SORBONNE UNIVERSITÉ

THÈSE DE DOCTORAT DE SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Matthieu Journault

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Sujet de la thèse :

Analyse statique modulaire précise par interprétation abstraite pour la preuve automatique de correction de programmes et pour l'inférence de contrats.

soutenue le 21 Novembre 2019

devant le jury composé de :

M. Antoine Miné	Directeur de thèse
Mme. Sandrine Blazy	Rapporteure
M. Andy King	Rapporteur
M. Emmanuel Chailloux	Examinateur
M. Tristan Le Gall	Examinateur
M. Pascal Sotin	Examinateur

Résumé

Assurer le passage à l'échelle des analyseurs statiques définis par interprétation abstraite pose des difficultés à la fois techniques et pratiques. Une méthode classique d'accélération consiste en la découverte et la réutilisation de contrats satisfaits par certaines commandes du code source (par exemple, le corps d'une fonction). Cette thèse s'intéresse à un sous-ensemble de C qui ne permet pas la récursivité, pour lequel on définit un analyseur modulaire capable d'inférer, de prouver et d'exploiter de tels contrats. Ces contrats sont: (1) précis: ils permettent d'exprimer une relationnalité numérique entre les états d'entrée et de sortie à travers l'utilisation de domaines numérique relationnels; (2) corrects: utiliser un contrat précédemment inféré pour une commande produit un état de sortie surapproximant l'ensemble des états effectivement atteignables à la fin de ladite commande; (3) dirigés par les entrées: des contrats ne sont découverts que pour les contextes d'appel rencontrés pendant l'analyse; si une divergence apparait, un opérateur d'élargissement est utilisé pour stabiliser les contextes d'entrée. Notre analyseur modulaire est défini au dessus d'un analyseur C existant et est donc capable de manipuler des types unions, des types structures, des tableaux, des allocations de mémoire (statique et dynamique), des pointeurs, y compris l'arithmétique de pointeur et le transtypage, appels de fonction, des chaînes de caractères, ... La représentation des chaînes de caractère est gérée par un nouveau domaine abstrait défini dans cette thèse. Ce domaine abstrait de chaînes stocke avec précision la longueur des chaînes C (position du premier `'\0'`) sans recourir à une représentation explicite des tableaux, qui peut devenir coûteuse, ce qui est utile pour prouver l'absence d'accès hors mémoire lors de manipulations de chaîne.

La plupart des domaines abstraits numériques utilisés dans le cadre de l'interprétation abstraite se concentrent sur la représentation d'ensembles de fonctions numériques partageant le même ensemble de définition. Si des ensembles hétérogènes apparaissent pendant une analyse, la plupart des analyseurs ont recours à du partitionnement ou à des méthodes non relationnelles. Dans cette thèse, nous proposons une technique paramétrique de transformation de la sémantique classique des domaines abstraits vers une sémantique d'ensembles hétérogènes. Cette technique peut être utilisée pour des domaines relationnels et ne maintient qu'un seul état abstrait numérique, par opposition au partitionnement. De plus, lorsque l'ensemble est homogène, on retrouve la précision et la complexité des domaines classiques.

Le dernier point d'intérêt de cette thèse est la définition d'un domaine abstrait capable de représenter des ensembles d'arbres dont les feuilles peuvent contenir des labels numériques. Cette abstraction est basée sur les langages régulier et les langages d'arbre régulier, et délègue une partie de son abstraction à un domaine numérique sous-jacent capable de représenter des ensembles d'environnements hétérogènes. Cette abstraction d'arbres a plusieurs applications parmi lesquelles: la représentation de la mémoire pour les langages fournissant des structures arborescentes, le développement d'un domaine sauvegardant des égalités symboliques entre les variables et les expressions dans un programme, le développement d'une pré-analyse pour le language C, fournissant des informations de framing.

Cette thèse s'étant déroulée dans le projet Mopsa, nous donnons un aperçu de certains résultats obtenus par l'équipe pendant la thèse. De plus, tous les domaines abstraits présentés ici ont été implantés dans l'analyseur statique Mopsa, une discussion sur certains résultats expérimentaux préliminaires est fournie pour chaque domaine.

Abstract

Ensuring the scalability of static analyzers defined by abstract interpretation poses both theoretical and practical difficulties. A classical technique known to speed up analyses is the discovery and reuse of summaries for some of the sequences of statements of the source code (e.g. functions bodies). In this thesis we focus on a subset of C that does not allow recursion and define a modular analyzer, able to infer, prove and use (to improve the efficiency) such summaries. The summaries considered here are (1) precise: they allow input/output numerical relationality via the use of relational numerical domain; (2) sound: the use of a summary inferred for a statement represents an over-approximation of the reachable set of states at the end of the considered statement; (3) inputs oriented: summaries are inferred only for calling contexts met during the analysis, if a divergence occurs a widening operator is used to stabilize input contexts. Our modular analyzer is built on top of a existing C analyzer and is therefore able to handle unions, structures, arrays, memory allocations (static and dynamic), pointers, pointer arithmetics, pointer casts, function calls, string manipulations,.... String handling is provided by a new abstract domain defined in this thesis, this string abstract domain aims at precisely tracking the length of C strings (position of the first `'\0'`), which is helpful to prove the absence of out-of-memory accesses in string manipulations.

Most numerical abstract domains used in the abstract interpretation framework focus on the representation of sets of numerical maps that all share the same definition set. When the need arises to represent heterogeneous sets of maps, most analyzers fall back to the use of partitioning. In this thesis we provide a lifting of classically used abstract domains to the representation of heterogeneous sets. This lifting can be used for relational domains and maintains only one numerical abstract state, by opposition to partitioning. Moreover when the set is homogeneous the lifted domain behaves as the underlying numerical domain in terms of both precision and efficiency.

The last point of interest of this thesis is the definition of an abstract domain able to represent sets of trees with numerically labeled leaves. This abstraction is based on regular and tree regular languages and delegates the handling of numerical constraints to an underlying domain able to represent heterogeneous sets of environments. This tree abstraction has several applications among which: memory representation for languages providing tree-like structures, the development of a domain tracking symbolic equalities between variables and expressions in a program, the development of a pre-analyzer providing framing information for the C language.

As the thesis took place in the Mopsa project, we provide an overview of some of the results obtained by the Mopsa team during the thesis. Finally, all domains presented in the thesis have been implemented as abstract domains in the Mopsa static analyzer, therefore some preliminary experimental results are presented and discussed in the thesis.

Contents

Chapter 1

Introduction

The increasing usage of software in various fields (aeronautic industry, medical world, army, ...) is an area of concern as we have witnessed in the past years several failures of such software. As examples, let us mention the Ariane 5 failure (caused by an integer overflow and an unhandled exception) [Le 97], the Patriot missile battery at Dhahran [Ske92], and finally the infamous Therac-25 radiation therapy machine which administered excessive quantities of beta radiation [LT93].

In order to prevent these from happening, we first need to detect that such failures may occur. Two approaches can be taken in catching these failures. The first method is to add dynamic testing, meaning that while the software is running it self-checks its behavior, in case of an error the execution is stopped, potentially leading to other failures. The second method is to discover before the execution of the program whether bugs will occur or not. This can be achieved by testing the program before its deployment, by reviewing the code or finally by using *formal methods* to obtain mathematical results about the behavior of the program. Among formal methods we can cite: program proving, model-checking and static analysis, all of which are described in the following. A particular attention will be given to *static analysis* as this is the context of this thesis.

Such failures are moreover very hard to discover by hand and it is even harder to prove the absence of such “bugs” without automation, due to the size of real life software and to the fact that the configuration space might be unbounded. In the process of automating this discovery, one is faced with the theoretical limit that: there exists no programs, taking programs as input and deciding whether or not they terminate [Tur37]. This result is extended by Rice’s theorem [Ric53] that states that all non-trivial properties about the language recognized by a Turing machine are undecidable. The set of all possible executions of a software is called its *semantics* and is defined by the programming language. The problem of proving the absence of bugs requires to infer and prove that the semantics of the program satisfies some properties, which can be seen as proving that the semantics of the targeted program is included into the set of all executions satisfying this property. The results above ensure that semantics are not computable in general, moreover they might not be representable in a computer.

As for other fields of computer science, a notion of *soundness* and *completeness* are defined. We say that a static analysis is sound with respect to the concrete semantics when the set of program executions considered by the analysis is greater than the actual semantics, conversely we say that it is complete when it is lower. As we have seen, a sound, complete and automatic analyzer able to handle all programs can not be designed. This leaves us with several choices: (1) giving up on Turing complete languages; (2) choosing any two of the three properties (sound, complete, automated) and giving up on the last. These observations led to the creation and development of several fields of computer science, their objective ranging from discovering bugs in software to mathematically proving that all executions of a program behave as its specifications asserts. The work presented in this thesis is automated and sound but is not complete in general.

In the next section we provide a small overview of some of the fields of program verification.

1.1 Program verification

1.1.1 Tests

In most cases, the only static analysis performed on programs is testing. Testing consists in launching the targeted program on some inputs (a finite number) and verifying that the property holds on this particular execution. Testing can be done by hand, by executing the program on some well-chosen inputs, or automatically by generating inputs (e.g. see [CH00]). This generation can be uniform and/or guided by the program. Note that, for a program failing on only one of its 2^{32} possible inputs (imagine for example an integer overflow), uniform testing will find the error with probability 2^{-32}. Moreover each input requires a run of the targeted program, which may be slow to execute or might not terminate at all. In any cases, if the input space is not bounded, testing is a complete and automated but unsound technique to prove that some property holds on the semantics of a program. Indeed only a finite number of executions can be checked.

1.1.2 Model Checking

Model Checking [CGP99, QS82] is a sound and complete verification process for transition systems that are finite or enjoy more regularity than that provided by Turing complete programming languages. The set of properties that can be checked is restricted to some logic (e.g. CTL, LTL), and the framework provides algorithms that can check whether a given structure is a model of a formula of this logic or not. When starting from a transition system too expressive for model checking, one can sometimes refine it by hand (by removing information useless to proving the targeted property) to its core behavior and perform model checking on the result. A classical introductory example of model checking is the beverage dispenser machine. This machine has several (a finite number) states and changes states according to user inputs, incoming money, beverage being dispensed, ... The property – *if I input enough money, I will get a beverage at some point* – can be encoded into a temporal logic, a predefined algorithm can thus check whether the description of the beverage dispensing machine satisfies the above property.

1.1.3 Program proving

One can analyze a program by hand by providing and proving properties satisfied by the semantics of the program. This can be helped and in some place automated using theorem provers (see for example Coq). This approach may lead to complete and sound reasoning about the semantics of a program, hence statically providing very powerful properties about executions of the program such as: this function adds an element to the tree and yields a balanced tree. Type systems, in a language such as OCaml, are a particular case of program proving restricted to a subset $\mathcal{P}$ of properties for which inferring and proving the strongest property satisfied by the program in $\mathcal{P}$ is decidable and has a "reasonable" time complexity (an example of property is `f: 'a -> 'a`, stating that `f` always outputs a value of the same type as its input). Program proving has been used successfully in the past years, leading to the development of software that are mathematically proved to follow their specifications (e.g. CompCert [Ler09]). Program proving is very powerful, but demands that programmers write proofs that their programs satisfy (potentially complicated) properties. This can be done as a part of the development process for some targeted software, however for existing codes and for most of the software written in industry this does not seem achievable. Deductive verification tools (such as Why3) only asks the user to provide invariants and contracts for loops and functions from the source code. Proof goals for these contracts and invariants are automatically generated, provers (such as Z3) then try to automatically prove these goals. This offers a less powerful but more automatic approach to theorem proving than theorem provers.

1.1.4 Abstract interpretation

Abstract interpretation [CC77b] is a theory formalizing semantics approximations. As the semantics of programs is not computable we build a sequence of semantics, each being an abstraction of the previous one (the notion of abstract/concrete semantics is relative). The goal of building such sequences is to forget information from the original semantics, the choice of how precision is lost being guided by the targeted properties. The last abstract semantics of the sequence is designed so that: (1) elements of this abstract semantics are machine representable; (2) usual semantics operators (union, transitions in the original system, fixpoint computations) are computable. As abstractions might lose information about the original semantics, results discovered on the abstract semantics may not always be exact. The goal of abstract interpretation being to certify software, we want to obtain properties of the form “no integer overflow will occur during the execution of this program”. Therefore the information loss yielding computability results can not be obtained by removing elements from the original semantics, otherwise it would be possible in the abstract worlds to observe the absence of integer overflow without it implying the same in the original world. For this reasons we chose to over-approximate the concrete semantics during the approximation steps. As mentioned above, the semantics S of a program (and more generally of a transition system) is the set of all its possible executions, a property can be defined as the set of all execution that do satisfy this property $\mathcal{P}$. Proving that the semantics satisfy the property, amounts to showing that $\mathsf{S} \subseteq \mathcal{P}$, it is therefore sufficient to show that a set of states containing S is included in the property $\mathcal{P}$.

Abstract interpretation is fully automatic, in the sense that the user designs *abstract domains* which will be able to precisely represent a (potentially unbounded) subset of all the possible properties. Abstract domains will then be able, given a program, to infer and prove that some property holds on the semantics of this program. As we mentioned, information may be lost due to over-approximations and therefore abstract interpretation provides sound but incomplete analyzers. This induces false positives being potentially reported by the analyzer.

The above mentioned abstract domains enjoy several composition mechanisms that can be used to target different input language features, or even refine each other in order to gain precision. This allows the designers of abstract domains to focus on building high precision analyses for small features of the language, as these will then be composed with already existing abstract domains tackling the rest of the language.

Abstract interpretation has enjoyed a growing success, as witnessed by commercial tools used in the industry: Polyspace Verifier, Astrée [KWN+10], Sparrow [OHL+12], Julia [Spo05], etc. Astrée was able to prove the absence of run-time errors in real life critical software such as the flight control software of the Airbus A340 fly-by-wire system, a program of 132,000 lines of C. Open-source industrial-strength frameworks to design analyzers by abstract interpretation have also been proposed: Frama-C [CKK+12], IKOS [BNSV14], Infer [CDD+15a], etc.

1.2 Content of the thesis

This thesis was made in the scope of Abstract interpretation. In this section we provide an overview of the different topics addressed in this thesis, as well as the reasons why these topics were chosen. These can be sorted into 5 categories: modularity, strings, numeric, trees, analyzer design and implementation.

Modularity. Modularity is to be understood in the sense that the analysis of a large software can be obtained as a composition of the analysis of the different parts composing the software

```
int incr(int x) {
  int y = x+1;
  return (y);
}

int main() {
  int a = incr(0);
  int b = incr(1);
  …
}
```

Program 1.1: Modularity example

```
while (*q != '\0') {
  *p = *q;
  p++;
  q++;
}
*p = *q;
```

Program 1.2: C String copy

(this can be applied at different scales: functions, libraries, …). Radhia Cousot and Patrick Cousot mention in [CC02], that performing modular analysis is a way to improve scalability of analyzers defined by abstract interpretation. Indeed as software may contain hundreds of thousands lines of codes, the ability to reduce the complete analysis to the composition of analysis results of its different parts may induce a great speed up. Moreover if, during the development, a component is not modified between two versions of the software, the results of its analysis in the first version may be reused for the second one, leading to incremental analyses. In the above presentation, we mentioned that the goal of analyzers defined by abstract interpretation is to compute an over-approximation of the semantics of a program. This over-approximation is obtained by following the definition of the original semantics of the program, by replacing the usual semantics operator (union, transitions, fixpoint computation, …) by an abstract counterpart. Consider the small example of Program 1.1, this program contains an incrementation function `incr`. The behavior of function `main` depends on the behavior of function `incr`. A non modular analysis will follow the execution of this program, therefore the body of function `incr` will be visited at least twice. However, in a modular analysis, we would start by inferring from the body of function `incr` that: $\forall \mathtt{x}, \mathtt{incr}(\mathtt{x}) = \mathtt{x} + 1$. This fact describes completely the behavior of `incr`, which will therefore not be visited again. When trying to design a modular analysis in the framework of abstract interpretation, one is faced with two challenges: precision and cost. Indeed during a classical analysis of a program, we do not compute an over-approximation of the semantics of all statements appearing in the program, we rather compute an over-approximation of the part of the semantics that is used for the analysis of the complete program. As an example: it is unnecessary to compute the complete semantics of statement `y = x + 1` in Program 1.1, as we will only be interested in the cases where $\mathtt{x} = 0$ or $\mathtt{x} = 1$. Computing the complete semantics of this statement might be more costly than computing only parts of it, however discovering such facts is non trivial when trying to analyze functions independently from one another. Furthermore as we perform static analysis by abstract interpretation, semantics are over-approximated. Analyzing a component of the program independently from the rest may lead to a result that is strictly less precise than what we would have obtained by partitioning the context and computing a semantics for each case. A modular analyzer is provided in Chapter 5, this chapter also provides a more detailed introduction to the problem of modular analysis as well as a presentation of already existing works. The modular analyzer of Chapter 5 is able to handle most of the C language features (including pointers) as we designed it on top of an existing analysis for the C language [Min06a]. It performs a top-down analysis of the program, following the control flow. When function calls are encountered the analyzer will (1) use a pre-discovered summary, hence

```c
typedef struct node
{
  int data;
  struct node* next;
} node;

node* append(node* head, int data)
{
  if (head==NULL) {
    return (create(data, NULL));
  } else {
    node *cursor=head;
    while(cursor->next != NULL)
      cursor=cursor->next;
    node* new_node=create(data,NULL);
    cursor->next=new_node;
    return head;
  }
}
```

Program 1.3: append function

avoiding an analysis of the function body, or (2) infer a context-dependant summary . Input contexts are however stabilized by use of widening before analysis, this ensures that, ultimately, summaries are stable. Finally in order to improve precision and reusability, input framing will be used.

