{aligned}$$

Remark 11.2 **Unique numerical domain**

Concrete states use numerical values in different places (e.g., the C state has machine numbers, pointer offsets and memory blocks' sizes). All these values will be centralized in a common numerical domain in the abstract state. This centralization allows expressing relations between all those numerical variables, possibly improving the precision.

11.2 Transfer functions

Just as the concrete semantics builds upon the underlying C and Python semantics, so does our abstract semantics. We need to implement a few transfer functions, specific to the Python/C API. We define the abstract semantics of boundary operators. It is slightly different from the concrete, since the C heap relies on auxiliary variables in the abstract.

The other transfer functions are obtained by a straightforward replacement of concrete operators with abstract ones. Thus, we only show the abstract semantics of the cases used in the examples of the next section. This includes the abstract semantics of C calls from Python, the creation of a Python object from a C `long` and tuple conversions.

11.2.1 Boundaries

The abstract boundaries are defined in Figures 11.1 and 11.2. The only difference from the concrete definitions shown in Figures 10.6 and 10.7 lies in the update of the heap views.

In the case of the Python to C boundary, the concrete C heap $\sigma.\eta_c$ was updated updated to know that v_p is a resource of type `PyAlloc`, and size sizeof(`PyObject`). This varies slightly in the abstract. First, abstract addresses integrate a notion of type, so the `PyAlloc` resource type is not necessary. Second, the size of allocated memory is encoded in an auxiliary variable $\underline{\text{bytes}}(\cdot)$. This approach is used to handle variable allocation sizes (e.g., in the case of a dynamic array). Relying on the numerical domain to express constraints on the actual size may improve precision, since a relational domain could infer relations with program variables. In the end, the abstract C heap update is performed by executing the statement $\underline{\text{bytes}}(v_p) = \text{sizeof}(\texttt{PyObject})$ (or with sizeof(`PyTypeObject`) in the case of classes).

$$
\begin{array}{l}
{}^{p}\hookrightarrow_{c}^{\sharp}(v_p, \sigma_p, \sigma_c) = \\
\quad \text{if } v_p \in \sigma.\eta_c \text{ then } (v_p, 0), \sigma_p, \sigma_c \text{ else} \\
\quad \text{letb } ty_p, \sigma_p = \mathbb{E}_p^{\sharp}[\![\, type(v_p) \,]\!]\sigma_p \text{ in} \\
\quad \text{letb } (ty_c, 0), \sigma_p, \sigma_c = {}^{p}\hookrightarrow_{c}^{\sharp}(ty_p, \sigma_p, \sigma_c) \text{ in} \\
\quad \text{let } \sigma_p, \sigma_c = \\
\qquad \text{if } \sigma_p(v_p) = \mathsf{Class}(c) \text{ then} \\
\qquad\quad \text{let } \sigma_c = \mathbb{S}_c^{\sharp}[\![\, \underline{\text{bytes}}(v_p) = \text{sizeof}(\texttt{PyTypeObject}) \,]\!]\sigma_c \text{ in} \\
\qquad\quad \mathbb{E}_{p\times c}^{\sharp}[\![\, \text{PyType\_Ready}(v_p) \,]\!](\sigma_p, \sigma_c) \\
\qquad \text{else } \sigma_p, \mathbb{S}_c^{\sharp}[\![\, \underline{\text{bytes}}(v_p) = \text{sizeof}(\texttt{PyObject}) \,]\!]\sigma_c \\
\quad \text{in} \\
\quad \text{let } \sigma_c = \mathbb{S}_c^{\sharp}[\![\, v_p\texttt{->ob\_type} = ty_c \,]\!]\sigma_c \text{ in} \\
\quad (v_p, 0), \sigma_p, \sigma_c
\end{array}
$$

Figure 11.1: Python to C value boundary

$$
\begin{array}{l}
{}^{c}\hookrightarrow_{p}^{\sharp}(v_c, \sigma_p, \sigma_c) = \\
\quad \text{if } v_c \notin \mathsf{Addr} \times \{\, 0 \,\} \mid\mid kind(\text{fst } v_c) \neq \texttt{Py} \text{ then } \bot, \sigma_p, \sigma_c \text{ else} \\
\quad \text{let } v = \text{ fst } v_c \text{ in} \\
\quad \text{if } v \in \sigma.\eta_p \text{ then } v, \sigma_p, \sigma_c \text{ else} \\
\quad \text{letb } ty_c, \sigma_c = \mathbb{E}_c^{\sharp}[\![\, (\texttt{(PyObject*)}v)\texttt{->ob\_type} \,]\!]\sigma_c \text{ in} \\
\quad \text{letb } ty_p, \sigma_p, \sigma_c = {}^{c}\hookrightarrow_{p}^{\sharp}(ty_c, \sigma_p, \sigma_c) \text{ in} \\
\quad v, \mathbb{S}_{py.heap}^{\sharp}[\![\, add(v) \,]\!]\sigma_p, \sigma_c
\end{array}
$$

Figure 11.2: C to Python value boundary

We introduce a $kind$ function, which given an address returns `Py` if it is a Python object (corresponding to the `PyAlloc` resource type), and `C` if it has been allocated by `malloc` (corresponding to the `Malloc` resource type). This function is used in the case of the C to Python boundary to make sure that the address represents a Python object. The concrete heap update $\sigma.\eta_p[v \mapsto \mathsf{Inst}(ty_p), \emptyset]$ is replaced by the execution of the statement $add(v)$ in Python's heap abstract domain. This statement signals to the heap abstract domain of Python (defined in Section 7.2.3) that it needs to track the fields of a new address v. This domain then adds this address to its internal state, and assumes that no fields are currently defined.

11.2.2 C call from Python

C calls from Python are handled similarly in the concrete (Figure 10.8) and in the abstract (Figure 11.3). The first argument of the function is passed through the boundary, and the other arguments are first gathered into a tuple, which is then passed through the boundary. Then, the C function is called. The output checks verify that f returns `NULL` if and only if an exception has been set. The returned value of this function is passed through the C to Python boundary.

$$
\begin{aligned}
&\mathbb{E}^{\sharp}_{p\times c}[\![\,(\mathsf{CFun}\ f)(e_1, e_2, \ldots, e_n)\,]\!](\sigma_p, \sigma_c) = \\
&\quad \text{letb } c_1, \sigma_p, \sigma_c = {}^{p}\!\hookrightarrow_{c}^{\sharp}(e_1, \sigma_p, \sigma_c) \text{ in} \\
&\quad \text{letb } p_2, \sigma_p = \mathbb{E}^{\sharp}_{p}[\![\, tuple(e_2, \ldots, e_n)\,]\!]\sigma_p \text{ in} \\
&\quad \text{letb } c_2, \sigma_p, \sigma_c = {}^{p}\!\hookrightarrow_{c}^{\sharp}(p_2, \sigma_p, \sigma_c) \text{ in} \\
&\quad \text{letb } c_f, \sigma_c = \mathbb{E}^{\sharp}_{c}[\![\, f(c_1, c_2)\,]\!]\sigma_c \text{ in} \\
&\quad \text{letb } c_f, \sigma_c = \texttt{out\_check}^{\sharp}(c_f, \sigma_c) \text{ in} \\
&\quad {}^{c}\!\hookrightarrow_{p}^{\sharp}(c_f, \sigma_p, \sigma_c)
\end{aligned}
$$

Figure 11.3: C call from Python

$$
\begin{aligned}
&\mathbb{E}^{\sharp}_{p\times c}[\![\,\mathsf{PyLong\_FromLong}(v_c)\,]\!](\sigma_p, \sigma_c) = \\
&\quad \text{letb } v_u, \sigma_p, \sigma_c = \mathbb{E}^{\sharp}_{num}[\![\, v_c\,]\!](\sigma_p, \sigma_c) \text{ in} \\
&\quad {}^{p}\!\hookrightarrow_{c}^{\sharp}\left(\mathbb{E}^{\sharp}_{p}[\![\, int(v_c)\,]\!](\sigma_p), \sigma_c\right)
\end{aligned}
$$