Strings. In order to show that modularity can be obtained even for analyzers targeting properties that are not purely numerical, we decided to enrich our C analyzer so that is was able to reason about mutable `'\0'`-terminated strings. Our analysis handles accesses to strings made via C pointer arithmetic manipulations, as an example consider the string copy from Program 1.2. Strings are used in all C software and are often error prone as they may lead to out-of-bound accesses. This new abstract domain is described in Chapter 4.

Trees. The precision and scalability of our modular static analyzer depends greatly upon the quality of its framing. The frame rule from separation logic [Rey02] enables to lift local reasoning: if a precondition P of a statement is “extended” with some facts, and the added facts do not interfere with the statement, then these facts can be added to the postcondition obtained from P. In order to improve both precision and analysis time we need our analyzer to “frame” its inputs. As an example consider Program 1.3, describing an append function that adds an integer d at the end of a list h. The behavior of append is independent from the content of the data fields of the list. Discovering this fact would allow us to forget the values contained in these fields before the analysis, and allow reusability of the summary independently from the values contained in the list. This can be achieved by providing the analyzer with facts gathered beforehand by a pre-analysis. We decided to use terms of the C language to describe the memory zones that may be used during a function call, for example in the case of the append function we would say that only $\{\mathsf{head}, *(\mathsf{head} + 4), \ldots\}$ are used by the function. We were therefore faced with the problem of analyzing a language manipulating terms containing numerical values. This led to the development of an abstract domain presented in Chapter 7.

Numeric domains. During the development of the tree abstract domain, we needed to be able to manipulate heterogeneous sets of environments. These are sets of environments that are not all defined on the same support. In classical abstract interpretation, concrete environments from the semantics are abstracted using numerical abstract domains. These domains have been widely studied (see Section 2.4 for an overview), and are able to abstract set of environments defined over a given finite set of variables. However when analyzing dynamic languages like python, or when designing high-level predicate abstractions, we need to represent and manipulate heterogeneous sets of environments. Very few results gave an answer to this problem, some of them being more “folk results” that were not well-defined, widely used and limited to non relational numerical domains. For these reasons we defined a general lifting of numerical abstract domains to be able to represent heterogeneous environments. This lifting and several of its improvements are presented in Chapter 6. Note moreover that the ability to represent heterogeneous environments can also be used, during a modular analysis, to merge input contexts of a function, not necessarily defined on the same variable set.

Mopsa. This thesis took place in the Mopsa project. The goal of this project is to design and develop a modular framework for the development of static analyzers defined by abstract interpretation. All implementations from this thesis were made in the Mopsa static analyzer. This served the dual purpose of: (1) easing the development of the new abstract domains as already existing features could be reused; (2) testing and improving the expressiveness of the Mopsa framework.

1.3 Outline of the thesis

The remainder of the thesis is organized as follow:

- Chapter 2 provides some of the theoretical prerequisites of this thesis.
- Chapter 3 details some of the features of the Mopsa static analyzer. This analyzer is an ongoing project being developed by several persons (including myself), this chapter has therefore a dual purpose: (1) it presents some the work performed on this analyzer during this thesis; (2) it shows some of the features of the analyzer in which all subsequent abstract domains and algorithms of this thesis were implemented.
- Chapter 4 presents an abstract domain targeting the discovery of out-of-bounds accesses in C `'\0'`-terminated strings. The definition of this abstraction requires to introduce the cell abstraction [Min06a], used to represent C memory manipulation in our analyzer.
- Chapter 5 presents a modular analyzer. We start by showing how this modular analyzer handles numerical programs. Moreover, in order to emphasize that our modular analysis can be used for abstract domains that are not purely numerical, we apply the technique on a abstract domain able to handle the full C language.
- Chapter 6 provides numerical abstractions that enable the representation of set of numerical maps that do not necessarily share the same support. These abstractions were first devised because they were needed in: the Mopsa framework, the definition of higher level abstractions as the one of Chapter 7.
- Chapter 7 presents an abstraction for sets of trees with leaves that can be labeled with numerical values. The goal of this abstraction is to provide a pre-analysis for the C language. This pre-analysis would be used in combination with results from Chapter 5.
- Chapter 8 concludes the thesis, and puts forth some future works.
- Finally, note that results we deemed proof-worthy are annotated as such in the body of the thesis, proofs can then be found in Appendix A.

Each chapter (except Chapter 2) features: (1) a motivating introduction, (2) a presentation of related works, (3) some implementation considerations and (4) a conclusion

Chapter 2

Background – Abstract interpretation

In this chapter we provide the theoretical background of this thesis. We present here the common part of the prerequisites of all subsequent chapters. Every chapter will then provide backgrounds of their own (such sections are marked with “Background” so as to differentiate contributions from state of the art results). As an example, results from automata theory are used in this thesis but will only be introduced in the Background section of Chapter 7.

This thesis will make extensive use of the abstract interpretation framework [CC77b]. For this reason this chapter is devoted to a presentation of some of the results of abstract interpretation used in the subsequent chapters.

The mathematical semantics of a programming language (no matter its definition pattern: denotational, equational, ...) is usually defined ([Cou97]) using the following:

- a set of all “states” $\mathcal{D}$;
- atomic transformers operating on $\wp(\mathcal{D})$: $\wp(\mathcal{D}) \to \wp(\mathcal{D})$;
- union;
- least fixed point.

Indeed unions are used for merging points (e.g. the end of an `if` statement) and the least fixed point are used to define the semantics of looping features (e.g. recursivity, `while`, `goto`).

As soon as : (1) the atomic transformers provided by the semantics are expressive enough (tests and basic numerical operations), (2) and the language features a looping mechanism, one can prove that knowing whether the semantics of a given program satisfies some non trivial property or not is non decidable. This undecidability may come from the following:

1. the sets of states manipulated might not be machine representable;
2. some of the atomic transformers might not be computable;
3. the union might not be computable;
4. the least fixed point might not be reached in finitely many steps.

As the precise semantics might not be computable, abstract interpretation will focus on the computation of an over-approximation of this semantics. Indeed as mentioned in the introductory Chapter 1, our goal is to prove that the semantics is included in some property. In order to obtain this, it is sufficient to prove that some over-approximation of the semantics is included in the property. Instead of operating on the set of states defined by the concrete semantics (called concrete states), we move the computations to a “simpler” world (called abstract states). The computation of an over-approximation of the semantics will then be made by replacing each operator used in the definition of the concrete semantics (e.g. union, atomic transformers, least fixed point) with an abstract counterpart. Elements from the abstract world will be given a meaning in the concrete world via the use of a concretization function. An abstract element is then called a sound over-approximation of some concrete element if its concretization covers the concrete element. We mentioned that the computation of this over-approximating abstract counterpart will be made by following the mathematical definition of the concrete semantics, therefore abstract interpretation provides an answer to each of the four sources of indecidability presented therebefore.

- Point 1 is answered by defining a machine representable abstract world;
- Point 2 and Point 3 are answered by defining over-approximating abstract transformers operating on the abstract world. These operators are over-approximating in the sense that their image covers at least the image of their concrete counterpart;
- Point 4 is answered by defining an binary operator $\triangledown$ that accelerates convergence to an over-approximation of the least fixed point.

The first section of this chapter is dedicated to the presentation of some of the mathematical notations used in this thesis. As we have seen the soundness notion is defined via a comparison operator. Section 2.2 is dedicated to a presentation of results from order and lattice theory including a definition and some properties of Galois connections, at the heart of the concrete/abstract pair. Section 2.3 is then dedicated to a case study, where we show how an abstract interpreter is built and how its soundness is ensured by results from Section 2.2. Finally as numerical domains are widely used in the following chapters, we dedicate Section 2.4 to a catalogue of the most used of them.

2.1 Notations

Substitution. Given two sets A and B, an element $a \in A$, an element $b \in B$, a function $f \in A \to B$, we denote as $f[a \mapsto b]$ the function from A to B associating b to a and $f(x)$ to x otherwise.

λ **expressions.** Given a set A, we will use the λ notations: $\lambda x \in A, f$ to denote the mathematical function defined on A and associating f to x.

Functions defined as extension. Given a set $\{x_1, \ldots, x_n\}$, the notation $(x_1 \mapsto a_1, \ldots, x_n \mapsto a_n)$ is used to denote the function defined on $\{x_1, \ldots, x_n\}$ and associating a_i to each of the x_i.

Functions projection and extension. Given two sets A and B, a function $f \in A \to B$, a subset A' of A we denote as $f_{|A'}$, the restriction of f to A'. Moreover given two functions $f \in A \to B$ and $g \in C \to B$ such that $A \cap C = \emptyset$, we denote as $f \uplus g$ the function: $\lambda x \in A \cup C. \begin{cases} f(x) & \text{if } x \in A \\ g(x) & \text{otherwise} \end{cases}$. Both notations are extended to set operations: Given a set of functions F, $F_{|A'} = \{f_{|A'} \mid f \in F\}$ and given another set of functions G, $F \uplus G = \{f \uplus g \mid f \in F, g \in G\}$.

Partial functions. Given two sets A and B, $A \nrightarrow B$ denotes the set of partial maps from A to B. We denote as $\mathbf{def}(f)$ the definition set of f.

Set of subsets. Given a set S, $\wp(S)$ denotes the set of all subsets of S and $\wp_f(S)$ denotes the set of all finite subsets of S.

Equivalence classes. When S is a set, $\sim$ is an equivalence relation on elements from S and $x \in S$, we denote as $[x]^\sim$ the equivalence class of x under $\sim$.

2.2 Order theory

2.2.1 Order and Galois connections

Abstract interpretation is a theory of sound approximations of semantics. The soundness notion will be expressed via an order relation (e.g. the inclusion). For this reason, we start by recalling some results on order relations.

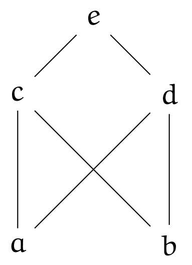


Figure 2.1: Hasse diagram of Example 2.1

Definition 2.1 (Partial order)**.** Given a set S, a *partial order* $\preccurlyeq$ on S, is a binary relation (a subset of $S \times S$) being:

- reflexive: $\forall x \in S, x \preccurlyeq x$;
- transitive: $\forall x, y, z \in S, x \preccurlyeq y \Rightarrow y \preccurlyeq z \Rightarrow x \preccurlyeq z$;
- anti-symmetric: $\forall x, y \in S, x \preccurlyeq y \Rightarrow y \preccurlyeq x \Rightarrow x = y$

Definition 2.2 (Poset)**.** When a set S, is equipped with a partial order $\preccurlyeq$, we say that $(S, \preccurlyeq)$ is a *poset*.

Definition 2.3 (Monotonicity)**.** Given two posets $(S_1, \preccurlyeq_1)$ and $(S_2, \preccurlyeq_2)$, and a function $f \in S_1 \to S_2$, f is said to be *monotonous* whenever $\forall x, y \in S_1, x \preccurlyeq_1 y \Rightarrow f(x) \preccurlyeq_2 f(y)$.

Definition 2.4 (lower-bound, upper-bound, lub, glb)**.** Given a poset $(S, \preccurlyeq)$ and a subset $X \subseteq S$, we say that m (resp. M) is a *lower* (resp. *upper*) *bound* of X, whenever $\forall x \in X, m \preccurlyeq x$ (resp. $\forall x \in X, x \preccurlyeq M$). We denote as $X^{\downarrow}$ (resp. $X^{\uparrow}$) the subset of S of all lower bounds (resp. upper bounds) of X. Let m (resp. M) be a lower (resp. upper) bound of X, we say that m (resp. M) is the *greatest lower bound* (glb) (resp. *least upper bound* (lub)), whenever $\forall m' \in X^{\downarrow}, m' \preccurlyeq m$ (resp. $\forall M' \in X^{\uparrow}, M \preccurlyeq M'$). When such a lub (resp. glb) exists it is unique and will be denoted $\curlywedge X$ (resp. $\curlyvee X$).

A commonly used visual representation of posets (and this thesis will not fail to the rule) are Hasse diagrams, which we now succinctly present.

Definition 2.5 (Hasse diagrams)**.** Given a poset $(S, \preccurlyeq)$, we define the following directed graph (V, E), where:

- $V = S$
- $E = \{(x, y) \mid x \preccurlyeq y \wedge x \neq y \wedge \forall z \in S, x \preccurlyeq z \preccurlyeq y \Rightarrow z = x \vee z = y\}$

Basically we have a transition between an element x and y if $x \preccurlyeq y$, but we remove transitions that can be inferred from the reflexive, and transitive, property of the order relation.

Remark 2.1. By the transitive and anti-symmetric property of the order relation we have that the graph is actually acyclic. Edges of a Hasse diagram are therefore represented visually by an edge going up from the smallest element of the two, to the biggest one.

Example 2.1 (Non existence of lub)**.** Consider the following set $S = \{a, b, c, d, e\}$ ordered by the relation $\preccurlyeq = \{(a, a), (b, b), (c, c), (d, d), (e, e), (a, c), (a, d), (b, c), (b, d), (a, e), (b, e), (c, e), (d, e)\}$, represented by the Hasse diagram of Figure 2.1. We can see that the set $\{c, d\}$ does not admit a greatest lower bound, however it admits a least upper bound $\curlyvee\{c, d\} = e$

Definition 2.6 (Chains)**.** Given a poset $(S, \preccurlyeq)$, a *chain* is a subset C of S such that: $\forall x, y \in C, x \preccurlyeq y \vee y \preccurlyeq x$. C is completely ordered.

Definition 2.7 (CPO)**.** A poset $(S, \preccurlyeq)$ is called a CPO if every chain of the poset admits a lub, it is then denoted $(S, \preccurlyeq, \curlyvee, \bot)$, where $\bot = \curlyvee \emptyset$.

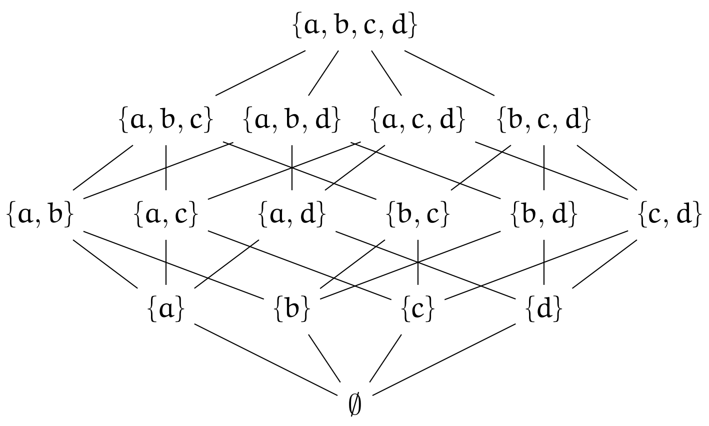


Figure 2.2: Hasse diagram of Example 2.1

Definition 2.8 (Lattice)**.** A poset $(S, \preccurlyeq)$ is called a *lattice* when every subset of size 2 of S admits a lub and a glb, those are then denoted $\curlyvee$ (for the lub) and $\curlywedge$ (for the glb). We then say that $(S, \preccurlyeq, \curlyvee, \curlywedge)$ is a lattice.