Figure 11.4: Conversion from C long to Python integer

$$
\begin{aligned}
&\mathbb{E}^{\sharp}_{p\times c}[\![\,\mathsf{PyTuple\_GetItem}(o, i)\,]\!](\sigma_p, \sigma_c) = \\
&\quad \text{letb } o_p, \sigma_p, \sigma_c = {}^{c}\!\hookrightarrow_{p}^{\sharp}(o, \sigma_p, \sigma_c) \text{ in} \\
&\quad \text{letb } i_p, \sigma_p, \sigma_c = {}^{c}\!\hookrightarrow_{p}^{\sharp} \circ\, \mathbb{E}^{\sharp}_{p\times c}[\![\,\mathsf{PyLong\_FromSsize\_t}(i)\,]\!](\sigma_p, \sigma_c) \text{ in} \\
&\quad \text{letb } r_p, \sigma_p, \sigma_c = \mathbb{E}^{\sharp}_{p}[\![\,\texttt{tuple.\_\_getitem\_\_}(o_p, i_p)\,]\!](\sigma_p, \sigma_c) \text{ in} \\
&\quad \text{if } \sigma_p = (\mathit{cur}, \_, \_) \text{ then } {}^{p}\!\hookrightarrow_{c}^{\sharp}(r_p, \sigma_p, \sigma_c) \\
&\quad \text{else } \mathsf{convert\_err}^{\sharp}(\sigma_p, \sigma_c)
\end{aligned}
$$

$$
\begin{aligned}
&\mathbb{E}^{\sharp}_{p\times c}[\![\,\mathsf{PyTuple\_Size}(o)\,]\!](\sigma_p, \sigma_c) = \\
&\quad \text{letb } o_p, \sigma_p, \sigma_c = {}^{c}\!\hookrightarrow_{p}^{\sharp}(o, \sigma_p, \sigma_c) \text{ in} \\
&\quad \text{letb } l_p, \sigma_p, \sigma_c = \mathbb{E}^{\sharp}_{p}[\![\,\texttt{tuple.\_\_len\_\_}(o_p)\,]\!](\sigma_p, \sigma_c) \text{ in} \\
&\quad \text{if } \sigma_p = (\mathit{cur}, \_, \_) \text{ then} \\
&\qquad \text{letb } l_c, \sigma_p, \sigma_c = {}^{p}\!\hookrightarrow_{c}^{\sharp}(l_p, \sigma_p, \sigma_c) \text{ in } \mathbb{E}^{\sharp}_{p\times c}[\![\,\mathsf{PyLong\_AsSsize\_t}(l_c)\,]\!](\sigma_p, \sigma_c) \\
&\quad \text{else } \mathsf{convert\_err}^{\sharp}(\sigma_p, \sigma_c)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{convert\_err}^{\sharp}(\sigma_p, \sigma_c) = \\
&\quad \text{let } \mathit{exn}\ e, \epsilon_p, \eta_p = \sigma_p \text{ in} \\
&\quad \text{letb } e_c, \sigma_p, \sigma_c = {}^{p}\!\hookrightarrow_{c}(e, (\mathit{cur}, \epsilon_p, \eta_p), \sigma_c) \text{ in} \\
&\quad \texttt{NULL}, \sigma_p, \mathbb{S}^{\sharp}_{c}[\![\,\mathsf{PyErr\_SetNone}(e_c)\,]\!]\sigma_c
\end{aligned}
$$

Figure 11.5: Tuple access and length semantics, from C

11.2.3 Conversion from a C long to a Python integer

The conversion from a C long to a Python integer, shown in Figure 11.4, is close to the concrete definition Figure 10.10. We explicitly ask for the numerical domain to evaluate the C expression v_c into a mathematical expression v_u. This expression is then passed to the constructor of Python integers, and the boundary is applied afterward.

11.2.4 Tuple conversions

The tuple conversions were defined in the concrete in Figure 10.11. We show the definition in the abstract in Figure 11.5. These functions have exactly the same structure.

11.3 Examples

This section provides detailed examples of the analyses of the Counter instantiation (i.e., the call to `Counter()`, at line 4 of Listing 9.1), the counter access (`c.counter`, at line 8 of Listing 9.1). We end by briefly discussing the use of an underlying relational numerical domain.

Example 11.3 **Counter instantiation**

We show an example of Counter instantiation in Figures 11.6 and 11.7 (corresponding to line 4 of Listing 9.1). We focus solely on the expressions and statements analyzed, and do not display how the state evolves for the sake of concision. The domain handling Python calls (Figure 6.19 in the concrete) evaluates the caller, and then the `__call__` method of the caller's class (here, `type`, since the caller is a class). The domain handling calls to methods of the `type` class catches the evaluation of `type.__call__` (Figure 6.37 in the concrete).

It starts by calling the class creation method, which is a Python function implemented in C, called `tp_new_wrapper` (as shown in Figure 10.5). We reuse the notation $\langle @^\sharp, e\rangle$ of Chapter 8 to denote a Python object having for abstract address $@^\sharp$ and having an optional builtin value described by expression e. This call is handled by the multilanguage domain, and its transfer function was shown in Figure 11.3. We omit the output checks. This domain starts by applying the boundary function to the first argument, and to the second argument bundled into a tuple object (whose resulting address is named $@^\sharp_{\texttt{tuple}}$). In the C semantics, a returned address having no offset will be shown without it. We omit the evaluations created by these boundaries for the sake of concision. In the end, the domain handling calls of C functions is called, to analyze the C implementation of `tp_new_wrapper`. This function extracts the first element of the tuple to call its `tp_new` field (lines 6058 and 6089 of the link). In the `Counter` class, this field is resolved (Figure 10.5) into the builtin function `PyType_GenericNew`, itself calling `PyType_GenericAlloc`. In the abstract, the size computation is done as in the source code (line 1011), but the allocation (lines 1014-1017) is handled by a builtin: we can thus ask the recency abstraction to allocate an address with the correct nominal type. We call this address $@^\sharp_{\mathsf{Inst}(\texttt{Counter})}$. We apply the Python to C boundary to this address in order to ensure that it is well-formed on the C side. The boundary has been formally defined in Figure 11.1. The effects of this boundary are to set the `ob_type` field of the structure to the parent type, and the size of the `struct`. At the end of the call to `tp_new_wrapper`, the C to Python boundary is applied.

Once the instance has been created using `__new__`, we call the initialization method `__init__` (now in Figure 11.7). In the case of the `Counter` class, this method points to an instance of a `wrapper_descriptor` (abbreviated `wd`), as shown in Figure 10.5. These objects are used to wrap checking code around C functions (Remark 10.10). In this case, the descriptor calls the function `wrap_init`, whose source code is directly handled by the C analysis. In turn, this function calls `CounterInit` (Listing 9.2), which sets the counter to

Figure 11.6: Sequence diagram of the analysis of Counter creation (Listing 9.1, line 4) – part 1

0 (omitted in the diagram here), and returns 0. Since the return code was non-negative, `wrap_init` considers the execution has been successful and returns `None`. The conversion of `None` is then handled by the boundary at the end of the C function call from Python.

Figure 11.7: Sequence diagram of the analysis of Counter creation (Listing 9.1, line 4) – part 2

Example 11.4 **Counter access**

We show how the counter value is fetched from the C side and converted to a Python integer in Figure 11.8 (corresponding to line 8 of Listing 9.1). The attribute domain (presented in the concrete in Figure 6.10, and in the abstract in Figure 7.7) handles the first evaluation. It starts by evaluating the left-hand side, yielding the counter instance in our case. Then, the `__getattribute__` method of the `object` class is used to search for the attribute. Its semantics is described in the concrete in Figure 6.31. The attribute exists in the class, it returns an instance of the `member_descriptor` class (abbreviated `md`). Since the `member_descriptor` class has `__get__` and `__set__` attributes, it is a data descriptor. Thus, we call the `__get__` function of the descriptor. This call is handled by the multilanguage domain. It performs the necessary calls to the boundary functions for the parameters (omitted in the diagram), and calls the builtin `member_get` function. In turn, this function will call the builtin `PyMember_GetOne`. We show the statements performed during the execution of `PyMember_GetOne`. We start with the parameters' assignments. Then, the `addr` pointer has

its offset changed by what was provided in the `member_descriptor` definition (Figure 10.5). In that case, the type of data descriptor is `T_INT`, and `PyLong_FromLong` is called to transform the C integer into a Python integer. The abstract transfer function was defined in Figure 11.4. Then, the Python integer object is passed through the boundary and returned.