Definition 2.9 (Complete lattice)**.** A lattice $(S, \preccurlyeq, \curlyvee, \curlywedge)$ is said to be a *complete lattice* whenever every subset of X admits a glb. This implies that every subset of X admits a lub (and conversely). We then denote $\bot = \curlyvee \emptyset$ and $\top = \curlywedge \emptyset$, and say that $(S, \preccurlyeq, \curlyvee, \curlywedge, \bot, \top)$ is a complete lattice.

Example 2.2 (Powerset lattice)**.** For any given set X the powerset lattice $(\wp(X), \cup, \cap, \emptyset, X)$ is a complete lattice. Figure 2.1 provides the Hasse diagram of the powerset lattice of the set $\{a, b, c, d\}$.

Definition 2.10 (Lifted lattice)**.** Given a (complete) lattice $(S, \preccurlyeq, \curlyvee, \curlywedge, \bot, \top)$ and a set X we can derive a new complete lattice: $((X \to (S \setminus \{\bot\})) \cup \overline{\bot}, \overline{\preccurlyeq}, \overline{\curlyvee}, \overline{\curlywedge}, \overline{\bot}, \overline{\top})$ where:

$$\begin{aligned}
\overline{\preccurlyeq} &\triangleq \{(f, f') \mid \forall x \in X, f(x) \preccurlyeq f'(x)\} \\
f \overline{\curlyvee} f &\triangleq \lambda x. f(x) \curlyvee f'(x) \\
f \overline{\curlywedge} f &\triangleq \lambda x. f(x) \curlywedge f'(x) \\
\overline{\top} &\triangleq \lambda x. \top
\end{aligned}$$

and $\overline{\bot}$ is a new element.

Remark 2.2. This last definition provides a way to lift a “value” lattice to an “environment” lattice by setting X to be a set of variables.

We now introduce vocabulary specific to abstract interpretation. Recall that instead of working in the concrete world (C), our goal is to translate the computations to a simpler abstract world (A). In order to give a meaning to elements of the abstract world as abstractions of elements from the concrete world, we define a concretization function, which is a function from the abstract world to the concrete one. We can then rely on the comparison operator provided by the concrete world to compare a concrete element with a concretized abstract element. As mentioned earlier, the notion of soundness is conveyed by the order relation of the concrete world, we will therefore say that an abstract element a is an sound abstraction of a concrete one c if the concretization of a is greater than c.

Definition 2.11 (Concretization, soundness, exactness)**.** Given two posets $(C, \preccurlyeq)$ and $(A, \sqsubseteq)$, a monotonic function $\gamma \in A \to C$ is called a *concretization* function. Given $c \in C$ and $a \in A$, we then say that:

- a is a *sound abstraction* of c whenever $c \preccurlyeq \gamma(a)$;
- a is an *exact abstraction* of c whenever $c = \gamma(a)$.

Even though the concretization is sufficient to reason about soundness (e.g. prove that an analyzer is sound in the sense that it considered the complete semantics of a program), we now define a stronger framework: Galois connections. The notion of Galois connection is used to link an abstract world and a concrete one, in addition to a concretization function, this provides an abstraction function. This function maps elements from the concrete world to elements from the abstract world. Moreover the equivalence from the next definition ensures that in a Galois connection the abstraction function provides the best possible sound abstraction (again for the order relation of the abstract world).

Definition 2.12 (Galois connections)**.** Given two posets $(C, \preccurlyeq)$ and $(A, \sqsubseteq)$, we say that the pair $(\alpha \in C \to A, \gamma \in A \to C)$ is a *Galois connection* when:

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

We then write $(C, \preccurlyeq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$

Proposition 2.1 (Properties on Galois connections [CC77b])**.** *Given a Galois connection* $(C, \preccurlyeq\ ) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ *we have:*

- γ *and* α *are monotonic*
- $\forall c \in C, c \preccurlyeq \gamma \circ \alpha(c)$
- $\forall a \in A, \alpha \circ \gamma(a) \sqsubseteq a$
- $\gamma \circ \alpha \circ \gamma = \gamma$ *and* $\alpha \circ \gamma \circ \alpha = \alpha$
- $\forall c \in C, \{a \mid c \preccurlyeq \gamma(a)\}$ *admits a glb and* $\alpha(c) = \bigsqcap\{a \mid c \preccurlyeq \gamma(a)\}$
- $\forall a \in A, \{c \mid \alpha(c) \sqsubseteq a\}$ *admits a lub and* $\gamma(a) = \curlyvee\{c \mid \alpha(c) \sqsubseteq a\}$
- *Given two Galois connections* $(S_1, \preccurlyeq_1) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} (S_2, \preccurlyeq_2)$ *and* $(S_2, \preccurlyeq_2) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} (S_3, \preccurlyeq_3)$ *we have the following Galois connection:* $(S_1, \preccurlyeq_1) \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} (S_3, \preccurlyeq_3)$
- *Given a Galois connection* $(C, \preccurlyeq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ *between two lattices* $(C, \preccurlyeq, \curlyvee, \curlywedge, \bot_c, \top_c)$ *and* $(A, \sqsubseteq, \sqcup, \sqcap, \bot_a, \top_a)$*, and a set* $\mathcal{V}$*, we have a Galois connection between the two lifted posets of Definition 2.10:* $((X \to C \setminus \{\bot_c\}) \cup \overline{\bot_c}, \overline{\preccurlyeq}) \underset{\overline{\alpha}}{\overset{\overline{\gamma}}{\leftrightarrows}} ((X \to A \setminus \{\bot_a\}) \cup \overline{\bot_a}, \overline{\sqsubseteq})$ *where*

$$\overline{\gamma}(f) \triangleq \begin{cases} \overline{\bot_c} & \textit{if } f = \overline{\bot_a} \\ \lambda s \in X.\gamma(f(s)) & \textit{otherwise} \end{cases}$$

$$\overline{\alpha}(f) \triangleq \begin{cases} \overline{\bot_a} & \textit{if } f = \overline{\bot_c} \\ \lambda s \in X.\alpha(f(s)) & \textit{otherwise} \end{cases}$$

2.2.2 An example of Galois connection: intervals [CC76]

We recall here the definition of the Galois connection between the lattice $(\wp(\mathbb{Z}), \subseteq, \cup, \cap, \emptyset, \mathbb{Z})$ and the interval lattice (see [CC76]). We define the set of *intervals* to be $\mathcal{I} \triangleq \{[a; b] \mid a \in \mathbb{Z} \cup \{-\infty\} \wedge b \in \mathbb{Z} \cup \{\infty\} \wedge a \leqslant b\} \cup \{\bot\}$. On $\mathcal{I}$ we define the concretization function γ such that

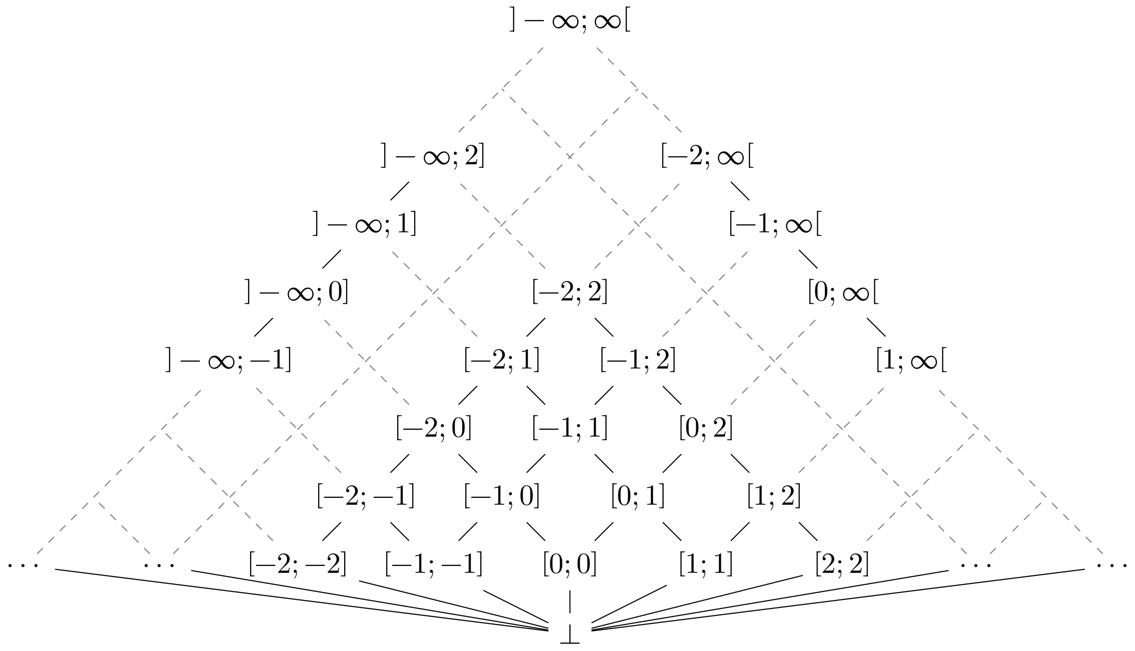


Figure 2.3: Hasse diagram of the interval lattice

$\gamma([a; b]) = \{x \in \mathbb{Z} \mid a \leqslant x \leqslant b\}$ and $\gamma(\bot) = \emptyset$. $\mathcal{I}$ is enriched with the following operators:

$$
\begin{aligned}
\sqsubseteq &\triangleq \{([a; b], [c; d]) \mid c \leqslant a \wedge b \leqslant d\} \\
[a; b] \sqcup [c; d] &\triangleq [\min(a, c); \max(b, d)] \\
[a; b] \sqcap [c; d] &\triangleq \begin{cases} [\max(a, c); \min(b, d)] & \text{if } \max(a, c) \leqslant \min(b, d) \\ \bot & \text{otherwise} \end{cases} \\
\top &\triangleq [-\infty; \infty]
\end{aligned}
$$

$(\mathcal{I}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ is a complete lattice, Figure 2.3 represents its Hasse diagram. Moreover let us consider the function $\alpha \in \wp(\mathbb{Z}) \to \mathcal{I}$ defined by:

$$
\alpha(S) = \begin{cases} [\min(S); \max(S)] & \text{if } S \neq \emptyset \\ \bot & \text{otherwise} \end{cases}
$$

We then have the following Galois connection $(\wp(\mathbb{Z}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{I}, \sqsubseteq)$. Note that we went from the world $\wp(\mathbb{Z})$, the elements of which are not all machine representable, to the interval world $\mathcal{I}$, the elements of which are machine representable. Obviously information was lost, consider for example the set $2\mathbb{Z}$ which can not be precisely represented with an interval. Its most precise abstraction is $]-\infty; \infty[$, representing $\mathbb{Z} \supsetneq 2\mathbb{Z}$.

2.2.3 Operators, fixpoint theorems

The semantics of a transition system is defined using not only set-theoretic operations, but also operators, representing transition steps in the system. An operator is merely a function between two posets, we start here by providing some vocabulary on such functions.

Definition 2.13 (function properties).

- Continuity: Given two CPO $(S_1, \preccurlyeq_1)$ and $(S_2, \preccurlyeq_2)$ (the lub of chains will be denoted resp. $\curlyvee_1$ and $\curlyvee_2$), a function $f \in S_1 \to S_2$ is said to be *continuous* whenever for every chain C_1 of S_1 we have: $\{f(x) \mid x \in C_1\}$ is a chain of S_2 and $f(\curlyvee_1 C_1) = \curlyvee_2\{f(x) \mid x \in C_1\}$.

- Join-morphism: a function f between a lattice $(S_1, \preccurlyeq_1, \curlyvee_1, \dots)$ and a lattice $(S_2, \preccurlyeq_2, \curlyvee_2, \dots)$ is said to be a *join-morphism* when: $\forall x, y \in A_1, f(x \curlyvee_1 y) = f(x) \curlyvee_2 f(y)$. When A_1 and A_2 are complete lattices and the property can be extended to arbitrary sets, we call f a complete join morphism. A join morphism is monotonous, a complete join morphism is continuous.

We have seen how a concretization function is used to define a notion of sound abstraction of a concrete element. We now define a notion of sound operators. Informally a function f^A operating on the abstract world is said to be a sound abstraction of its concrete counterpart f whenever the image of every abstract element is a sound abstraction of the image of the concretization of this element. This property is stable by composition, which ensures that an analyzer built as the composition of sound operators will itself be sound. Similarly a notion of exactness is defined.

Definition 2.14 (Sound and exact abstraction of operators). Given a monotonic function γ from the poset $(A, \sqsubseteq)$ to the poset $(C, \preccurlyeq)$ and an operator $f \in C \to C$. We say that f^A is a *sound abstraction of operator* f whenever

$$\forall x \in A, f(\gamma(x)) \preccurlyeq \gamma(f^A(x))$$

we say that is an *exact abstraction of operator* f whenever

$$\forall x \in A, f(\gamma(x)) = \gamma(f^A(x))$$

Remark 2.3. When we are given a Galois connection between the concrete and abstract world: $(C, \preccurlyeq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, $f^A \triangleq \alpha \circ f \circ \gamma$ is a sound abstraction of f. Moreover it is the best possible abstraction of f in the sense that: if g is another sound abstraction of f then $\forall a \in A, f^A(a) \sqsubseteq g(a)$.

The definition of a semantics makes use of fixpoints (e.g. for the definition of the semantics of a `while` statement), more precisely it uses the notion of least fixed point, which is the smallest of all fixpoints. We provide here two fixpoint theorems that will ensure that the semantics is well-defined (there exists a least fixed point). Moreover as the definition of our abstract semantics will copy that of the original semantics we provide transfer theorems ensuring a fixpoint computed in the abstract world over-approximates the least fixed point from the original semantics.

Definition 2.15 (Fixpoints). Given a poset $(S, \preccurlyeq)$ and a function $f \in S \to S$, we define:

- a *fixpoint* of f is an element $x \in S$ such that $f(x) = x$, we denote as $\mathbf{fp}(f)$ the set of all such fixpoints.
- $\mathbf{lfp}(f)$ denotes (when it exists) the least fixed point of f.

Theorem 2.1 (Tarski's fixpoint theorem [Tar55]). *If* $(S, \preccurlyeq, \curlyvee, \curlywedge, \bot, \top)$ *is a non empty complete lattice and* f *is a monotonic function, then* $\mathbf{fp}(f)$ *is a non empty complete lattice,* $\mathbf{lfp}(f)$ *exists and is such that* $\mathbf{lfp}(f) = \bigcurlywedge\{x \in S \mid f(x) \preccurlyeq x\}$

Theorem 2.2 (Kleene's fixpoint theorem [CC77b]). *Given a CPO* $(S, \preccurlyeq, \bigcurlyvee, \bot)$*, a continuous function* $f \in S \to S$ *we have* $\mathbf{lfp}(f)$ *exists,* $\{f^n(\bot) \mid n \in \mathbb{N}\}$ *is a chain, and* $\mathbf{lfp}(f) = \bigcurlyvee\{f^n(\bot) \mid n \in \mathbb{N}\}$*.*

Theorem 2.3 (Tarski's fixpoint approximation [Cou97]). *Given a complete lattice* $(C, \preccurlyeq, \curlyvee, \curlywedge, \bot_c, \top_c)$*, a poset* $(A, \sqsubseteq)$*, a concretization function* γ *from* A *to* C*,* f *a monotonic operator on* C*,* g *an operator on* A *that is a sound approximation of* f *with respect to* γ*, then* $\forall a \in A, g(a) \sqsubseteq a \Rightarrow \mathbf{lfp}(f) \preccurlyeq \gamma(a)$*.*

Theorem 2.4 (Kleene's fixpoint approximation [Cou97]). *Given a CPO* $(C, \preccurlyeq, \bigcurlyvee, \bot_c)$*, a poset with minimal element* $(A, \sqsubseteq, \bot_a)$*, a concretization function* γ *from* A *to* C*,* f *a continuous operator on* C*,* g *an operator on* A *that is a sound approximation of* f *with respect to* γ*, if the sequence* $\{g^i(\bot_a) \mid i \in \mathbb{N}\}$ *has a least upper bound* $l = \bigsqcup\{g^i(\bot_a) \mid i \in \mathbb{N}\}$ *then* $\mathbf{lfp}(f) \preccurlyeq \gamma(l)$*.*

2.2.4 Widening

In the introductory remarks to this chapter we underlined four difficulties encountered while trying to compute the concrete semantics. To this point we provided an answer to the first three by translating the problem to an abstract world. However there remains the problem that the least fixed point used in the concrete may not be computed in finitely many steps in the abstract. As we will see in the next section, following Kleene's fixpoint theorem, we compute over-approximations of fixpoints of functions f by considering the iterates $({f^{A}}^{i}(\bot))_{i\in\mathbb{N}}$, where f^A is a sound approximation of operator f. The sequence $({f^{A}}^{i}(\bot))_{i\in\mathbb{N}}$ will often be increasing. Whenever the abstract world has finite height (meaning that there exists no infinite sequence $(y_i)_{i\in\mathbb{N}}$ such that $\forall i, y_i \sqsubseteq y_{i+1}$ and $y_i \neq y_{i+1}$) the iterates $({f^{A}}^{i}(\bot))_{i\in\mathbb{N}}$ converges. However in all abstractions considered in this thesis, the abstract lattice will have infinite height, therefore the sequence $({f^{A}}^{i}(\bot))_{i\in\mathbb{N}}$ is not ensured to converge. For this reason we define a widening operator, the goal of which is to stabilize sequences.

Definition 2.16 (Widening operator)**.** Given a poset $(A, \sqsubseteq)$ we say that a binary operator $\triangledown \in A \times A \to A$ is a *widening operator* if:

1. $\forall x, y \in A, x \sqsubseteq x\triangledown y \wedge y \sqsubseteq x\triangledown y$
2. for all sequences $(x_i)_{i\in\mathbb{N}} \in A^{\mathbb{N}}$ the sequence $(y_i)_{i\in\mathbb{N}} \in A^{\mathbb{N}}$ defined by $(y_0 = x_0) \wedge (y_{i+1} = y_i \triangledown x_{i+1})$ satisfies $\exists N \in \mathbb{N}, \forall k \geqslant N, y_k = y_N$

Theorem 2.5 (Widening ensures convergence [CC77b])**.** *Given a complete lattice C, a poset A with minimal element $\bot_A$, a concretization function γ from A to C, a monotonic operator f on C, an operator g on A that is a sound approximation of f with respect to γ, then the sequence $(y_i)_{i\in\mathbb{N}}$ defined by:*

$$\begin{aligned} y_0 &= \bot_A \\ y_{i+1} &= y_i \triangledown g(y_i) \end{aligned}$$

*is such that $\exists N \in \mathbb{N}, \forall k \geqslant N, y_k = y_N$. Moreover for such a N, we have **lfp**$(f) \preccurlyeq \gamma(y_N)$.*

2.3 Abstract Interpretation

2.3.1 By example

As abstract interpretation is a framework easing the development of static analyzers by providing sufficient conditions to ensure the soundness of the analysis, we show here an example of static analyzer defined by abstract interpretation. Let us consider a modified version of register machines as toy language for this section. The reason we chose to present register machine as toy language is the fact that numerical operations are limited to incrementation and decrementation. Therefore we do not have to deal with the definition of the evaluation of expressions. This language will be denoted as RM in the remainder of this section. The different points presented in the following are the classical steps followed by the definition of an abstract interpreter.

Syntax of the language. We start by providing the syntax of the language. We assume given a finite non empty set of variables (the registers) denoted $\mathcal{R}$. We denote as *stmt* the set of all statements from our language, defined by the following grammar.

$$
\begin{array}{lll}
\mathit{stmt} \triangleq & |\ \texttt{while (r!=0) \{}\mathit{stmt}\texttt{\}} & \mathtt{r} \in \mathcal{R} \\
& |\ \texttt{if (r!=0) \{}\mathit{stmt}\texttt{\} else \{}\mathit{stmt}\texttt{\}} & \mathtt{r} \in \mathcal{R} \\
& |\ \texttt{skip} & \\
& |\ \mathit{stmt}\texttt{;}\mathit{stmt} & \\
& |\ \texttt{r++} & \mathtt{r} \in \mathcal{R} \\
& |\ \texttt{r--} & \mathtt{r} \in \mathcal{R}
\end{array}
$$

Memory state. The semantics of the language will operate on sets of states which are maps assigning an integer value to each of the registers. We denote as $\mathcal{D}$ the set of all such sets of maps: $\mathcal{D} \triangleq \wp(\mathcal{R} \to \mathbb{Z})$.

Semantics of atomic statements. As mentioned in the introductory remarks to this section, the definition of the semantics requires the definition of the transfer functions for atomic statements. Given an atomic statement `stmt` we denote as $\mathbb{S}[\![\texttt{stmt}]\!] \in \mathcal{D} \to \mathcal{D}$ the transfer function associated to statement `stmt`.

$$
\begin{aligned}
\mathbb{S}[\![\texttt{r==0}]\!] &\triangleq \lambda R \in \mathcal{D}.\{\rho \mid \rho \in R \wedge \rho(\mathtt{r}) = 0\} \\
\mathbb{S}[\![\texttt{r!=0}]\!] &\triangleq \lambda R \in \mathcal{D}.\{\rho \mid \rho \in R \wedge \rho(\mathtt{r}) \neq 0\} \\
\mathbb{S}[\![\texttt{r++}]\!] &\triangleq \lambda R \in \mathcal{D}.\{\rho[\mathtt{r} \mapsto \rho(\mathtt{r}) + 1] \mid \rho \in R\} \\
\mathbb{S}[\![\texttt{r--}]\!] &\triangleq \lambda R \in \mathcal{D}.\{\rho[\mathtt{r} \mapsto \rho(\mathtt{r}) - 1] \mid \rho \in R\} \\
\mathbb{S}[\![\texttt{skip}]\!] &\triangleq \lambda R \in \mathcal{D}.R
\end{aligned}
$$

Semantics of compound statements. Now that we have provided a mathematical definition of the semantics of atomic statements, we define by induction on the syntax, the semantics of compound statements. As mentioned before, these definitions rely on the use of the mathematical set union and smallest fixed points.

$$
\begin{aligned}
&\mathbb{S}[\![\texttt{while (r!=0) \{stmt\}}]\!] \triangleq \lambda R \in \mathcal{D}.\mathbb{S}[\![\texttt{r==0}]\!](\mathbf{lfp}(\lambda S \in \mathcal{D}.R \cup \mathbb{S}[\![\texttt{stmt}]\!] \circ \mathbb{S}[\![\texttt{r!=0}]\!](S))) \\
&\mathbb{S}[\![\texttt{if (r!=0) \{stmt}_t\texttt{\} else \{stmt}_e\texttt{\}}]\!] \triangleq \lambda R \in \mathcal{D}.\mathbb{S}[\![\texttt{stmt}_t]\!] \circ \mathbb{S}[\![\texttt{r!=0}]\!](R) \cup \mathbb{S}[\![\texttt{stmt}_e]\!] \circ \mathbb{S}[\![\texttt{r==0}]\!](R) \\
&\mathbb{S}[\![\texttt{stmt}_1\texttt{;stmt}_2]\!] \triangleq \mathbb{S}[\![\texttt{stmt}_2]\!] \circ \mathbb{S}[\![\texttt{stmt}_1]\!]
\end{aligned}
$$

We mentioned that any powerset is a complete lattice. In particular here $\mathcal{D}$ is a complete lattice, and we can show that for every statement `stmt` $\mathbb{S}[\![\texttt{stmt}]\!]$ is a complete join-morphism. It follows that for every $R \in \mathcal{D}$ and for every $\texttt{stmt} \in \mathit{stmt}$ the function $F \triangleq \lambda S \in \mathcal{D}.R \cup \mathbb{S}[\![\texttt{stmt}]\!] \circ \mathbb{S}[\![\texttt{r!=0}]\!](S) \in \mathcal{D} \to \mathcal{D}$ is a complete join-morphism. By application of both Theorem 2.1 and Theorem 2.2 we get that F admits a least fixed point. Therefore our semantics is well-defined.

Abstract domain. The first step in building a static analyzer by abstract interpretation is the definition of an *abstract domain*. An abstract domain is a structure providing all the operators required for the definition of an abstract semantics. We recall that the abstract semantics is defined by following the definition of the concrete semantics and replacing some of the uncomputable operators by computable, over-approximating ones. Moreover as the least fixed point might not be reached (via Kleene's theorem) in a finite number of iterations, we ask abstract domains to provide an operator accelerating convergence, denoted $\triangledown$. An abstract domain therefore provides:

- A lattice $(\mathcal{D}^\sharp, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$;
- A Galois connection with the concrete domain $(\mathcal{D}, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{D}^\sharp, \sqsubseteq)$;
- Abstract transformers for the atomic statements of the language, denoted as $\mathbb{S}^\sharp[\![\mathtt{stmt}]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$: $\mathbb{S}^\sharp[\![\mathtt{r==0}]\!]$, $\mathbb{S}^\sharp[\![\mathtt{r!=0}]\!]$, $\mathbb{S}^\sharp[\![\mathtt{r++}]\!]$, $\mathbb{S}^\sharp[\![\mathtt{r--}]\!]$;
- A widening operator $\triangledown \in \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$.

Moreover in order to ensure soundness and computability of the overall analysis, produced by the composition of such operators, we require the following to hold:

- $\sqcup$ is a sound over-approximation of the union operator $\cup$ (we extend Definition 2.14):

$$\forall R_1^\sharp, R_2^\sharp \in \mathcal{D}^\sharp, \gamma(R_1^\sharp) \cup \gamma(R_2^\sharp) \subseteq \gamma(R_1^\sharp \sqcup R_2^\sharp)$$

- For all atomic statements $\mathtt{stmt}$, $\mathbb{S}^\sharp[\![\mathtt{stmt}]\!]$ is a sound abstraction of operators $\mathbb{S}[\![\mathtt{stmt}]\!]$, in the sense of Definition 2.14;
- Elements from $\mathcal{D}^\sharp$ are machine representable;
- $\triangledown$, $\sqcup$, and $\mathbb{S}^\sharp[\![\mathtt{stmt}]\!]$ for any $\mathtt{stmt}$ are computable operators.

Example continued: the interval domain. Let us now define an abstract domain for the RM language. Our first abstraction will be to lose the relationality of the concrete world $\mathcal{D} = \wp(\mathcal{R} \to \mathbb{Z})$. We have shown that $(\wp(\mathbb{Z}), \subseteq, \cup, \cap, \emptyset, \mathbb{Z})$ is a complete lattice. We lift this lattice using Definition 2.10, to a lattice $L_1 = (\mathcal{R} \to \wp(\mathbb{Z}) \setminus \{\emptyset\}) \cup \{\overline{\bot}\}, \overline{\subseteq}, \overline{\cup}, \overline{\cap}, \overline{\bot}, \overline{\top})$. We now define a Galois connection between the concrete world and this lattice:

$$\wp(\mathcal{R} \to \mathbb{Z}) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} ((\mathcal{R} \to \wp(\mathbb{Z}) \setminus \{\emptyset\}) \cup \{\overline{\bot}\}, \overline{\subseteq})$$

$$\gamma_1(m) = \begin{cases} \emptyset & \text{if } m = \overline{\bot} \\ \{\rho \in \mathcal{R} \to \mathbb{Z} \mid \forall r \in \mathcal{R}, \rho(r) \in m(r)\} & \text{otherwise} \end{cases}$$

$$\alpha_1(R) = \begin{cases} \overline{\bot} & \text{if } R = \emptyset \\ \lambda r \in \mathcal{R}.\{\rho(r) \mid \rho \in R\} & \text{otherwise} \end{cases}$$

Note that this abstraction looses relations between values in registers. As an example consider the set of environments $R = \{(r_1 \mapsto 1, r_2 \mapsto 0), (r_1 \mapsto 0, r_2 \mapsto 1)\}$, we have $\alpha(R) = (r_1 \mapsto \{0,1\}, r_2 \mapsto \{0,1\})$ and therefore $\gamma \circ \alpha(R) = \{(r_1 \mapsto 1, r_2 \mapsto 0), (r_1 \mapsto 0, r_2 \mapsto 1), (r_1 \mapsto 0, r_2 \mapsto 0), (r_1 \mapsto 1, r_2 \mapsto 1)\}$.

Moreover we have shown in Section 2.2.2 a Galois connection between $\wp(\mathbb{Z})$ and $\mathcal{I}$ the set of intervals of $\mathbb{Z}$. By lifting this Galois connection using Proposition 2.1 and Definition 2.10 we get a Galois connection between the lattice L_1 and the lattice $((\mathcal{R} \to (\mathcal{I} \setminus \{\bot\}) \cup \{\overline{\bot}\}, \overline{\sqsubseteq}, \overline{\sqcup}, \overline{\sqcap}, \overline{\top}, \overline{\bot})$, we denote $\mathcal{D}^\sharp \triangleq (\mathcal{R} \to (\mathcal{I} \setminus \{\bot\}) \cup \{\overline{\bot}\}$

$$((\mathcal{R} \to \wp(\mathbb{Z}) \setminus \{\emptyset\}) \cup \{\overline{\bot}\}, \overline{\subseteq}) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} (\mathcal{D}^\sharp, \overline{\sqsubseteq})$$

By the composability result provided in Proposition 2.1 we get a Galois connection between our concrete world $\mathcal{D}$ and our abstract world $\mathcal{D}^\sharp$. Moreover please note that as $\mathcal{R}$ is finite, elements from our abstract world are machine representable and our abstract operators are computable. Let us now define sound abstract transformers on $\mathcal{D}^\sharp$, over-approximating the semantics of the atomic statements of language RM.

$$
\begin{aligned}
\mathbb{S}^\sharp\llbracket \texttt{r==0} \rrbracket &\triangleq \lambda R^\sharp \in \mathcal{D}^\sharp . \begin{cases} \bot & \text{if } R^\sharp = \bot \vee 0 \notin R^\sharp(r) \\ R^\sharp[r \mapsto [0;0]] & \text{otherwise} \end{cases} \\
\mathbb{S}^\sharp\llbracket \texttt{r!=0} \rrbracket &\triangleq \lambda R^\sharp \in \mathcal{D}^\sharp . \begin{cases} \bot & \text{if } R^\sharp = \bot \vee R^\sharp(r) = [0;0] \\ R^\sharp[r \mapsto [1;a]] & \text{else if } R^\sharp(r) = [0;a] \\ R^\sharp[r \mapsto [a;-1]] & \text{else if } R^\sharp(r) = [a;0] \\ R^\sharp & \text{otherwise} \end{cases} \\
\mathbb{S}^\sharp\llbracket \texttt{r++} \rrbracket &\triangleq \lambda R^\sharp \in \mathcal{D}^\sharp . \begin{cases} \bot & \text{if } R^\sharp = \bot \\ R^\sharp[r \mapsto [a+1;b+1]] & \text{otherwise when } R^\sharp(r) = [a;b] \end{cases} \\
\mathbb{S}^\sharp\llbracket \texttt{r--} \rrbracket &\triangleq \lambda R^\sharp \in \mathcal{D}^\sharp . \begin{cases} \bot & \text{if } R^\sharp = \bot \\ R^\sharp[r \mapsto [a-1;b-1]] & \text{otherwise when } R^\sharp(r) = [a;b] \end{cases}
\end{aligned}
$$

In order not to make the previous definitions too cumbersome, we assumed that $\infty \pm 1 = \infty$ and $-\infty \pm 1 = -\infty$. All above abstract operators are sound and or the best possible abstract operators that can be associated to their concrete counterpart.