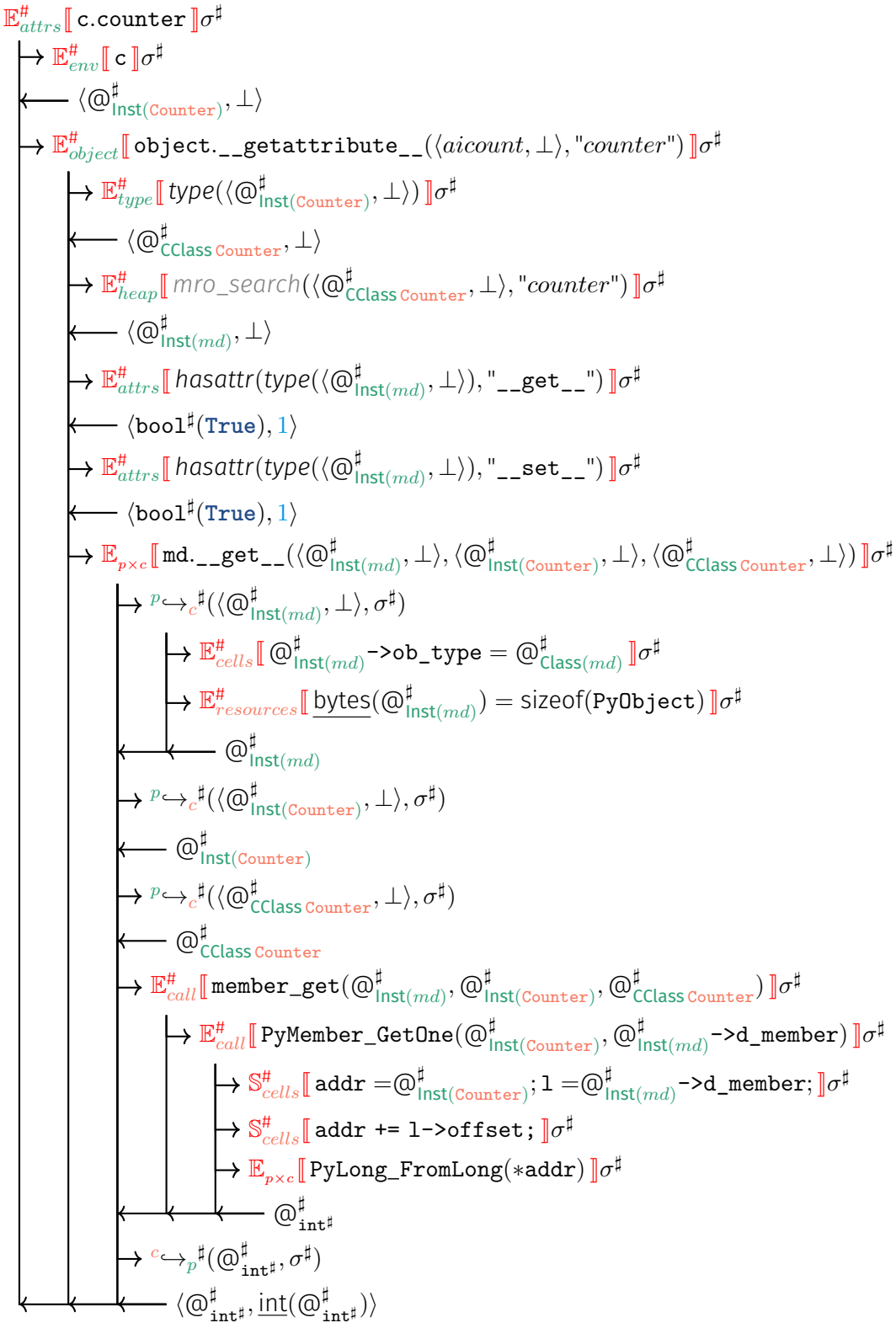

Figure 11.8: Sequence diagram of the analysis of Counter access (Listing 9.1, line 8)

Remark 11.5 **Relational analysis**

Sharing the address allocation and numerical abstractions allows expressing relational invariants between the languages. In the example in Listing 11.2, a non-relational analysis would be able to infer that $0 \leq \texttt{i} \leq 99$, but it cannot infer that the number of calls to `incr` is finite. It would thus infer that $-2^{31} \leq \texttt{r} < 2^{31}$, report an overflow error and be unable to prove the assertion at the end. The value of `r` originates from the C value of the `count` field in the instance defined in `c`. With a relational analysis where C and Python variables are shared in the numerical domains, it is possible to infer that $\underline{\text{int}}(@^\sharp_{\text{int}^\sharp}) + 1 = \underline{\text{int}}(\wr @_{Counter}, 16, \texttt{int} \wr)$. $\underline{\text{int}}(@^\sharp_{\text{int}^\sharp})$ is the numeric value of the integer bound to i. $\underline{\text{int}}(\wr @_{Counter}, 16, \texttt{int} \wr)$ is the numerical value of the `Counter` instance (i.e., the value of `count` in the `Counter` `struct`, here represented as the cell [107] referenced by the `Counter` instance, at offset 16 being a 32-bit integer). Our analyzer is able to prove that the assertion holds using the octagon abstract domain [108].

Listing 11.2: Program where relationality between languages improves precision

```
import counter
from random import randint

c = counter.Counter()
for i in range(randint(1, 100)):
    c.incr()
r = c.counter
assert(r == i+1)
```

11.4 Concretization & soundness

Definition 11.6 **Multilanguage concretization**

We assume we have relational concretization functions for the Universal, Python and C domains:

$$\gamma_U(\sigma^\sharp_u) \in \mathcal{P}(\Sigma_u))$$
$$\gamma_P(\sigma^\sharp_p) \in \mathcal{P}(\mathcal{P}(\Sigma_u) \times \mathcal{P}(\Sigma_p))$$
$$\gamma_C(\sigma^\sharp_c) \in \mathcal{P}(\mathcal{P}(\Sigma_u) \times \mathcal{P}(\Sigma_c))$$

The overall multilanguage concretization ensures that the concretizations of Python and C rely on the same concretized universal state S_u.

$$\gamma(\sigma^\sharp_u, \sigma^\sharp_p, \sigma^\sharp_c) = \{\, (\sigma_p, \sigma_c) \mid \exists S_u, S_u \subseteq \gamma_U(\sigma_u) \\ (S_u, \sigma_p) \in \gamma_P(\sigma^\sharp_p) \\ (S_u, \sigma_c) \in \gamma_C(\sigma^\sharp_c) \,\}$$

Theorem 11.7 **Soundness of the multilanguage analysis**

Assuming the underlying abstract semantics of Python and C are sound, the multilanguage analysis is sound.

Proof. The cases delegating to either the Python or the C language are straightforward, since we assume that the underlying analyses are sound. The only cases left in the soundness proof are those of the operators working at the boundary. Since the abstract semantics of those operators is in point-to-point correspondence with the concrete semantics, the soundness proof is straightforward. □

11.5 Implementation

We have implemented our multilanguage analysis within Mopsa. We were able to reuse off-the-shelf value analyses of C programs [121] and Python programs (the value analysis of Chapter 8) already implemented into Mopsa. The only modification needed was to add a multilanguage domain, implementing the semantics of the operators at the boundary shown in Chapter 10.

11.5.1 Configuration

The configuration for the multilanguage analysis is shown in Figure 11.9. The multilanguage domain is at the top. It dispatches statements not operating at the boundary to the underlying Python or C analysis. The Python and C analyses are in a cartesian product, ensuring that when a statement goes through them, it will be handled by only one of the two sub-configurations. Both Python and C analyses share the underlying "universal" domains, to which they can delegate some statements. Sharing the recency abstraction allows to share the domain of both heap abstractions. The numerical abstraction displayed here only uses intervals, but it can be changed to a reduced product between a relational domain and intervals, as we used in Remark 11.5.

C configuration. The first two lines of the C configuration are iterators handling parts of the C syntax. Each variable or abstract heap block is then decomposed into a set of auxiliary variables called cells, by domain "C.cells". This domain uses the cell abstraction of Miné [107]. It handles transparently union types and type-punning. The decomposition in cells is adapted dynamically according to the actual access patterns during the execution (since the static type can be deceiving in C). The cell domain can be put in a reduced product with a string abstraction, "C.strings", such as the one presented by Journault et al. [81], tracking the position of 0 in character arrays. Both domains rewrite expressions into dereference-free expressions on scalar variables. These are handled by a Cartesian product:

- "C.machineNum" translates machine integer arithmetic with overflows and wrap-arounds, into mathematical arithmetic;
- "C.pointers" translates pointer arithmetic into byte-offset arithmetic while maintaining in its internal abstract state the bases (i.e. pointed-to variables) of each pointer.

Both these domains collaborate to rewrite scalar expressions into expressions on mathematical integers, which are then handled natively by classic numerical abstract domains.

Implementation's length. This multilanguage domain consists in 2,500 lines of OCaml code (measured using the `cloc` tool), implementing 64 builtin functions such as the ones presented in the concrete semantics. This is small compared to the 11,700 lines of OCaml for the C analysis and 12,600 lines of OCaml for the Python analysis. These domains rely on "universal" domains representing 5,600 lines of OCaml and a common framework of 13,200 lines of OCaml.

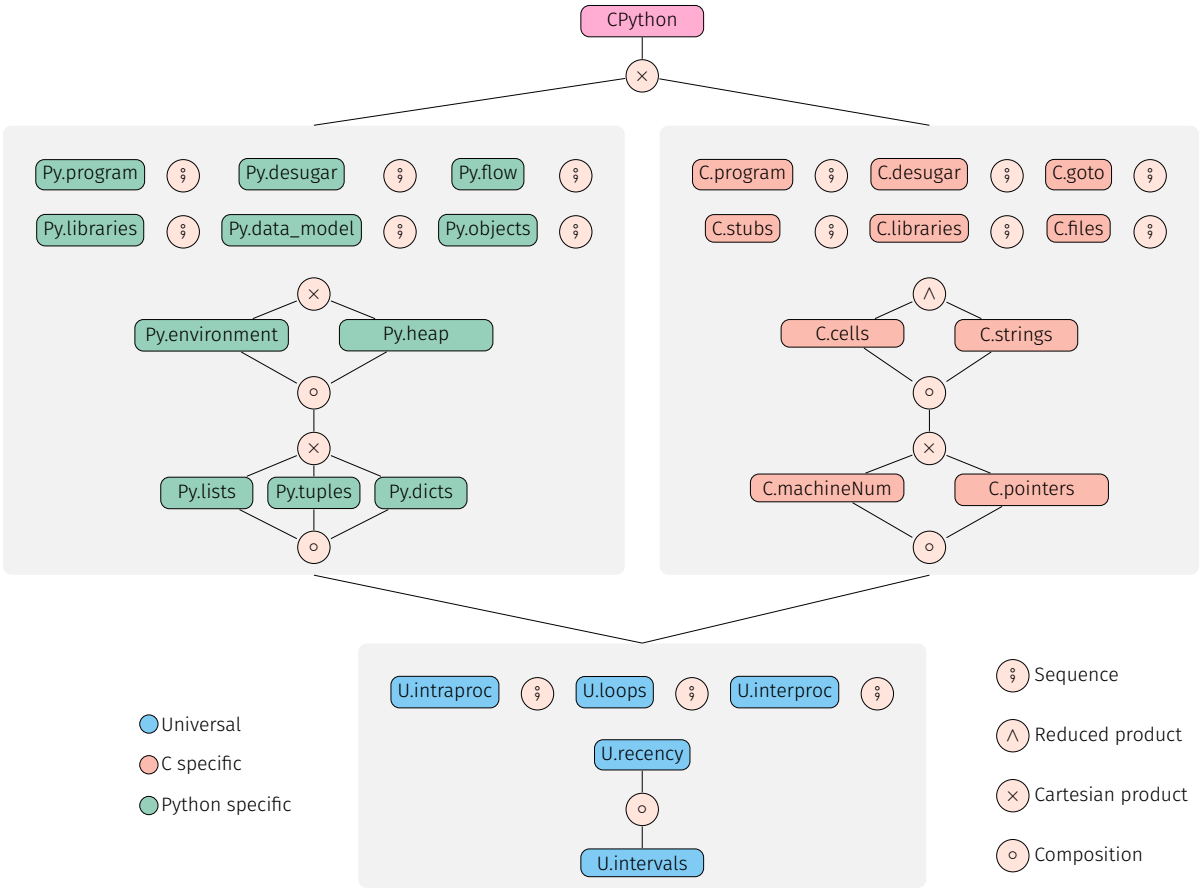

Figure 11.9: Multilanguage configuration in Mopsa

CPython builtins. In addition to the 64 builtins handled by the multilanguage domain, we reuse the C implementation of 60 CPython functions as-is. Potential runtime errors in the API implementation modeled by our concrete semantics as builtins cannot be detected. Assuming that the CPython implementation is correct is a reasonable assumption, as we want to verify client C and Python code. We ensure in our implementation that these builtins are called with valid arguments. However, for the part of the API where our implementation uses the original C implementation, runtime errors would be detected. Currently, half of the API supported reuses the original code of the interpreter.

11.5.2 Build setup

We described how native C modules can be compiled for Python, through a `setup.py` file relying on the `setuptools` library in Section 9.1.8. This build system may be complex and impact the analyzed source code. Both the developer and the `setuptools` library may define compilation flags affecting the source code (e.g., by defining macros), include system (or Python) headers and link with different libraries. We rely on an internal tool call `mopsa-build` to record the C source files and the related compilation flags. Using this tools, we just have to call `mopsa-build python3 setup.py` instead of `python3 setup.py`, before running our analysis. `mopsa-build` intercepts calls to the compiler and to the linker, in order to record these pieces of information. The C frontend of Mopsa, based on `clang`, is then able to process the recorded pieces of information. `mopsa-build` was originally designed to ease the analysis of C programs using a build system.

11.6 Experimental evaluation

11.6.1 Corpus selection

In order to perform our experimental evaluation, we selected six popular Python libraries from GitHub (having in average 412 stars). These libraries are written in C and Python and do not have external dependencies. The `noise` library [49] aims at generating Perlin noise. Libraries `levenshtein`, `ahocorasick`, `cdistance` [67, 117, 103] implement various string-related algorithms. `llist` [72] defines linked-list objects (both single and double ones). `bitarray` [134] provides an implementation for efficient arrays of bits. Our analysis is context-sensitive in order to perform a precise value analysis. Thus, we needed some client code to analyze those libraries. We decided to analyze the tests defined by those libraries: they should cover most use-cases of the library, and ensure that the transition between Python and C are frequent, which is ideal to stress-test our analysis. Some libraries (`noise`, `bitarray`, and `llist`) come with example programs with less than 50 lines of code that we analyze within 15 seconds. We have not been able to find applications with a well-defined entry point using those libraries (or they had big dependencies such as `numpy`). Our experimental evaluation thus focuses on the analysis of the libraries' tests.

Library	|C|	|Py|	Tests	🕓	✓$_c$		✓$_p$		Assert.	Py⇝C
`noise`	722	675	15/15	19s	99.6%	(4952)	100.0%	(1738)	0/21	6.6
`ahocorasick`	3541	1336	46/92	59s	93.1%	(1785)	98.0%	(4937)	30/88	5.4
`levenshtein`	5441	357	17/17	1.6m	79.9%	(3106)	93.2%	(1719)	0/38	2.7
`cdistance`	1433	912	28/28	1.9m	95.3%	(1832)	98.3%	(11884)	88/207	8.6
`llist`	2829	1686	167/194	4.3m	99.0%	(5311)	98.8%	(30944)	235/691	51.7
`bitarray`	3244	2597	159/216	4.6m	96.3%	(4496)	94.6%	(21070)	97/374	14.9

Figure 11.10: Analysis results of libraries using their unit tests

11.6.2 Analysis results

We show the results of our analysis in Figure 11.10. The column |C| (resp. |Py|) shows the lines of code in C (resp. Python), measured using the `cloc` tool. The Python code corresponds mainly to the tests. It may also include Python library code creating custom classes, built upon C classes. For example, `frozenbitarray` is defined in Python, on top of the `bitarray` class. The "tests" column shows the number of tests we are able to analyze, compared to the total number of tests defined by the library. The 🕓 column shows the time taken to analyze all the tests. Columns ✓$_c$ (resp. ✓$_p$) show the selectivity of our analysis – the number of safe operations compared to the total number of runtime error checks, the latter being also displayed in parentheses – performed by the analyzer for C (resp. Python). The selectivity is computed by Mopsa during the analysis, as we explained in Remark 3.14. The C analysis checks for runtime errors including integer overflows, divisions by zero, invalid memory accesses and invalid pointer operations. The Python analysis checks also for runtime errors, which include uncaught `AttributeError`, `TypeError`, `ValueError` exceptions. Runtime errors happening at the boundary are considered as Python errors since they will be raised as Python `SystemError` exceptions. The second to last column shows the number of functional properties (expressed as assertions) defined by the tests that our analyzer is able to prove correct automatically. The last column shows the number of transitions between the analyzed Python code and the C code, averaged per test.