We now provide a widening operator for the interval domain. As for the other set abstract operators, we define $\triangledown$ as a lifting to maps of the following operator:

$$
[a;b] \triangledown_0 [c;d] \triangleq [\begin{cases} -\infty & \text{if } c < a \\ a & \text{otherwise} \end{cases} ; \begin{cases} \infty & \text{if } d > b \\ b & \text{otherwise} \end{cases}]
$$

Finally please note that all soundness and computability conditions are met by our definitions.

Abstract semantics. Now that we have defined an abstract domain, an abstract semantics can be automatically defined by following the definition of the concrete semantics and replacing every operator by its abstract counterpart. In the following $\sqsubseteq$, $\sqcap$, $\sqcup$, $\bot$, $\top$ denotes the abstract operator on the lifted interval abstract domain.

$$
\begin{aligned}
&\mathbb{S}^\sharp\llbracket \texttt{while (r!=0) \{stmt\}} \rrbracket \triangleq \lambda R^\sharp \in \mathcal{D}^\sharp . \mathbb{S}^\sharp\llbracket \texttt{r==0} \rrbracket (\lim_{n \to \infty} F^n(\bot)) \\
&\quad \text{with } F(S^\sharp) = S^\sharp \triangledown (R^\sharp \sqcup \mathbb{S}^\sharp\llbracket \texttt{stmt} \rrbracket \circ \mathbb{S}^\sharp\llbracket \texttt{r!=0} \rrbracket (S^\sharp)) \\
&\mathbb{S}^\sharp\llbracket \texttt{if (r!=0) \{stmt}_t\texttt{\} else \{stmt}_e\texttt{\}} \rrbracket \triangleq \lambda R^\sharp \in \mathcal{D}^\sharp . \mathbb{S}^\sharp\llbracket \texttt{stmt}_t \rrbracket \circ \mathbb{S}^\sharp\llbracket \texttt{r!=0} \rrbracket (R^\sharp) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \sqcup \mathbb{S}^\sharp\llbracket \texttt{stmt}_e \rrbracket \circ \mathbb{S}^\sharp\llbracket \texttt{r==0} \rrbracket (R^\sharp) \\
&\mathbb{S}^\sharp\llbracket \texttt{stmt}_1\texttt{;stmt}_2 \rrbracket \triangleq \mathbb{S}^\sharp\llbracket \texttt{stmt}_2 \rrbracket \circ \mathbb{S}^\sharp\llbracket \texttt{stmt}_1 \rrbracket \\
&\mathbb{S}^\sharp\llbracket \texttt{skip} \rrbracket \triangleq \lambda R^\sharp \in \mathcal{D}^\sharp . R^\sharp
\end{aligned}
$$

Using the soundness and computability of abstract operators for atomic statements and for abstract set operators, the stabilizing property of the widening operator, and the fixpoint transfer theorems, we can show that the composition of these operators in the definition of our abstract semantics is sound and computable.

Theorem 2.6 (Soundness and computability). *For every statement* `stmt`*,* $\mathbb{S}^\sharp\llbracket \texttt{stmt} \rrbracket$ *is a sound and computable abstraction of operator* $\mathbb{S}\llbracket \texttt{stmt} \rrbracket$.

Application. Let us consider Program 2.1 written in RM where $\mathcal{R} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. We can show that $\mathbb{S}\llbracket \mathsf{add} \rrbracket \in \mathcal{D} \to \mathcal{D}$ is such that $\forall x, y \in \mathbb{N}, \mathbb{S}\llbracket \mathsf{add} \rrbracket(\{(\mathsf{a} \mapsto x, \mathsf{b} \mapsto y, \mathsf{c} \mapsto 0)\}) = \{(\mathsf{a} \mapsto 0, \mathsf{b} \mapsto 0, \mathsf{c} \mapsto x + y)\}$. Moreover if `a` or `b` is initially negative, then the set of reachable states at the end of

```
// 0 ⩽ a ⩽ 2∧0 ⩽ b ∧ c =0
●while (a!=0) {
      a--;
      c++
};
while (b!=0) {
      b--;
      c++
}
```

Program 2.1: add

Program 2.1 is empty. Indeed, no such inputs yield terminating executions and $\mathbb{S}\llbracket . \rrbracket$ ignores non terminating executions as it focuses on partial correction.

Let us now compare this concrete behavior with the one of our abstract semantics. We choose as starting point $S_0^\sharp = (\mathsf{a} \mapsto [0;2], \mathsf{b} \mapsto [0;\infty[, \mathsf{c} \mapsto [0;0])$. Consider the first while statement, we have:

$$\begin{aligned}
F^0(\bot) &= \bot \\
F^1(\bot) &= \bot \nabla (S_0^\sharp \sqcup \mathbb{S}\llbracket \mathsf{a--;c++} \rrbracket \circ \mathbb{S}^\sharp\llbracket \mathsf{a!=0} \rrbracket(\bot)) \\
&= S_0^\sharp \\
F^2(\bot) &= S_0^\sharp \nabla (S_0^\sharp \sqcup \mathbb{S}\llbracket \mathsf{a--;c++} \rrbracket \circ \mathbb{S}^\sharp\llbracket \mathsf{a!=0} \rrbracket(S_0^\sharp)) \\
&= S_0^\sharp \nabla (S_0^\sharp \sqcup (\mathsf{a} \mapsto [0;1], \mathsf{b} \mapsto [0;\infty[, \mathsf{c} \mapsto [1,1])) \\
&= S_0^\sharp \nabla (\mathsf{a} \mapsto [0;2], \mathsf{b} \mapsto [0;\infty[, \mathsf{c} \mapsto [0,1])) \\
&= (\mathsf{a} \mapsto [0;2], \mathsf{b} \mapsto [0;\infty[, \mathsf{c} \mapsto [0,\infty[&& \triangleq S_2 \\
F^3(\bot) &= S_2^\sharp \nabla (S_0^\sharp \sqcup \mathbb{S}\llbracket \mathsf{a--;c++} \rrbracket \circ \mathbb{S}^\sharp\llbracket \mathsf{a!=0} \rrbracket(S_2^\sharp)) \\
&= S_2^\sharp \nabla (S_0^\sharp \sqcup (\mathsf{a} \mapsto [0;1], \mathsf{b} \mapsto [0;\infty[, \mathsf{c} \mapsto [1,\infty[)) \\
&= S_2^\sharp \nabla (\mathsf{a} \mapsto [0;2], \mathsf{b} \mapsto [0;\infty[, \mathsf{c} \mapsto [0,\infty[)) \\
&= S_2^\sharp
\end{aligned}$$

From this fixed point computation we deduce that (● denotes the first while loop): $\mathbb{S}^\sharp\llbracket ● \rrbracket(S_0^\sharp) = \mathbb{S}^\sharp\llbracket \mathsf{a==0} \rrbracket(S_2^\sharp) = (\mathsf{a} \mapsto 0, \mathsf{b} \mapsto [0;\infty[, \mathsf{c} \mapsto [0;\infty[)$. By a similar computation on the second loop we get that: $\mathbb{S}^\sharp\llbracket \mathsf{add} \rrbracket(S_0^\sharp) = (\mathsf{a} \mapsto [0;0], \mathsf{b} \mapsto [0;0], \mathsf{c} \mapsto [0;\infty[)$. Therefore we were able to prove automatically that: whenever register a starts with a value in $[0;2]$, register b starts with a non negative value and register c starts with a null value then at the end of the execution, we have register a and b containing null value and register c contains a non negative value. Note that this result provides properties satisfied by an infinite number of program executions and could not have been obtained by executing the concrete semantics on every considered inputs. Obviously this is not the strongest invariant satisfied by Program 2.1, however the aforementioned result might be sufficient for the considered application (e.g. prove that no negative values are stored in registers).

```
if (a!=0) {
   a--;
   c++;
   if (a!=0) {
      a--;
      c++;
      while (a!=0) {
         a--;
         c++
      }
   } else {skip}
} else {skip}
```

Program 2.2: loop unrolling

```
while (i < 123) {
  i++;
}
```

Program 2.3: incrementation

2.3.2 Improving precision of the abstract fixpoint computation

Several techniques can be employed to improve the precision of the analysis for Program 2.1, some of which will be presented in the remainder of this chapter.

Delayed widening. The fixed point computation presented above resorted to widening as soon as it started iteration computation. We could soundly and without loosing computability remove this widening computation for any finite number of iterations.

Loop unrolling. This improvement can be described as a syntactic code transformation. As implied by its name, this technique separates the first loop iterations from one another. As an example loop ● would be replaced by the code fragment from Program 2.2. Applying this program transformation, we would have obtained the following more precise invariant: $\mathbb{S}^\sharp[\![\bullet]\!](S_0^\sharp) = (\mathsf{a} \mapsto [0;0], \mathsf{b} \mapsto [0;\infty[, \mathsf{c} \mapsto [0;2])$ at the end of the first loop.

For the presentation of the following two techniques we assume our language extended with some features so that Program 2.3 is part of our language. We provide neither the concrete nor the abstract semantics of added statements as it is straightforward.

Widening with thresholds. One of the biggest source of imprecision for static analysis by abstract interpretation is the use of widening operations to converge to an over-approximation of the least fixed point. As we have seen in our examples, jumps are performed from abstract elements representing two concrete values $[0;1]$ to the unbounded abstract element $[0;\infty[$. We have seen that unrolling loops or delaying widening could prevent this when the number of loop iterations performed in the concrete are relatively low. Widening with thresholds consists in adding several stops (a finite number) before over-approximating to ∞. As an example consider a program incrementing an integer while it is lower that `123` (Program 2.3), and the set of thresholds $\{10, 100, 1000\}$. The first iterations of the abstract fixpoint computation would go as follow:

$$\begin{aligned}[0;0] \triangledown [0;1] &= [0;10]\\ [0;10] \triangledown [0;11] &= [0;100]\\ [0;100] \triangledown [0;101] &= [0;1000]\end{aligned}$$

$[0; 1000]$ is then stable and can soundly be used as a loop invariant. Note that the number of iterations depends upon the number of thresholds and not on the actual value `123` from the program. Of course it is interesting to gather widening thresholds from the program syntax (e.g. `123` here) and even more interesting to gather them dynamically during the analysis.

Decreasing iterations. Once again consider Program 2.3. Assume that we start an analysis with the hypothesis that $(\mathtt{i} \mapsto [0; 100])$, following the fixpoint computation introduced above we would obtain the following loop invariant: $(\mathtt{i} \mapsto [0; \infty[)$. This represents an over-approximation of the actual fixpoint, that may be refined. This improvement can be obtained by applying F one more time and intersecting the results with the abstract fixpoint. This can be seen as continuing the iterations of F without widening. This process can then be iterated. Note however that it might not converge. For this reason we define a narrowing operator that ensures convergence.

Definition 2.17 (Narrowing operator)**.** Given a lattice $(A, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ we say that a binary operator $\triangle \in A \times A \to A$ is a *narrowing operator* if:

1. $\forall x, y \in A, x \sqcap y \sqsubseteq x \triangle y \sqsubseteq x$;
2. for all sequences $(x_i)_{i\in\mathbb{N}} \in A^{\mathbb{N}}$ the sequence $(y_i)_{i\in\mathbb{N}} \in A^{\mathbb{N}}$ defined by $(y_0 = x_0) \wedge (y_{i+1} = y_i \triangle x_{i+1})$ satisfies $\exists N \in \mathbb{N}, \forall k \geqslant N, y_k = y_N$.

Example 2.3 (Interval example continued)**.** We define here a narrowing operator for the interval domain:

$$[a; b] \triangle [c; d] = [\begin{cases} c & \text{if } a = -\infty \\ a & \text{otherwise} \end{cases} ; \begin{cases} d & \text{if } b = \infty \\ b & \text{otherwise} \end{cases}]$$

This operator can then be lifted to the interval abstract domain presented above.

Using this narrowing operator we can now improve the precision of the invariant inferred for Program 2.3. We start by applying the loop transformer on the previously obtained abstract fixpoint: $(\mathtt{i} \mapsto [0; 100]) \sqcup \mathbb{S}^\sharp[\![\mathtt{i++}]\!] \circ \mathbb{S}^\sharp[\![\mathtt{i < 123}]\!](\mathtt{i} \mapsto [0; \infty[) = (\mathtt{i} \mapsto [0; 123])$. We then narrow our abstract fixpoint with this value: $(\mathtt{i} \mapsto [0; \infty]) \triangle (\mathtt{i} \mapsto [0; 123]) = (\mathtt{i} \mapsto [0; 123])$. The result is a sound over-approximation of the concrete semantics of the loop and the process can be iterated. This result is more precise than the initial loop invariant. Note moreover that the postcondition obtained for Program 2.3 under starting hypothesis $0 \leqslant \mathtt{i} \leqslant 100$, is: $\mathbb{S}^\sharp[\![\neg(\mathtt{i < 123})]\!]((\mathtt{i} \mapsto [0; 123])) = (\mathtt{i} \mapsto [123; 123])$. This is actually the most precise invariant we could obtain and we were able to infer and prove it with 3 abstract executions of the loop body, and this invariant holds for 100 different program traces.

2.3.3 Composing abstract domains

In the previous section we focused on the analysis of a toy language containing only numerical variables. In a real-life programming language, the abstract domain performing the analysis needs to handle a lot of different features (strings, pointers, ...). Moreover we want to be able to easily adapt the analyzer to the kind of properties we want to prove. This is not achieved by defining from scratch a new abstract domain for each type of analysis, but rather by composing already existing ones. We have seen how domains can be derived from another by lifting their value semantics to an environment semantics. In this section we show two more methods for the derivation of new abstract domains: either as a direct product of two domains, or as a reduced product of two domains.

Definition 2.18 (Direct product)**.** Given two abstract domains $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$ abstracting $\mathcal{D}$, we define its product domain to be $\mathcal{D}_{1\times 2}^\sharp \triangleq \mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp$ with concretization and abstraction functions:

$$\begin{aligned} \gamma_{1\times 2} &\triangleq \lambda(R_1^\sharp, R_2^\sharp) \in \mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp.\gamma_1(R_1^\sharp) \cap \gamma_2(R_2^\sharp) \\ \alpha_{1\times 2} &\triangleq \lambda R \in \mathcal{D}.(\alpha_1(R), \alpha_2(R)) \end{aligned}$$

All abstract operators are lifted by applying the underlying operator on the corresponding component of the pair. As an example:

$$(R_1^\sharp, R_2^\sharp) \sqcup_{1\times 2} (R_1^{\sharp\prime}, R_2^{\sharp\prime}) \triangleq (R_1^\sharp \sqcup_1 R_1^{\sharp\prime}, R_2^\sharp \sqcup_2 R_2^{\sharp\prime})$$

The direct product provides an abstract domain that can represent more precise properties than both its arguments. However both underlying abstract domains will behave as if they were alone. A similar precision level would be obtained by analyzing the program once with $\mathcal{D}_1^\sharp$ only and once with $\mathcal{D}_2^\sharp$ only and conjuncting the results. $\mathcal{D}_1^\sharp$ is not able to use information discovered by $\mathcal{D}_2^\sharp$ and reciprocally.

For this reason a notion of reduced product was introduced. A *reduction operator* ρ is a function in $\mathcal{D}_{1\times 2}^\sharp \to \mathcal{D}_{1\times 2}^\sharp$. Its goal is to propagate information from the $\mathcal{D}_1^\sharp$ component to the $\mathcal{D}_2^\sharp$ component and reciprocally. As $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$ are both able to represent elements from $\mathcal{D}^\sharp$, $\mathcal{D}^\sharp$ can be used as a common language for the computation of reductions: $\rho_{1\times 2}(R_1^\sharp, R_2^\sharp) \triangleq (\alpha_1(\gamma_{1\times 2}((R_1^\sharp, R_2^\sharp))), (\alpha_2(\gamma_{1\times 2}((R_1^\sharp, R_2^\sharp))))$. This reduction operator concretizes the product abstract element, therefore obtaining the conjunction of the information from $R_1^\sharp$ and $R_2^\sharp$, and abstracts it back to $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$. However, this (optimally precise) reduction operator might not necessarily be computable. Moreover as we will show in the next section, some abstract domains do not provide an abstraction function α. For this reason reduced products are parameterized by a user-defined reduction function as none can be always automatically derived.

Definition 2.19 (Reduced product)**.** Given two abstract domains $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$ abstracting $\mathcal{D}$, and a reduction function ρ, we define the reduced product domain in the same manner as the product domain. However abstract operators are modified by applying the reduction operator after each computation of the classic product operator. As an example:

$$(R_1^\sharp, R_2^\sharp) \sqcup_{1\times 2} (R_1^{\sharp\prime}, R_2^{\sharp\prime}) \triangleq \rho(R_1^\sharp \sqcup_1 R_1^{\sharp\prime}, R_2^\sharp \sqcup_2 R_2^{\sharp\prime})$$

Contrary to the product domain, the reduced product of two domains $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$ can discover more precise properties that what could be inferred by two analyses: one using $\mathcal{D}_1^\sharp$ and one using $\mathcal{D}_2^\sharp$.

2.3.4 Concretization only framework

The results presented above made use of the Galois connection framework. Galois connections are defined by two functions $\alpha \in \mathcal{D} \to \mathcal{D}^\sharp$ and $\gamma \in \mathcal{D}^\sharp \to \mathcal{D}$. However some abstract lattices can be concretized via a monotonic function γ to a concrete lattice without there existing a best abstraction function α (an example is provided in Section 2.4.2, another in Section 7.3.1). Note however that the concretization function γ is sufficient to express the "meaning" of the abstract lattice, as we have seen in the previous definitions and examples. When we are given an abstract poset $(A, \sqsubseteq)$, a concrete poset $(C, \preccurlyeq)$ and a monotonic concretization function γ, we will say that we have a *representation* , denoted $(C, \preccurlyeq) \xleftarrow{\gamma} (A, \sqsubseteq)$. The definition of abstract domain provided in Section 2.3.1 can be modified so as to not require a Galois connection but a representation, all soundness requirements were defined with respect to γ and are unchanged. The computability and soundness result of Thm. 2.6 can be obtained as well, within the representation framework. For a more in-depth presentation of the representation framework we refer to [Bou92b].

2.4 Numerical Domains

As emphasized by the example of Section 2.3.1, abstract domains able to represent sets of numerical maps are an essential part of an analyzer built by abstract interpretation. Such abstract domains are heavily used as they enable: (1) to express constraints on program variables and (2) the definition of higher level abstractions. For instance in [Cou03] Patrick Cousot has shown how to devise predicate abstractions able to infer and prove functional properties of programs. Such abstractions state that some formula holds, the numerical free variables of these formulas are then delegated to some underlying numerical domain. Consider the following example: $\mathsf{P}_{\mathsf{a}}(x, y) \triangleq \forall i, x \leqslant i < y \Rightarrow \mathsf{a}[i] = 0$, where a is an array in some program. The predicate P_{a} states that the array was set to 0 for all indexes between x and y. The predicate in conjunction with the numerical abstraction $(x \mapsto [0; 1], y \mapsto [10; 10])$, states that: $\mathsf{P}_{\mathsf{a}}(0, 10) \vee \mathsf{P}_{\mathsf{a}}(1, 10)$. Another example of such a predicate abstraction can be found in [JM18], finally the string abstraction of Chapter 4 is also a predicate abstraction. The abstract semantics of numerical domains can moreover be lifted in several ways: (1) they can represent sets of environments with unbounded supports (this is particularly used for the representation of environments with unbounded memory locations, as is the case when dynamic memory allocation is allowed), see [GDD+04] or Section 6.7 for more details. (2) Moreover they can represent sets of environments with heterogeneous supports. This can be achieved by means of partitioning or by using the abstraction defined in Chapter 6. For all the above reasons, numerical domains have been widely studied. Popular examples include: Signs domain [CC76], Intervals [CC76], Simple congruences [Gra89, Gra97], Interval congruences [Mas93], Power analysis [Mas01], Linear equalities [Kar76], Linear congruences [Gra91], Trapezoidal congruences [Mas92], Polyhedra [CH78], Ellipsoids [Fer04], Zones [Min01a], Octagons [Min01b]. The example of Section 2.3.1 has presented the interval abstract domain in action. This example emphasized that the interval abstract domain was not able to convey relations between the values potentially stored in registers. This lack of relationality was one of the reason of the loss of precision (with respect to the concrete world) of the abstract analysis performed in Section 2.3.1. In addition to the already presented interval abstract domain, this thesis makes use of two relational domains: the octagon domain and the polyhedra domain. The rest of this section is organized as follow: Section 2.4.1 provides a definition of numerical abstract domains, Section 2.4.2 presents the polyhedra abstract domain, finally Section 2.4.3 recalls the definition and some results on the octagon abstract domain.

2.4.1 Definitions

In the following, we assume given a set $\mathbb{I} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$.

Numerical environments. Given a finite set of variables $\mathcal{W}$, we call *numerical environments* elements of $\mathcal{E} \triangleq \mathcal{W} \to \mathbb{I}$.

Numerical expressions. We define a set of *numerical expressions* on $\mathcal{W}$:

$$\begin{array}{lll} \mathit{expr} \triangleq & | \; v & v \in \mathcal{W} \\ & | \; i & i \in \mathbb{I} \\ & | \; \mathit{expr} \bowtie \mathit{expr} & \bowtie \in \{<, \leqslant, =, \neq, \geqslant, >, +, -, \times, /\} \end{array}$$

Let us denote as $\mathbf{fv} \in \mathit{expr} \to \wp(\mathcal{V})$ the set of free variables occurring in a expression. Given such an expression $\mathsf{expr} \in \mathit{expr}$ we define an evaluation function $\mathbb{E}[\![\mathsf{expr}]\!] \in \mathcal{E} \to \wp(\mathbb{I} \cup \mathbb{B})$ where

$\mathbb{B} \triangleq \{\texttt{true}, \texttt{false}\}$:

$$
\begin{aligned}
&\mathbb{E}[\![v \in \mathcal{W}]\!] \triangleq \lambda\rho \in \mathcal{E}.\{\rho(v)\}\\
&\mathbb{E}[\![i \in \mathbb{I}]\!] \triangleq \lambda\_ \in \mathcal{E}.\{i\}\\
&\mathbb{E}[\![\mathsf{expr}_1 \bowtie \in \{<, \leqslant, =, \neq, \geqslant, >\}\mathsf{expr}_2]\!] \triangleq \lambda\rho \in \mathcal{E}.\\
&\begin{cases} \{\texttt{true}\} & \text{when } \exists i_1 \in \mathbb{E}[\![\mathsf{expr}_1]\!](\rho) \cap \mathbb{I}, \exists i_2 \in \mathbb{E}[\![\mathsf{expr}_2]\!](\rho) \cap \mathbb{I}, i_1 \bowtie_{\mathbb{I}} i_2 \\ \emptyset & \text{otherwise} \end{cases} \cup\\
&\begin{cases} \{\texttt{false}\} & \text{when } \exists i_1 \in \mathbb{E}[\![\mathsf{expr}_1]\!](\rho) \cap \mathbb{I}, \exists i_2 \in \mathbb{E}[\![\mathsf{expr}_2]\!](\rho) \cap \mathbb{I}, i_1 \overline{\bowtie}_{\mathbb{I}} i_2 \\ \emptyset & \text{otherwise} \end{cases}\\
&\mathbb{E}[\![\mathsf{expr}_1 \bowtie \in \{+, -, \times, /\}\mathsf{expr}_2]\!] \triangleq \lambda\rho \in \mathcal{E}. \bigcup_{\substack{i_1 \in \mathbb{E}[\![\mathsf{expr}_1]\!](\rho)\\ i_2 \in \mathbb{E}[\![\mathsf{expr}_2]\!](\rho)}} \begin{cases} \emptyset & \text{if } i_2 = 0 \wedge \bowtie = / \\ \{i_1 \bowtie_{\mathbb{I}} i_2\} & \text{otherwise} \end{cases}
\end{aligned}
$$

where $\overline{\bowtie}$ denotes the usual $\neg$ transformation of a binary operator and $\bowtie_{\mathbb{I}}$ its usual evaluation as a function from $\mathbb{I} \times \mathbb{I}$ to $\mathbb{B}$.

Numerical atomic operators. We define two statements: assignment and test ($\texttt{Assume}(\texttt{expr})$ where $\texttt{expr} \in$ *expr*) with the following semantics in $\wp(\mathcal{E}) \to \wp(\mathcal{E})$:

$$
\begin{aligned}
\mathbb{S}[\![v \leftarrow \mathsf{expr}]\!] &\triangleq \lambda R \in \wp(\mathcal{E}).\{\rho[v \mapsto i] \mid \rho \in R \wedge i \in \mathbb{E}[\![\mathsf{expr}]\!](\rho) \cap \mathbb{I}\}\\
\mathbb{S}[\![\texttt{Assume}(\mathsf{expr})]\!] &\triangleq \lambda R \in \wp(\mathcal{E}).\{\rho \mid \rho \in R \wedge \texttt{true} \in \mathbb{E}[\![\mathsf{expr}]\!](\rho)\}
\end{aligned}
$$

Numerical abstract domains A *numerical abstract domain* is an abstract domain (in the sense of the definition from Section 2.3.1) with the following constraints:

- The concrete lattice that is represented is of the form $\mathcal{D} = \wp(\mathcal{E})$.
- The abstract domain provides sound numerical operators abstracting operators $\mathbb{S}[\![v \leftarrow \mathsf{expr}]\!]$ and $\mathbb{S}[\![\texttt{Assume}(\mathsf{expr})]\!]$.

Note moreover that in all the previous definitions, the set of variables was fixed for presentation purposes. We actually assume that the numerical abstract domain provides an abstract domain for every finite set of variables $\mathcal{W}$, as is the case for all numerical abstract domain. In the following whenever there is an ambiguity on the set of variables, operators will be superscripted with the set of variables on which it operates. Finally a numerical abstract domain provides the following additional operators that allow the addition and removal of variables from the environment: $\mathbb{S}^\sharp[\![\mathsf{remove}(v)]\!] \in \mathcal{D}^{\sharp^{\mathcal{W}}} \to \mathcal{D}^{\sharp^{\mathcal{W}\setminus\{v\}}}$ (where $v \in \mathcal{W}$) and $\mathbb{S}^\sharp[\![\mathsf{add}(v)]\!] \in \mathcal{D}^{\sharp^{\mathcal{W}}} \to \mathcal{D}^{\sharp^{\mathcal{W}\cup\{v\}}}$ (where $v \notin \mathcal{W}$), these operators satisfy the following soundness conditions:

$$
\begin{aligned}
&\forall X^\sharp \in \mathcal{D}^{\sharp^{\mathcal{W}}}, \gamma^{\mathcal{W}}(X^\sharp)_{|\mathcal{W}\setminus\{v\}} \subseteq \gamma^{\mathcal{W}\setminus\{v\}}(\mathbb{S}^\sharp[\![\mathsf{remove}(v)]\!](X^\sharp))\\
&\forall X^\sharp \in \mathcal{D}^{\sharp^{\mathcal{W}}}, \gamma^{\mathcal{W}}(X^\sharp) \uplus \{(v \mapsto i) \mid i \in \mathbb{I}\} \subseteq \gamma^{\mathcal{W}\cup\{v\}}(\mathbb{S}^\sharp[\![\mathsf{add}(v)]\!](X^\sharp))
\end{aligned}
$$

We say that these operators are exact whenever the equality holds, (as in Definition 2.14). Note that the classical $\texttt{forget}(v)$ operation, can be obtained by removing and adding back variable v.

2.4.2 The polyhedra abstract domain [CH78]

As mentioned in the introductory remarks to this section, the polyhedra domain is a relational domain. Given a set $\mathcal{W} = \{x_1, \ldots, x_n\}$, its abstract elements are of the form:

$$\bigwedge_{i=1}^{m} \sum_{j=1}^{n} a_{i,j} x_j + b_i \geqslant 0$$

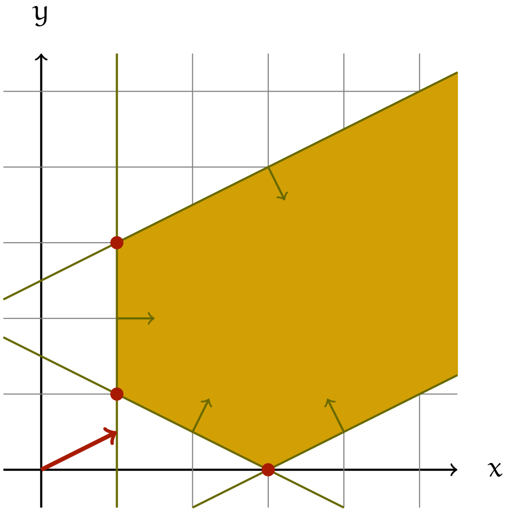


Figure 2.4: Polyhedron example

The concretization of such an abstract element is the set of numerical environments $\rho \in \mathcal{W} \to \mathbb{I}$ such that $\forall i \in \{1, \dots, m\}, \sum_{j=1}^{n} a_{i,j}\rho(x_j) + b_i \geqslant 0$. Constraints can actually contain strict inequalities however, for presentation purposes, we assume that all constraints are loose. Note that using such a set of linear constraints, we can express the following input/output relation holding at the end of Program 2.1 when all registers are non negative initially: $\texttt{a'} + \texttt{b'} - \texttt{c} = 0$, where a primed register represent the initial value of the register.

Duality

A polyhedron can also be described as the convex hull of a set of generators, that is vertices and rays, where rays represent directions in which the polyhedron is unbounded. As an illustration consider the example of Figure 2.4 showing a polyhedron in ●. This polyhedron can be represented by 4 constraints (in ●) or by 3 vertices and one ray (in ●). Numerical abstract domains provide several operators, in the case of polyhedra the generator/constraint duality can be found in the implementation of these operators. Indeed a sound over-approximation of the union can be obtained by computing the union of the two generator and ray sets; whereas a sound (and exact) over-approximation of the intersection of two polyhedra can be obtained by computing the union of the two constraint sets. The conversion between generator and constraint representation can be computed via Chernikova's algorithm[Che68, LV92]. Note however that this conversion might induce a blow-up in the representation (and therefore in the cost of subsequent operations), indeed transforming an hypercube from the constraints representation to the generator representation requires a number of generator exponential in the number of initial constraints. Most polyhedra libraries (e.g. PPL [BHZ08] or NewPolka [JM09]) work in the *double description framework* [MRTT53], in which both the constraint and generator representation are stored. The VPL library [FMP13, MMP17] provides abstract operators working on a *constraints only* representation of polyhedra

No Galois connection

As mentioned in Subsec 2.3.4 some abstract lattices do not enjoy a best abstraction function. The polyhedra lattice is a non complete lattice that does not enjoy a best abstraction function. Consider indeed as a concrete element the unit disk in dimension 2. There exists an infinite family of polyhedra encompassing this disk however none of them is more precise (for the inclusion relation) than all others.

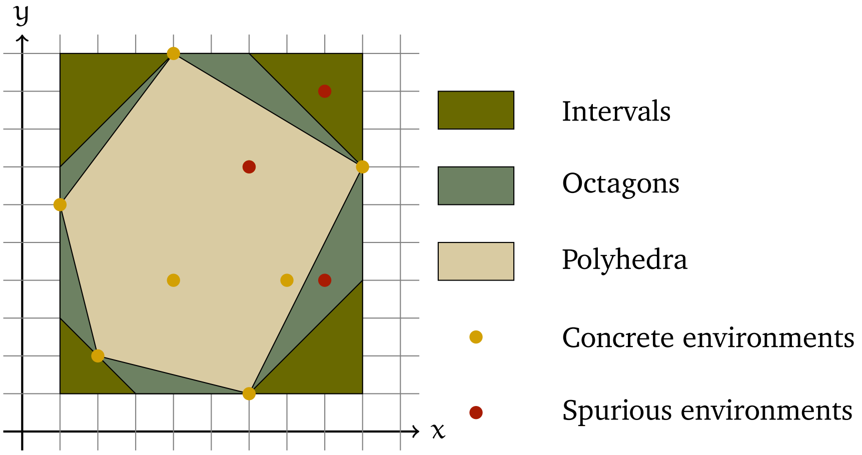


Figure 2.5: Comparison between numerical domains

2.4.3 The octagon abstract domain [Min01b]

We have presented two numerical abstract domain: the interval domain and the polyhedra domain. The interval abstraction provides a low-cost, low-precision abstraction, whereas the polyhedra abstraction is more precise, but exponentially (in the number of dimensions of the concrete environment) more costly. In this subsection we present the octagon domain [Min01b], this domain is more costly that the interval domain as it is relational, however it is not able to represent every linear relations between program variables as for the polyhedra domain. The octagon domain contains elements that are a conjunction of constraints of the form $\pm x \pm y \leqslant c$. The concretization of such a set of constraints is defined as for the polyhedra abstraction. The number of irredundant constraints in a polyhedra is unbounded, whereas octagons on n dimensions can be restricted to $2n^2$ constraints and retain their maximal level of precision (as subsequent constraints would necessarily be redundant). Abstract transformers with a $\mathcal{O}(n^3)$ time complexity have been devised for the octagon abstract domain.