We observe that Mopsa is able to analyze these libraries in a few minutes with high selectivity for the detection of Python and C runtime errors. Our analysis is able to detect some bugs

that were previously known. For example, the `ahocorasick` module forgets to initialize some of its iterator classes, and some functions of the `bitarray` module do not set an exception when they return an erroneous flag, raising a `SystemError` exception. We have not manually checked if unknown bugs were detected by our analysis, and we do not know either how many alarms are false alarms. We have instrumented Mopsa to display the number of crossings (from Python to C, or C to Python). The average number of crossings per test is shown in the last column of Figure 11.10. The minimal number of crossings is one per test. Thus these tests seem correct to benchmark our approach since they all alternate calls to native C code and Python code.

The multilanguage analysis is limited by the current precision level of the underlying C and Python analyses but would naturally benefit immediately from any improvements in these. However, we focused on the multilanguage domains only in this study. We leave as future work the improvements required independently on the C and Python analyses for our benchmarks. We now describe a few areas where the analysis could benefit from improvements. Mopsa is unable to support some tests for now, either because they use unsupported Python libraries or because the C analysis is too imprecise to resolve some pointers. The unsupported tests of the `ahocorasick` analysis are due to imprecisions in the C analysis, which is not able to handle a complex trie data structure being stored in a dynamic array and reallocated over and over again. In `llist`, some tests use the `getrefcount` method of the `sys` module, which is unsupported (and related to CPython's reference-based garbage collector, which we do not support). In addition, some tests make pure-Python classes inherit from C classes: this is currently not supported in our implementation, but it is an implementation detail that will be fixed. For the `bitarray` tests, tests are unsupported because they use the unsupported `pickle` module performing object serialization, or they use the unsupported `sys.getsizeof` method, or they perform some unsupported input-output operations in Python. In addition, the C analysis is too imprecise to resolve some pointers in 18 tests.

The selectivity is lower in the C analysis of `levenshtein`, where dynamic arrays of structures are accessed in loops: the first access at `tab[i].x` may raise an alarm and continue the analysis assuming that `i` is now a valid index access. However, subsequent accesses to `tab[i].y`, `tab[i].z` will also raise alarms as the non-relational numerical domain is unable to express that `i` is a valid index access. Proving the functional properties is more challenging and not the main goal of our analysis, which aims at detecting runtime errors. For example, the assertions of the `noise` library check that the result of complex, iterative non-linear arithmetic lies in the interval $[-1, 1]$. Some assertions in the `llist` or `bitarray` library aim at checking that converting their custom container class to a list preserves the elements. Due to the smashing abstraction [14] of the Python lists, we cannot prove these assertions.

11.7 Related work

11.7.1 Native code analysis

Some works focus on analyzing native C code in the context of language interoperability without analyzing the host language. Tan and Croft [145] perform an empirical study of native code use in Java and provide a classification by bug patterns; a similar study has been performed by Hu and Zhang [70] for the Python/C API. Kondoh and Onodera [85] check that native calls to Java methods should handle raised exceptions using a typestate analysis. Li and Tan [92] ensure that the native control-flow is correctly interrupted when a Java exception is raised. The work of Li and Tan [93, 95] infers which Java exceptions may be raised by JNI code, allowing the exception type-safety property of Java programs to be extended to the JNI. CpyChecker [99] is a GCC plugin searching for common erroneous patterns in C code using the CPython API. Two works [94, 100] aim at detecting reference counting errors in C code using the CPython API. Brown et al. [20] define specialized analyses for specific patterns of C++ interoperability that

may jeopardize type or memory safety of JavaScript. Contrary to these works, we analyze both host and guest languages.

11.7.2 Multilanguage analyses

Buro et al. [24] define a theory based on abstract interpretation to combine analyses of different languages, and show how to lift the soundness property to the multilanguage setting. They provide an example of multilanguage setting where they combine a toy imperative language with a bit-level arithmetic language. The notion of boundary functions used in their work performs a full translation from the state of one language to the other. Our semantics works on the product of the states, although it can be seen as an abstraction of the semantics of C and Python, where the boundary performs a full state conversion (but the boundary from Python to C would be a concretization). From an implementation standpoint, our approach avoids costly state conversions at the boundary and allows sharing some abstract domains.

Chipounov et al. [29] perform symbolic execution of binaries, thus avoiding language considerations. Their approach is extended by the work of Bucur et al. [21], which supports any interpreted language by performing symbolic execution over the interpreter. Our approach is more costly to implement since we do not automatically lift the interpreter's code to obtain our analyzer. Thanks to its higher-level, we think our approach should be more precise and efficient.

The next works compute summaries of the effects of native code on the chosen abstract property in a bottom-up fashion. Those effects are then translated into the host language, where a standard analyzer for the host language can then be used, since the native code has been removed. The use of summaries to convey the abstract meaning of functions makes it easier to rely on independent analyzers for each langage. However, the language and properties we target require precise context-sensitive value analyses that are difficult to perform bottom-up. Since Python is a dynamic programming language with a flexible semantics, it is not possible to analyze programs precisely in a context-insensitive fashion. Additionally, a precise description of the Python heap at a native call is mandatory to analyze the called C code, check for pointer errors, and infer effects. Tan and Morrisett [146] compile C code into an extended JVML syntax form, allowing the use of the bug-finding tool Jlint afterwards. Furr and Foster [57, 58, 59] perform inference of OCaml and Java types in C FFI code, which they crosscheck with the types used in the client OCaml/Java code. They assume there are no out-of-bounds accesses and no type casting in the C code. An inter-language, bottom-up taint analysis for Java and native binary code in the setting of Android applications is proposed by Wei et al. [155]. Lee et al. [90] aim at detecting wrong foreign function calls and mishandling of Java exceptions in Java/JNI code. They extract summaries of the Java callbacks and field accesses from the JNI code using Infer, transform these summaries into Java code, and call the FlowDroid analyzer on the whole. Contrary to these works, our analyzer supports both languages, and it switches between languages just as the real execution does. The properties we target require precise context-sensitive value analyses that are difficult to perform bottom-up.

11.7.3 Library analyses

Previous work aims at analyzing libraries with no access to their client code [1, 130] using a "most-general client". The work of Kristensen and Møller [87] refines the notion of most-general client in the setting of dynamic programming languages. However, it focuses on libraries where functions are typed. Python libraries are not explicitly typed. Extending their work to our untyped, multilanguage setting is left as future work.

11.8 Conclusion

We have defined and implemented a multilanguage analysis for Python programs with native C extensions using the Python/C API. Our analyzer is able to reuse value analyses of Python and C off-the-shelf. It shares the address allocation and numerical abstractions between the Python and C abstract domains. We are able to analyze within a few minutes real-world Python libraries written in C and having up to 5,500 lines of code. Future work includes analyzing programs using the other interoperability mechanisms mentioned in Chapter 9, as well as having a relational analysis that scales. An interesting application of this work would be to verify (or even infer) type annotations of the standard library modules, currently created manually and distributed within the typeshed [152] project.

Part V

Conclusion & Future Work

12

Conclusion & Future Work

This thesis aims at developing and implementing techniques for the static analysis of Python programs.

We have defined the concrete semantics of a large subset of Python, and focused on making it explainable by providing references to the actual implementation's source code, for each case of the semantics. We built upon this semantics to define a concrete multilanguage semantics of Python programs with native C modules relying on the Python/C API as the interoperability mechanism.