The octagon domain is able to express the most precise loop invariant of the first loop of Program 2.1: $\mathsf{a} + \mathsf{c} \leqslant 0 \wedge -\mathsf{a} - \mathsf{c} \leqslant 0$. However the complete program invariant can not be expressed by the octagon domain.

The octagon domain is strictly more expressive than the interval domain. Indeed $x \in [a; b]$ can be expressed with the two octagonal constraints: $x + x \leqslant 2b$ and $-x - x \leqslant -2a$.

As a conclusion to the presentation of the three abstract domains: intervals, octagons, and polyhedra, Figure 2.5 provides the classical graphical comparison between these three domains. This figure underlines that polyhedra are more precise than octagons, themselves more precise than intervals.

2.4.4 Notations

In the remainder of thesis, we assume given, for every finite set $\mathcal{W}$, a numerical abstract domain $\mathcal{N}^{\mathcal{W}}$ providing a representation $(\wp(\mathcal{W} \to \mathbb{I}), \subseteq) \xleftarrow{\gamma^{\mathcal{W}}} (\mathcal{N}^{\mathcal{W}}, \sqsubseteq^{\mathcal{W}})$. We assume moreover that $\mathcal{N}^{\mathcal{W}}$ provides abstract transformers for the atomic statements from Section 2.4.1 (these are denoted as $\mathbb{S}^\sharp[\![\square]\!]^{\mathcal{W}}$), as well as for abstract operators for set operations: $\sqcap^{\mathcal{W}}$, $\sqcup^{\mathcal{W}}$, and finally a widening operator: $\triangledown^{\mathcal{W}}$. Moreover we overload notations and given $x \in \mathcal{W}$, $y \notin \mathcal{W}$, and $N^\sharp \in \mathcal{N}^{\mathcal{W}}$ then $N^\sharp[x \mapsto y] \in \mathcal{N}^{\mathcal{W} \cup \{y\} \setminus \{x\}}$ is the abstract element obtained by renaming x into y. If $\mathcal{W}'$ is another finite set $N^\sharp_{|\mathcal{W}'} \in \mathcal{N}^{\mathcal{W}'}$ is obtained by iteratively removing variables that are in $\mathcal{W}$ and not in $\mathcal{W}'$ and iteratively adding variables that are in $\mathcal{W}'$ and not in $\mathcal{W}$ (using $\mathbb{S}^\sharp[\![\mathtt{add}]\!]$ and $\mathbb{S}^\sharp[\![\mathtt{remove}]\!]$). When $N^\sharp \in \mathcal{N}^{\mathcal{W}}$, we call $\mathcal{W}$ the *support* of $N^\sharp$, denoted as $\mathbf{def}(N^\sharp)$. Finally when $(e_i)_{i \in \{1,n\}} \in$

expr, we denote as $\mathbb{S}^\sharp[\![\mathtt{Assume}(e_1, \ldots, e_n)]\!](N^\sharp)$ the composition of all $\mathtt{Assume}$: $\mathbb{S}^\sharp[\![\mathtt{Assume}(e_1)]\!] \circ \cdots \circ \mathbb{S}^\sharp[\![\mathtt{Assume}(e_n)]\!](N^\sharp)$.

Constraints extraction. We assume that, given a numerical abstract element $N^\sharp$, we can extract a finite set of constraints satisfied by $N^\sharp$, those are denoted **constraints**$(N^\sharp)$. For example if the numerical domain is the interval domain, constraints have the form $\pm x \geqslant a$. If the numerical domain is the octagon domain the **constraints** operator yields all the linear relations among variables that define the octagon.

2.5 Conclusion

In this chapter we provided a brief overview of the results in abstract interpretation that are required for the understanding of the remainder of the thesis.

Even though we presented the Galois connection framework, most of the abstractions used and defined in this thesis will not necessarily enjoy a best abstraction function. For this reason we will mainly use the notion of representations.

Numerical abstract domains will be extensively used and will be lifted in two different manners: to represent input/output relations, as sketched in the polyhedra presentation, to represent sets of environments not all defined on the same definition sets.

Chapter 3

Mopsa

3.1 Introduction

During this thesis, I was part of the Mopsa project [MOJ18], the goal of which is the development of a static analyzer. For this reason, in addition to my participation to the development of the framework of the analyzer, I implemented all abstract domains from subsequent chapters inside the Mopsa analyzer. We provide in this chapter a brief overview of the collective results of the Mopsa team regarding this analyzer. Moreover, we will underline in subsequent chapters which features of the analyzer were used for the development of the abstract domains.

3.1.1 Motivations

In the introductory chapter we mentioned that several tools based on the abstract interpretation framework have been designed and successfully used. Most tools however focus on analyzing families of C-like or Java-like languages, and do not target dynamic languages (e.g., Python, JavaScript) and *a fortiori* cannot handle both kinds of languages. Moreover, while all tools are based on a modular combination of abstractions, as advocated by abstract interpretation [CC79], these are often restricted to specific abstractions: e.g., Astrée [CCF$^+$06a] achieves a reduced product of numeric abstractions but has a monolithic memory abstraction; Frama-C's abstractions can be composed either through coarse-grain plug-ins, or as non-relational value abstractions [BBY17]. To try and address these issues, and push abstract interpretation frameworks further, we have started the design of a *Modular Open Platform for Static Analysis* (Mopsa) in OCaml. It is based on a standard interpreter by induction on the syntax and a collection of abstractions (see, e.g., [BCC$^+$10]) but with unique features:

- all domains share a common extensible Abstract Syntax Tree datatype; it can express both high-level syntax close to the source languages, and simpler intermediate ones used internally by abstractions (e.g., numeric fragments);
- we eschew static simplifications common to frameworks (e.g., LLVM [LA04]); transformations are performed dynamically during interpretation and benefit from inferred facts;
- all domains share a common interface and are easy to compose; notably, iterators are viewed as domains, and domains are responsible for statement simplification;
- domains need only handle a fragment of the AST; they can be reused after extending the AST to new languages.

Mopsa is a work in progress: its architectural foundations are implemented, but it is not ready to analyze realistic programs. As a proof of concept, analyses for both a fragment of C and Python have been implemented (during this thesis I worked only on the C component of the analyzer), showing it can support legacy abstractions (e.g., numeric domains, low-level C memory models [Min06a]) and is a great help developing novel ones (e.g., modular C analysis [JMO18] from Chapter 5, Python analysis [FOM18], tree manipulating language analysis [JMO19] from Chap-

```
module type DOMAIN = sig
  (* Lattice definition *)
  type t
  val bottom   : t
  val top      : t
  val leq      : t -> t -> bool
  val join     : t -> t -> t
  val meet     : t -> t -> t
  val widening : t -> t -> t
  (* Transfer functions *)
  val exec : stmt -> ('a, t) man -> 'a flow -> 'a flow option
  val eval : expr -> ('a, t) man -> 'a flow -> 'a evl option
  val ask  : 'r query -> ('a, t) man -> 'a flow -> 'r option

end
```

Program 3.1: Unified signature for abstract domains.

ter 7). Mopsa was extended with a "universal" language, that is a toy-language which is helpful for testing new abstractions. This language mainly allows the manipulation of numeric variables. It has been extended to feature tree manipulating operations, as detailed in Chapter 7.

3.1.2 Outline of the chapter

Sections 3.2 and 3.3 present the domain signature and domain composition; Section 3.4 reports on early experiments (analyzing part of Juliet for C and Python's regression tests); Section 3.5 concludes.

3.2 Unified Domains Signature

Abstract domains are OCaml modules implementing the `DOMAIN` signature in Program 3.1. In the following, we describe the main types and functions, and state the core design goals.

3.2.1 Lattice

Each domain defines a type `t` characterizing its abstract state, as well as corresponding lattice operators (`join`, `widening`, etc.). The internal abstract state of a domain is private: other domains have no *direct* access. This separation ensures a low coupling between domains. Note that: on the one hand, abstract domains have a private type, unknown by the other domains, but on the other hand, they can extend a public type used by other abstract domains to query them on properties that they know might hold. This mechanism is called querying and is shown below.

3.2.2 Managers

During the analysis of one statement, a domain may require computing the post-condition of statements handled by other domains. Indeed, consider the two following examples: (1) the domain handling loop iterations need to execute the body of the loop, (2) the domain handling C pointers will rewrite the statement `*p=0` into `a=0` (when `p` points to `a`), it will then delegate

```
type ('a, 't) man = {
  (* Functions on the global abstract element *)
  bottom    : 'a;
  top       : 'a;
  leq       : 'a -> 'a -> bool;
  join      : 'a -> 'a -> 'a;
  meet      : 'a -> 'a -> 'a;
  widen     : 'a -> 'a -> 'a;

  (* Accessors to the domain's abstract element *)
  get : 'a -> 't;
  set : 't -> 'a -> 'a;

  (** Transfer functions *)
  exec : stmt -> 'a flow -> 'a flow;
  eval : expr -> 'a flow -> ('a, expr) evl;
  ask : 'r. 'r query -> 'a flow -> 'r;
}
```

Program 3.2: Manager definition.

the post-condition computation to another domain. To allow inter-domain communication without sacrificing modularity, each domain transfer function is not defined over its private type `t`, but on the product of types from all domains: using notations from Section 2.3.3, a domain is given an element from $\mathcal{D}^\sharp_{1\times 2}$ rather than its component from $\mathcal{D}^\sharp_1$. To ensure that each domain can be programmed independently from the others, the product is a type parameter `'a`, and we use an encoding of polymorphic records in OCaml: the manager `('a, t) man` is a record providing lattice operators on `'a`, transfer functions over all domains in `'a`, and accessor functions `get: 'a -> t` and `set: t -> 'a -> 'a` to allow a domain to access and update its private abstraction within the global abstraction. The complete definition of the manager type is shown in Program 3.2. Note that the manager also provides lattice operators on the complete abstraction, this can be used for example by the loop iterator to join the results from successive body analysis.

3.2.3 Flows

Mopsa operates by induction on the syntax of the program. To handle non-local control flows, we use continuations (similarly to Astrée [BCC+10]): we collect not only environments reaching the current program location, but also those at previously encountered jump locations. Suspended flows are merged back into the current flow when reaching the corresponding jump target. Expressing flows as continuations makes it possible to abstract very complex non-local control, such as generators in Python [FOM18]. Flow continuations are implemented as maps from *flow tokens* into the global abstraction `'a`. Tokens belong to an extensible type, thus allowing a domain to add new control flow abstractions independently from the remaining domains.

Example 3.1. Consider the syntax of RM from Section 2.3.1 extended with two statements ($\mathcal{L}$ is a finite set of labels, containing a special label `cur`):

$$\mathit{stmt} \triangleq \cdots \mid \mathcal{L} : \mathit{stmt} \mid \texttt{goto}\ \mathcal{L}$$

```
type token += T_goto of Ast.label

let exec stmt man flow =
  match stmt with
  | S_c_goto(label) -> (* Case of  𝕊⟦goto label⟧ *)
    let cur_env = Flow.find T_cur man flow in
    let flow' = Flow.add (T_goto label) cur_env man flow in
    let flow'' = Flow.set T_cur man.bottom man flow' in
    Post.return flow''
  | S_c_label(label, stmt) -> (* Case of  𝕊⟦label: stmt⟧ *)
    let goto_env = Flow.find (T_goto label) man flow in
    let flow' = Flow.add T_cur goto_env man flow in
    let flow'' = man.exec stmt flow' in
    Post.return flow''
  | _ -> None
```

Program 3.3: Transfer function for forward `goto` statements.

The abstract domain from Chapter 2 was defined over the lifted interval domain $\mathcal{D}^\sharp$. The concrete (resp. abstract semantics) is lifted to $\mathcal{L} \to \mathcal{D}$ (resp. $\mathcal{D}^\sharp$). The lattice operations are lifted pointwise, the abstract semantics of statements is lifted by applying the corresponding transformer on the environment associated with the `cur` label. The concrete semantics of the two new statements is defined as:

$$\mathbb{S}[\![\texttt{l: stmt}]\!](m \in \mathcal{L} \to \mathcal{D}) \triangleq \mathbb{S}[\![\texttt{stmt}]\!](m[\texttt{cur} \mapsto m(\texttt{cur}) \cup m(\texttt{l}), \texttt{l} \mapsto \emptyset])$$

$$\mathbb{S}[\![\texttt{goto l}]\!](m \in \mathcal{L} \to \mathcal{D}) \triangleq (m[\texttt{cur} \mapsto \emptyset, \texttt{l} \mapsto m(\texttt{cur})])$$

The abstract semantics is then derived by replacing $\emptyset$ with $\perp$ and $\cup$ with $\sqcup$.

Example 3.2. We give in Program 3.3 an example of a transfer function for handling forward `goto` statements in C (the concrete/abstract semantics is similar as the one defined on our RM toy language from the previous example). Flows are enriched with a new token `T_goto` annotated with the target label. When reaching a statement `goto label`, the current environments are moved to token `T_goto label`, before being reset to $\perp$ because the next control location becomes unreachable. Encountering `label: stmt` joins environments associated to `T_goto label` into current environments, and returns a post-condition of `stmt` via the manager. The default case returns `None`, indicating that other domains should handle the other constructions. Such a pure iterator domain does not maintain any abstract state by itself, hence `t = unit`. Despite its brevity, the code in Figure 3.3 is the complete transfer function of a domain that adds C-style forward `goto` to an analysis. It is completely decoupled from other domains, and thus highly reusable.

3.2.4 Evaluations

Domains in Mopsa can implement dynamic rewriting rules to simplify expressions by exploiting the available abstract information. As an example consider the transformation from modular arithmetic from C expressions into mathematical arithmetic, easier to handle in relational numeric domains, after ensuring the absence of overflows in the current state. Another contribution of Mopsa is the ability to handle natively disjunctive evaluations. Domains can return different

```
type ('a, 'e) evl_case = {
  expr    : 'e option;
  flow    : 'a flow;
}
type ('a, 'e) evl = ('a, 'e) evl_case Dnf.t
```

Program 3.4: Type of evaluations in Mopsa.

```
(* Case of  𝔼⟦list.__getitem__(self, i)⟧  *)
let flow1, flow2, flow3 = getitem_cases self i man flow in
let evl1 =
  let flow' = man.exec (mk_raise "IndexError") flow1 in
  Eval.empty flow'
in
let evl2 = Eval.singleton (summary self) flow2 in
let evl3 =
  let tmp = mk_tmp () in
  let flow' = man.exec (expand tmp (summary self)) flow3 in
  Eval.singleton tmp flow'
in
Eval.or_list [evl1; evl2; evl3]
```

Program 3.5: Evaluation of index access on lists in Python.

results for different parts of the input abstract environment, indeed the `'a evl` type in the previous definitions is a disjunctive normal form of evaluation cases. These evaluation cases contain an expression e and an abstract environment in which it is sound to rewrite the original expression into e. This can be seen in the definition of the Program 3.4. Consider the following example from Python analysis.