The recency abstraction, handles dynamic memory allocations. It was originally tailored for the analysis of low-level code, and subsequently extended to other languages such as JavaScript [74], [124]. We have implemented different allocation-site sensitivities adapted to the analysis of Python programs and its dual type system. We have focused on defining coarse containers abstractions, that work by delegation over scalars domains, and handle dynamic length and heterogeneous types. We have built upon these abstractions to define both a type analysis and a relational value analysis of Python programs. We have compared the type analysis and the value analysis, and noticed that the value analysis does not remove any type-related errors in our benchmarks. In addition, we have built upon Python's value analysis and a C analysis of Ouadjaout and Miné [121] to define a multilanguage analysis of Python and C programs. To the best of our effort, we have strived to build a sound analyzer: except for some features for the Python language that are explicitly not supported (in which case, their use is reported by the analysis), we did not deliberately omit any aspect of the concrete semantics and we performed sound-by-design abstractions without compromise. We took care to link our semantic to the CPython source code and checked the analysis with over 700 conformance tests from various sources (including a fragment of CPython's tests). A more formal and complete proof of soundness of the abstract semantics with respect to the concrete one, and more extensive tests with respect to CPython would however provide more confidence, and are left for future work.

All the abstractions we used have been defined modularly, in order to ensure loose-coupling and cooperation between abstract domains, which we believe is crucial to define precise, relational analyses. We have also taken care to define relational concretization functions modularly.

These abstractions have been implemented into Mopsa. The implementation of our type and value analyses scale to real-world benchmarks used by Python developers to test the performance of their interpreter. This thesis culminated with the multilanguage analysis, which is able to analyze tests of popular, real-world libraries relying on the Python/C API.

There are plenty of future directions left to be explored:

▷ **Modular domains and soundness proofs.** Starting from Chapter 2, we have focused on the development of relational and modular abstract domains. We have also defined concretization operators modularly, in order to make them more compact and loosely coupled. It would be interesting to define other classical abstractions in this framework (disjunctive analyses, shape analyses, ...) prior to their implementation into Mopsa. Another future work is to prove the soundness of each domain modularly, independently of the other abstract domains used in the configuration.

▷ **Executable concrete semantics of Python.** The concrete semantics shown in Chapter 6 is not executable in itself. We currently test it through Python's value analysis, which is a close implementation that may perform overapproximations. It would thus be interesting to have a concrete interpreter mode in Mopsa, or to develop a reference semantics that is executable. In the case of the latter option, several formalisms exist, such as the K framework [133], or skeletal semantics [17]. Another point is that we have only tested our semantics against handcrafted test cases from previous researchers working on the semantics of Python, and from Python's developers. In the case of a reference implementation, it would be interesting to have automatic test case generation that would provide more complete coverage of the defined semantics, and compare it with CPython.

▷ **Unsupported features of Python.** Our analyses ignore finalizers defined in `__del__` methods, and does not support dynamic code evaluation through `eval` and `exec`. We have not encountered these cases in our analysis. Any attempt to support those features would start by studying precisely the context in which those features are used. It would also be interesting to support the asynchronous functions of Python. Mopsa does not support the analysis of recursive functions for now; its framework has to be modified to address this limitation.

▷ **Dictionary abstractions.** We have defined coarse dictionary abstractions in Chapter 5, that lead to imprecisions, as shown in Chapters 7 and 8. These abstractions are currently one of the limiting factors of the analyses' precision. More precise dictionary abstractions are thus an important future work. It would first require a deeper understanding of dictionaries' uses in Python programs. It might be interesting to adapt previous tree abstractions, such as the segmented tree abstraction of Cousot et al. [36].

▷ **Packing strategies.** Value analyses require packing strategies to reduce the computational cost of relational abstract domains such as octagons or polyhedra. We have presented a simple static packing in Chapter 8. It would however be nice to have more dynamic packing strategies. In addition, the static packing presented had to assume what kind of auxiliary variables were defined by other domains, which goes against Mopsa's loose coupling of domains, where a minimal number of assumptions have to be made over domains. A language-agnostic approach to packing could also be applied in the case of multilanguage analyses.

▷ **Function summaries.** Functions are analyzed by inlining them, which is costly and repetitive for those that are called a high number times in a program. We have developed a simple caching solution in Chapter 7. Analyzing a function in isolation is already difficult in the setting of a static programming language such as C, and has been previously studied by Journault [80] for example. In particular, these analyses have to make assumptions on the target of each pointer. In the case of a dynamic programming language such as Python, the complex semantics would require much stronger assumptions. It would however be interesting to have partial function summaries, paving the way for performance improvement, and less global analyses – as we have currently only explored whole program analyses, requiring the full calling context to work. We would also gain support for

the analysis of recursive functions, which are not supported in Mopsa for now. Together with library analyses, we believe these two points are the most important objectives in order to improve scalability.

▷ **Library analyses.** Our current approach focuses on whole-program analyses. Most Python applications rely on libraries. We have relied on a way to handle Python's type annotations in Chapter 7. This allowed us to easily support libraries by leveraging public annotations available from the typeshed [152] project. However, using type annotations added additional assumptions to the soundness of our analysis. In addition, these annotations are not always available. It would thus be interesting to develop alternative techniques to handle libraries and to analyze them. In the case of the multilanguage analysis, it would be interesting to instrument our implementation to verify (or infer) type annotations of the standard library, thus giving back to the typeshed [152] project.

▷ **Other analysis settings.** Our work has focused on the analysis of Python programs, enhanced with C code in the case of Part IV. It would be interesting to check how the techniques we developed would have to be adapted in the analysis of other programming languages, and other multilanguage settings.

Bibliography

[1] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. Combining type-analysis with points-to analysis for analyzing Java library source-code. In *SOAP@Programming Languages Design and Implementation (PLDI)*, 2015.

[2] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Dynamic Languages Symposium (DLS)*, 2007.

[3] Vincenzo Arceri and Isabella Mastroeni. Analyzing dynamic code: A sound abstract interpreter for *Evil* eval. *ACM Trans. Priv. Secur.*, 2021.

[4] Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In *Static Analysis Symposium (SAS)*, 2006.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 2018.

[6] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for Dylan. In *OOPSLA*, 1996.

[7] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C specification. URL `https://frama-c.com/html/acsl.html`.

[8] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Commun. ACM*, 2021.

[9] David M. Beazley and the SWIG developers. Swig, simplified wrapper and interface generator. `http://www.swig.org/`, 2021. Accessed: 2021-08.

[10] Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-Francois Puget. Revising hull and box consistency. In *International Conference on Logic Programming (ICLP)*, 1999.

[11] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static analysis and verification of aerospace software by abstract interpretation. *Foundations and Trends in Programming Languages*, 2015.

[12] Dirk Beyer. Software verification: 10th comparative evaluation (SV-COMP 2021). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2021.

[13] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. RacerD: compositional static race detection. *Proc. ACM Program. Lang.*, 2018.

[14] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*, 2002.

[15] Sandrine Blazy, David Bühler, and Boris Yakobowski. Structuring abstract interpreters through state and value abstractions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2017.

[16] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. In *Principles of Programming Languages (POPL)*, 2014.

[17] Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. Skeletal semantics and their interpretations. *Proc. ACM Program. Lang.*, 2019.

[18] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications*, 1993.

[19] Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[20] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson R. Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in JavaScript bindings. In *IEEE Symposium on Security and Privacy (SP)*, 2017.

[21] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[22] David Bühler. *Structuring an Abstract Interpreter through Value and State Abstractions:EVA, an Evolved Value Analysis for Frama-C. (Structurer un interpréteur abstrait au moyen d'abstractions de valeurs et d'états :Eva, une analyse de valeur évoluée pour Frama-C).* PhD thesis, University of Rennes 1, France, 2017.

[23] Samuele Buro and Isabella Mastroeni. On the multi-language construction. In *European Symposium on Programming (ESOP)*, 2019.

[24] Samuele Buro, Roy L. Crole, and Isabella Mastroeni. On multi-language abstraction - towards a static analysis of multi-language programs. In *Static Analysis Symposium (SAS)*, 2020.

[25] Bor-Yuh Evan Chang, Cezara Drăgoi, Roman Manevich, Noam Rinetzky, and Xavier Rival. Shape analysis. *Foundations and Trends in Programming Languages*, 2020.

[26] Rebecca Chen and the Pytype development team. Pytype. `https://github.com/google/pytype`, 2021. Accessed: 2021-08.

[27] Marc Chevalier. *Proving the security of software-intensive embedded systems by abstract interpretation*. PhD thesis, Ecole Normale Supérieure, Paris Science et Lettres, France, 2020.

[28] Marc Chevalier and Jérôme Feret. Sharing ghost variables in a collection of abstract domains. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2020.

[29] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[30] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 1986.

[31] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

[32] Patrick Cousot. Types as abstract interpretations. In *Principles of Programming Languages (POPL)*, 1997.

[33] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, 1977.

[34] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*, 1978.

[35] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the Astrée static analyzer. In *ASIAN*, 2006.

[36] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. A scalable segmented decision tree abstract domain. In *Essays in Memory of Amir Pnueli*, 2010.

[37] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Principles of Programming Languages (POPL)*, 2011.

[38] Arlen Cox, Bor-Yuh Evan Chang, and Sriram Sankaranarayanan. QUIC graphs: Relational invariant generation for containers. In *ECOOP*, 2013.

[39] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Automatic analysis of open objects in dynamic language programs. In *Static Analysis Symposium (SAS)*, 2014.

[40] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - A software analysis perspective. In *Software Engineering and Formal Methods (SEFM)*, 2012.

[41] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[42] David Delmas and Antoine Miné. Analysis of software patches using numerical abstract interpretation. In *Static Analysis Symposium (SAS)*, 2019.

[43] David Delmas and Jean Souyris. Astrée: From research to industry. In *Static Analysis Symposium (SAS)*, 2007.

[44] David Delmas, Abdelraouf Ouadjaout, and Antoine Miné. Static analysis of endian portability by abstract interpretation. In *Static Analysis Symposium (SAS)*, 2021.

[45] The Infer development team. Infer, a static analysis tool for java, c++, objective-c, and c., 2021. URL `https://github.com/facebook/infer`.

[46] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *European Symposium on Programming (ESOP)*, 2010.

[47] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *Principles of Programming Languages (POPL)*, 2011.

[48] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *Commun. ACM*, 2019.

[49] Casey Duncan. Native-code and shader implementations of perlin noise for Python. `https://github.com/caseman/noise`, 2021. Accessed: 2021-04.

[50] Greg Erwing and the Cython development team. Cython: C-extensions for python. `https://cython.org/`, 2021. Accessed: 2021-08.

[51] N. Van Es, Q. Stiévenart, and C. De Roover. Garbage-free abstract interpretation through abstract reference counting. In *ECOOP*, 2019.

[52] Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In *European Symposium on Programming (ESOP)*. Springer, 2013.

[53] Levin Fritz and Jurriaan Hage. Cost versus precision for approximate typing for Python. In *Partial Evaluation and Program Manipulation (PEPM)*, 2017.

[54] Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. Static value analysis of python programs by abstract interpretation. In *Nasa Formal Methods (NFM)*, 2018.

[55] Jedrzej Fulara. *Abstract Analysis of Numerical and Container Variables*. PhD thesis, Uniwersytet Warszawski.

[56] Jedrzej Fulara. Generic abstraction of dictionaries and arrays. *Electron. Notes Theor. Comput. Sci.*, 2012.

[57] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. In *Programming Languages Design and Implementation (PLDI)*, 2005.

[58] Michael Furr and Jeffrey S. Foster. Polymorphic type inference for the JNI. In *European Symposium on Programming (ESOP)*, 2006.

[59] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. *ACM Trans. Program. Lang. Syst.*, 2008.

[60] Graeme Gange, Zequn Ma, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. A fresh look at zones and octagons. *ACM Trans. Program. Lang. Syst.*, 2021.

[61] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Principles of Programming Languages (POPL)*, 2016.

[62] GitHub. State of the github octoverse. `https://octoverse.github.com`, 2021. Accessed: 2021-09.

[63] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

[64] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *Principles of Programming Languages (POPL)*, 2005.

[65] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 1989.

[66] Dwight Guth. A formal semantics of Python 3.3. Master's thesis, University of Illinois, 2013. URL `https://www.ideals.illinois.edu/bitstream/handle/2142/45275/Dwight_Guth.pdf?sequence=1&isAllowed=y`.

[67] Antti Haapala, Esa Määttä, Jonatas CD, Mikko Ohtamaa, and David Necas. Levenshtein Python C extension module. `https://github.com/ztane/python-Levenshtein/`, 2021. Accessed: 2021-04.

[68] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *Programming Languages Design and Implementation (PLDI)*, 2008.

[69] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. MaxSMT-based type inference for Python 3. In *Computer Aided Verification (CAV)*, 2018.

[70] Mingzhe Hu and Yu Zhang. The Python/C API: evolution, usage statistics, and bug patterns. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2020.

[71] S. Jagannathan, P. Thiemann, S. Weeks, and A. K. Wright. Single and loving it: Must-alias analysis for higher-order languages. In *Principles of Programming Languages (POPL)*, 1998.

[72] Adam Jakubek and Rafał Gałczyński. Linked lists for CPython. `https://github.com/ajakubek/python-llist`, 2021. Accessed: 2021-04.

[73] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification (CAV)*, 2009.

[74] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Static Analysis Symposium (SAS)*, 2009.

[75] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Static Analysis Symposium (SAS)*, 2010.

[76] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of javascript web applications. In *SIGSOFT Foundations of Software Engineering FSE*, 2011.

[77] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[78] M. Journault, A. Miné, R. Monat, and A. Ouadjaout. MOPSA: modular open platform for static analysis. `https://gitlab.com/mopsa/mopsa-analyzer`, 2021. Accessed: 2021-04.

[79] M. Journault, A. Miné, R. Monat, and A. Ouadjaout. Benchmarks used by MOPSA. `https://gitlab.com/mopsa/benchmarks`, 2021. Accessed: 2021-09.

[80] Matthieu Journault. *Precise and modular static analysis by abstract interpretation for the automatic proof of program soundness and contracts inference*. PhD thesis, Sorbonne Université, France, 2019.

[81] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. Modular static analysis of string manipulations in C programs. In *Static Analysis Symposium (SAS)*, 2018.

[82] Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, 2019.

[83] Sven Keidel and Sebastian Erdweg. Sound and reusable components for abstract interpretation. *Proc. ACM Program. Lang.*, 2019.

[84] Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. Compositional soundness proofs of abstract interpreters. *Proc. ACM Program. Lang.*, 2018.

[85] Goh Kondoh and Tamiya Onodera. Finding bugs in Java native interface programs. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

[86] Herb Krasner. The cost of poor software quality in the us: A 2020 report. `https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf`, 2021. Accessed: 2021-08.

[87] Erik Krogh Kristensen and Anders Møller. Reasonably-most-general clients for JavaScript library analysis. In *International Conference of Software Engineering (ICSE)*, 2019.

[88] Maximilian A. Köhl. An executable structural operational formal semantics for python. Master's thesis, Saarland University – Department of Computer Science, 2021. URL `https://arxiv.org/abs/2109.03139`.

[89] Jeffrey C Lagarias. The 3x+ 1 problem: An overview. *American Mathematical Society*, 2010.

[90] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. Broadening horizons of multilingual static analysis: Semantic summary extraction from C code for JNI program analysis. In *Automated Software Engineering (ASE)*, 2020.

[91] Jukka Lehtosalo and the Mypy development team. Mypy. `http://mypy-lang.org/`, 2021. Accessed: 2021-08.

[92] Siliang Li and Gang Tan. Finding bugs in exceptional situations of JNI programs. In *Computer and Communications Security (CCS)*, 2009.

[93] Siliang Li and Gang Tan. JET: exception checking in the Java native interface. In *SPLASH*, 2011.

[94] Siliang Li and Gang Tan. Finding reference-counting errors in Python/C programs with affine analysis. In *ECOOP*, 2014.

[95] Siliang Li and Gang Tan. Exception analysis in the Java Native Interface. *Sci. Comput. Program.*, 2014.

[96] P. Liang, O. Tripp, M. Naik, and M. Sagiv. A dynamic evaluation of the precision of static heap abstractions. In *OOPSLA*, 2010.

[97] Jiangchao Liu and Xavier Rival. An array content static analysis based on non-contiguous partitions. *Comput. Lang. Syst. Struct.*, 2017.

[98] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 2015.

[99] David Malcolm. A static analysis tool for cpython extension code. `https://gcc-python-plugin.readthedocs.io/en/latest/cpychecker.html`, 2018. Accessed: 2021-04.

[100] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. RID: finding reference count bugs with inconsistent path pair checking. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[101] Daniel Marino and Todd D. Millstein. A generic type-and-effect system. In *Types in Language Design and Implementation (TLDI)*, 2009.

[102] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.*, 2009.

[103] Michaël Meyer. Distance library. `https://github.com/doukremt/distance`, 2021. Accessed: 2021-04.

[104] M. Might and O. Shivers. Improving flow analyses via γcfa: abstract garbage collection and counting. In *International Conference on Functional Programming (ICFP)*, 2006.

[105] A. Miné. Static analysis by abstract interpretation of concurrent programs. Technical report, École normale supérieure, May 2013. `http://www-apr.lip6.fr/~mine/hdr/hdr-compact-col.pdf`.

[106] A. Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages (FnTPL)*, 2017.

[107] Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Languages, Compilers, Tools and Theory of Embedded Systems (LCTES)*, 2006.

[108] Antoine Miné. The octagon abstract domain. *High. Order Symb. Comput.*, 2006.

[109] Antoine Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science*, 2012.

[110] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Static type analysis by abstract interpretation of python programs. In *ECOOP*, 2020.

[111] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Static type analysis by abstract interpretation of python programs (artifact). *Dagstuhl Artifacts Ser.*, 2020.

[112] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Value and allocation sensitivity in static Python analyses. In *SOAP@Programming Languages Design and Implementation (PLDI)*, 2020.

[113] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. A multilanguage static analysis of python programs with native c extensions. In *Static Analysis Symposium (SAS)*, 2021.

[114] Raphaël Monat. Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries (Artifact), 2021. URL `https://doi.org/10.5281/zenodo.5510486`.

[115] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. A Multi-Language Static Analysis of Python Programs with Native C Extensions, July 2021. URL `https://doi.org/10.5281/zenodo.5141314`.

[116] David Monniaux and Francesco Alberti. A simple abstraction of arrays and maps by program translation. In *Static Analysis Symposium (SAS)*, 2015.

[117] Wojciech Muła and Philippe Ombredanne. Pyahocorasick library. `https://github.com/WojciechMula/pyahocorasick`, 2021. Accessed: 2021-04.

[118] Ionel Cristian Mărieș. Understanding python metaclasses, 2015. URL `https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/`.

[119] Benjamin Barslev Nielsen and Anders Møller. Value partitioning: A lightweight approach to relational static analysis for javascript. In *ECOOP*, 2020.

[120] National Institude of Standards and Technology. NIST's national vulnerability database. `https://nvd.nist.gov/vuln/search`, 2021. Accessed: 2021-08.

[121] Abdelraouf Ouadjaout and Antoine Miné. A library modeling language for the static analysis of C programs. In *Static Analysis Symposium (SAS)*, 2020.

[122] Alain OURGHANLIAN. Evaluation of static analysis tools used to assess software important to nuclear power plant safety. *Nuclear Engineering and Technology*, 2015. doi:10.1016/j.net.2014.12.009. URL `https://hal-edf.archives-ouvertes.fr/hal-01857446`.

[123] Daejun Park, Andrei Stefanescu, and Grigore Rosu. KJS: A complete formal semantics of JavaScript. In *Programming Languages Design and Implementation (PLDI)*, 2015.

[124] Jihyeok Park, Xavier Rival, and Sukyoung Ryu. Revisiting recency abstraction for javascript: towards an intuitive, compositional, and efficient heap abstraction. In *SOAP@Programming Languages Design and Implementation (PLDI)*, 2017.

[125] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: The full monty. In *OOPSLA*, 2013.

[126] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In *Computer Aided Verification (CAV)*, 2020.

[127] Ranson, Hamilton, and Fong. A semantics of Python in Isabelle/HOL. Technical report, University of Regina, 2008. URL `http://www.cs.uregina.ca/Research/Techreports/2008-04.pdf`.

[128] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 1953.

[129] Armin Rigo and Maciej Fijalkowski. Cffi, C Foreign Function Interface for Python. `https://cffi.readthedocs.io/en/latest/`, 2021. Accessed: 2021-08.

[130] Noam Rinetzky, Arnd Poetzsch-Heffter, Ganesan Ramalingam, Mooly Sagiv, and Eran Yahav. Modular shape analysis for dynamically encapsulated programs. In *European Symposium on Programming (ESOP)*, 2007.

[131] Xavier Rival. Understanding the origin of alarms in astrée. In *Static Analysis Symposium (SAS)*, 2005.

[132] Xavier Rival and Kwangkeun Yi. *Introduction to Static Analysis*. MIT Press, 2020.

[133] Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebr. Program.*, 2010.

[134] Ilan Schnell. Bitarray library. `https://github.com/ilanschnell/bitarray`, 2021. Accessed: 2021-04.

[135] Holger Siegel and Axel Simon. Summarized dimensions revisited. *Electron. Notes Theor. Comput. Sci.*, 2012.

[136] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *ECOOP*, 2007.

[137] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Making numerical program analysis fast. In *Programming Languages Design and Implementation (PLDI)*, 2015.

[138] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Fast polyhedra abstract domain. In *Principles of Programming Languages (POPL)*, 2017.

[139] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. A practical construction for decomposing numerical abstract domains. *Proc. ACM Program. Lang.*, 2018.

[140] Victor Skvortsov. Python behind the scenes, 2020. URL `https://tenthousandmeters.com/tag/python-behind-the-scenes/`.

[141] Gideon Joachim Smeding. An executable operational semantics for Python. *Universiteit Utrecht*, 2009. URL `http://gideon.smdng.nl/wp-content/uploads/thesis.pdf`.

[142] Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. Static analysis with demand-driven value refinement. *Proc. ACM Program. Lang.*, 2019.

[143] Victor Stinner and the Python Benchmark Suite team. Performance benchmarks from Python's reference interpreter. `https://github.com/python/pyperformance/`, 2021. Accessed: 2021-08.

[144] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In *Principles of Programming Languages (POPL)*, 2016.

[145] Gang Tan and Jason Croft. An empirical security study of the native code in the JDK. In *USENIX*, 2008.

[146] Gang Tan and Greg Morrisett. Ilea: inter-language analysis across Java and C. In *OOPSLA*, 2007.

[147] The Pyre development team. Pyre-check. `https://github.com/facebook/pyre-check`, 2021. Accessed: 2021-08.

[148] The Python Development Team. Parsing arguments and building values. `https://docs.python.org/3.8/c-api/arg.html`, 2021. Accessed: 2021-12.

[149] The Python Development Team. Ctypes, a foreign function library for Python. `https://docs.python.org/3.8/library/ctypes.html`, 2021. Accessed: 2021-08.

[150] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, July 2009.

[151] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 1937.

[152] Typeshed contributors. Typeshed. `https://github.com/python/typeshed/`, 2021. Accessed: 2021-04.

[153] Guido van Rossum and the Python development team. Python/C API Reference Manual. `https://docs.python.org/3.8/c-api/index.html`, 2021. Accessed: 2021-04.

[154] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. Python Enhancement Proposal 484. `https://www.python.org/dev/peps/pep-0484/`, 2021. Accessed: 2021-08.

[155] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. JN-SAF: precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of Android applications with native code. In *Computer and Communications Security (CCS)*, 2018.

[156] W. Eric Wong, Xue-Lin Li, and Phillip A. Laplante. Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures. *J. Syst. Softw.*, 2017.

[157] Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. Python predictive analysis for bug detection. In *SIGSOFT Foundations of Software Engineering FSE*, 2016.

[158] Boris Yakobowski. Fast whole-program verification using on-the-fly summarization. In *Workshop on Tools for Automatic Program Analysis*, 2015.

List of Figures

List of Listings

List of Definitions, Examples, Properties and Remarks