Example 3.3. Program 3.5 shows the evaluation of an index access on Python lists. As lengths of lists are parametric, they can not be represented by extension (i.e. by allocating one dimension of the numerical domain for each element of the list). For this reason we use a summarization variable that represents all possible values of all possible elements in the list (see [GDD+04]). Moreover in order to track the number of elements in the list, a special numerical variable is added to the numerical domain, this variable encodes all possible list lengths, as done in [FOM18]. When analyzing a read access, the function `getitem_cases` is used to partition the pre-condition w.r.t. the length. The first flow `flow1` represents the case when index `i` is outside the size of `self`: $\texttt{flow1} \triangleq \mathbb{S}^\sharp[\![\texttt{Assume}(\neg(-\texttt{len}(\texttt{self}) \leqslant \texttt{i} < \texttt{len}(\texttt{self})(\triangleq \mathrm{C}_1)))]\!](\texttt{flow})$. The associated evaluation is an empty expression since an exception is raised. The second flow `flow2` is obtained by filtering the pre-condition verifying that the list contains exactly one element: $\texttt{flow2} \triangleq \mathbb{S}^\sharp[\![\texttt{Assume}(\texttt{len}(\texttt{self}) = 1(\triangleq \mathrm{C}_2))]\!] \circ \mathbb{S}^\sharp[\![\texttt{Assume}(\mathrm{C}_1)]\!](\texttt{flow})$. In this case, the summary variable is returned as a sound simplification of index access. Finally, when the list contains at least two elements: $\texttt{flow3} \triangleq \mathbb{S}^\sharp[\![\texttt{Assume}(\neg\mathrm{C}_2)]\!] \circ \mathbb{S}^\sharp[\![\texttt{Assume}(\mathrm{C}_1)]\!](\texttt{flow})$ a temporary variable is created as a copy of the summary variable using the expand function, similarly to the case of summarized arrays from [GDD+04].

This pre-condition partitioning mechanism will be used extensively in the abstract domains defined in subsequent chapters.

3.2.5 Queries

When a domain needs specific information maintained by another domain, it can fetch them via *queries*, similarly to Astrée [CCF+06a]. The type `_ query` is an extensible public GADT type that can be enriched by domains. For example, an interval query can be defined by:

```
type _ query += Q_interval: expr -> Itv.t query
```

Numeric domains handle requests to `Q_interval` in the transfer function `ask`, and client domains retrieve this information via their manager by calling `man.ask (Q_interval e) flow`. GADT typing ensures that the returned value has type `Itv.t`. An abstract domain defining a new query of type `u` provides a `join` and `meet` lattice operator on type `u`. These are then used to compose results when several domains in the abstraction are able to answer the query, for example both the interval and the polyhedra abstract domain can answer interval queries.

3.3 Domain Composition

The global abstraction of an analysis instance is constructed at run-time by composing domains according to a user-given configuration file. An illustrative example is depicted in Figure 3.1 representing a simplified version of an analysis of C. The configuration starts with a long sequence of iterators, C program to C goto, *i.e.* state-less domains that handle individual parts of the C compound syntax by induction, including loops, `switch`, `goto`, etc. The configuration merges domains reused from the Universal toy-language (underlined) and C-specific domains. Following these iterators, the C analysis contains a composition of domains that handle atomic statements such as assignments and tests. Dynamic memory is handled by heap using recency abstraction [BR06]: each allocation site is associated with at most two abstract blocks, one representing the lastly allocated block at this site, and one representing all the blocks allocated before — this domain could be easily replaced with any domain that partitions the possibly unbounded set of allocated blocks into a bounded set of abstract blocks. Each variable or abstract heap block is then decomposed into a set of virtual variables, called *cells*, of scalar type, by C cells. In order to handle transparently union types and type-punning, we use the cell abstraction from [Min06a], where the decomposition is adapted dynamically according to the actual access pattern during the execution (rather than based on the static type, which can be deceiving), a more in-depth presentation of the cell abstraction will be provided in Chapter 4. The cell domain is composed, using a reduced product, with the C smash abstraction. This abstraction summarizes several cells into one, this is particularly useful for the representation of arrays. Domain C pointers is then used to handle assignments and tests on pointers, while the numerical statements are delegated to an interval numerical abstraction boxes lifted to a semantics of machine integers.

Three types of generic composers are used to connect leaf domains:

Iterators. This composer iterates sequentially over a set of domains, which is materialized with ↠ in Figure 3.1. Lattice operators are defined pointwise, and transfer functions are called in sequence until one succeeds: domain i is invoked only if previous domains $j < i$ returned `None`. This composition is useful to combine a set of iterators defining transfer functions for disjoint parts of the AST, such as loops, function calls, etc. In addition, it allows plugging new iteration schemes easily, e.g., replacing the induction-based iterators with CFG-based ones.

Reduced products. The classic method employed in most static analyzers (e.g., [BCC$^+$10]) for creating reduced products is via cascading binary functors. Mopsa provides instead a generic n-ary reduced product composer that enables better propagation of reduction channels. Lattice operators are defined pointwise and transfer functions of argument domains are called in parallel. All post-conditions are given to a user-defined reduction operator, that can access directly abstract elements of any domain in the pool and can exploit the published reduction channels to refine the final post-condition.

Stacks. Another novelty in Mopsa is a stacked reduced product of n functors sharing the same subordinate domain instance. This form of composition is useful when several domains depend on an external domain to delegate management of parts of their abstract state. Consider the case of cell and smashing abstractions from [Min06a, BCC$^+$10] in Figure 3.1. Both domains need a pointer domain for address resolution and a numeric domain for abstracting numeric environments. By sharing the same numeric domain instance, relations between cells and smashed variables can be inferred. Please note that for presentation purposes and as this was not crucial to the understanding of the results of subsequent sections, the signatures presented in this section have been simplified *w.r.t.* the actual implementation, we refer to [MOJ18] for a more complete presentation. The signature of `exec` from Program 3.1 does not actually permits the use of sharing under a reduced product. Indeed, when a domain C is shared by two domains A and B, combined in a reduced product, the handling of a statement by A and B might generate two different sets of statements, propagated to C, thus yielding two different elements from C. These are merged back together by a function `merge` defined by C, taking as arguments the two diverging abstract elements and the log of all statements propagated to C by A and by B.

Results from this thesis in Mopsa. As an illustration of composition mechanisms, we now show how results from subsequent sections will be added/composed in Mopsa. Consider the "default" C configuration from Figure 3.1.

- The string analysis from Chapter 4 will be incorporated by replacing the domain C smash by the domain C string, where the C string domain is defined in Chapter 4. Moreover the boxes domain will be replaced with a polyhedra domain in order to allow the expression of relations.
- The modular iterator from Chapter 5 will then be incorporated in Mopsa by replacing the C interproc domain with the one defined in Chapter 5.
- Results from Chapter 6 were tested in the universal language and in the C language by replacing the numerical component boxes with the application of the Clip functor (defined in Chapter 6) to a numerical abstraction.

Finally the tree abstraction from Chapter 7 will be added as a stacked domain above the Clip abstraction in the configuration for the universal language.

Note that subsequent chapters will only present analyses based on Universal or C, never Python. Python configurations and benchmarks from this chapter were only presented for the purpose of illustrating the behavior of Mopsa.

3.4 Experiments

We instantiated two analyzers, one for C (Figure 3.1) and one for Python (Figure 3.2). We note that the configurations share several domains (underlined), including iterators, heap and numeric abstractions. C benchmarks were taken from the Juliet Test Suite (v 1.3) for C/C++. We selected all the C tests corresponding to the following categories: `CWE190` on integer overflows, `CWE369` on divisions by zero, and `CWE476` on null pointer dereferences. Each category provides

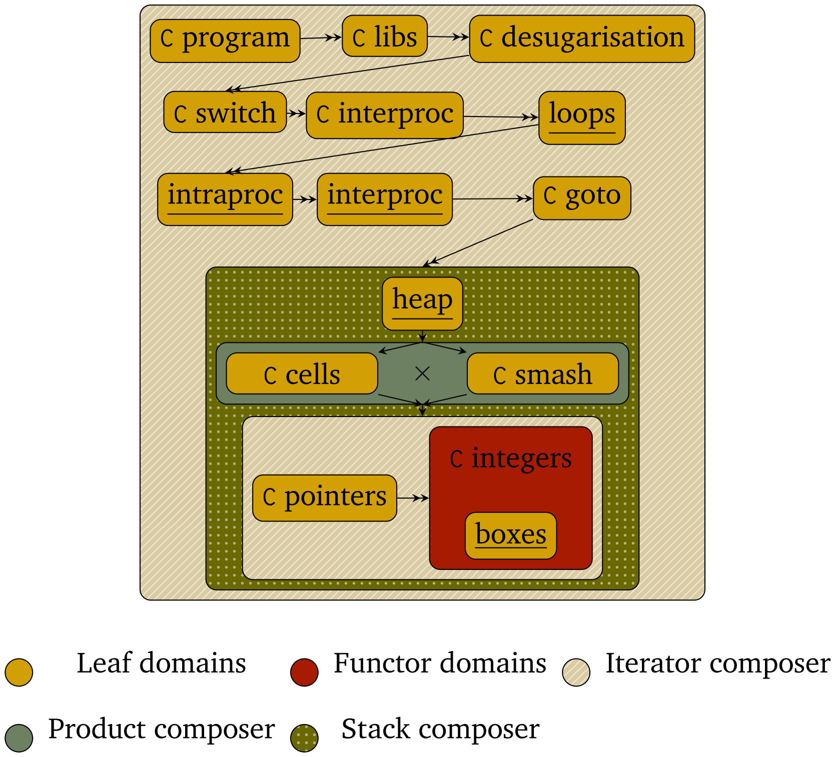


Figure 3.1: C analysis configuration.

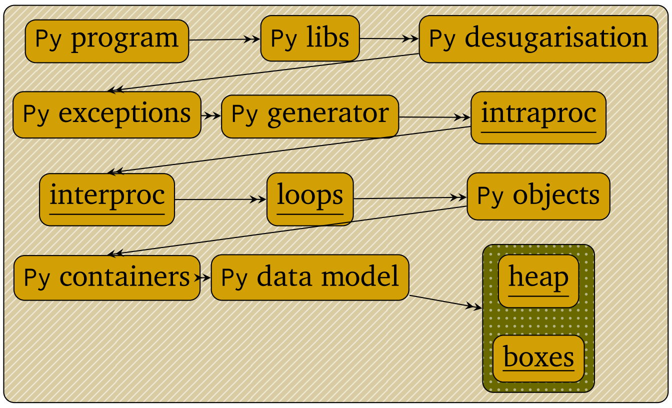


Figure 3.2: Python analysis configuration.

several test cases, themselves split in two programs: a bad program containing an error, and its corrected version. Python benchmarks were performed on regression tests from the official Python 3.6.3 distribution, that test the builtins of the language and the standard library. The current configuration for the Python analysis does not provide an analyzer able to handle all the 500 regression tests; we selected 9 tests in the scope of our configuration (see [FOM18] for more information). Tables 3.1 and 3.2 provide results of these experimentations: ✓ are good programs for which no alarm was raised, ✓ are bad programs for which an alarm was raised, ✗ are good programs for which the analyzer raised an alarm (false positive), ✱ are programs that could not be handled by the analyzer (e.g., by lack of stubs). False positives in `CWE369` are due to the use of a non-partitioned numerical domain for the analysis of programs testing divisions by zero; thanks to the modularity of the analyzer, this could be avoided by adding a partitioning domain.

Regression test	Lines	Tests	Time	✓	✗	✱	Coverage
`test_augassign`	273	7	645ms	4	2	1	85.71%
`test_baseexception`	141	10	20ms	6	0	4	60.00%
`test_bool`	294	26	47ms	12	0	14	46.15%
`test_builtin`	454	21	360ms	3	0	18	14.29%
`test_contains`	77	4	418ms	1	0	3	25.00%
`test_int_literal`	91	6	29ms	6	0	0	100.00%
`test_int`	218	8	88ms	3	0	5	37.50%
`test_list`	106	9	88ms	3	0	6	33.33%
`test_unary`	39	6	11ms	2	0	4	33.33%
Total	1693	97	1.71s	40	2	55	43.30%

Table 3.1: Benchmarks on regression tests of Python 3.6.

tests	Loc	Tests	Time	✓	✓	✗	✱	Coverage
`CWE190`	440k	6840	11.0mn	2584	3213	0	1043	84.75%
`CWE369`	109k	1368	3.30mn	76	380	304	608	55.55%
`CWE476`	25k	522	0,207mn	270	252	0	0	100%
Total	574k	8730	14.5mn	2930	3845	304	1651	77.60%

Table 3.2: Benchmarks on CWE from the Juliet test suite.

3.5 Conclusion

We have presented the design principles of Mopsa, a new platform for static analysis development through compositions of highly reusable abstractions. It is not yet feature-complete, but it is already successfully used in research projects to support the analysis of non-trivial C and Python codes [JMO18, FOM18]. In the future, we plan to extend the support for C standard libraries and Python built-ins to be able to analyze realistic codes. Other research projects on Mopsa include static analysis of novel properties beyond safety, such as a portability analysis. These future works are those of the Mopsa project team and are unrelated with the additions made to Mopsa in the following chapters.

Chapter 4

String abstraction

In Chapter 5 we provide the definition of a modular analyzer. As a proof of concept of the expressiveness of this modular analyzer and to meet some cost/precision trade off needs in the Mopsa project, we decided to add to our C analyzer a predicate abstraction targeting out-of-memory accesses in string manipulations in C. As we were not able to directly reuse results from the literature (detailed in Section 4.5), we defined a new predicate abstraction presented in this chapter. Furthermore the modular analysis of Chapter 5 and the string analysis from this chapter were both designed on top of an already existing analyzer for the C language, introduced in this chapter.

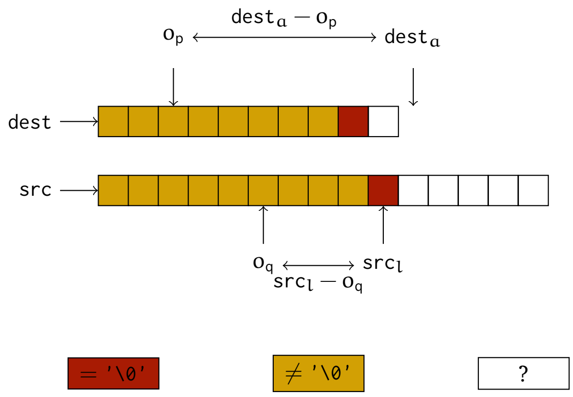
4.1 Introduction

4.1.1 Motivations

In a C string, a '\0' character designates the end of the string. Henceforth the length of a string is defined to be the index of the first '\0' character appearing in the string. As emphasized

```
while (*q != '\0') {
  *p = *q;
  p++;
  q++;
}
*p = *q;
```

Program 4.1: strcpy



```
for (;;) {
  if (!(*s = *t)) return ; ++s; ++t;
  if (!(*s = *t)) return ; ++s; ++t;
  if (!(*s = *t)) return ; ++s; ++t;
  if (!(*s = *t)) return ; ++s; ++t;
 }
```

Program 4.2: strcpy from Qmail

in Program 4.1, the correctness of a string manipulating program (in the sense that it does not yield an out of bound memory access) depends upon the length and the allocated size of the buffer in which it is contained. Therefore in the fashion of [WFBA00] we summarize strings by two values: the position of the first `'\0'` character (denoted as e.g. $\texttt{src}_l$) and the maximal string size (i.e. the buffer size, denoted as e.g. $\texttt{src}_a$). This summarization will make use of the capacity of an underlying numerical domain to relate the summary variable denoting the length of the string, with program variables or offsets of pointers. Such abstractions are called *predicate abstractions* [Cou03].

The fragment of `C` on which we want to perform out-of-bounds analysis supports string manipulations, unions, structures, arrays, memory allocations (static and dynamic), pointers, pointer arithmetics, pointer casts, function calls, ... Accordingly we need to build our analyzer upon an existing analyzer able to deal with low level features of C.

By using the idea of representing a string as its length using a numeric abstraction, our analyzer is able to handle the `strcpy` example of Program 4.1. More precisely, consider that `char* p` points to some `char[10] dest` string and `char* q` points to some `char[20] src` string with $\texttt{dest} \neq \texttt{src}$, furthermore let variables o_q, o_p denote the initial offset of `p` and `q`. Our analyzer is able to prove that if $\texttt{src}_l < \texttt{src}_a$ and $\texttt{src}_l - o_q < \texttt{dest}_a - o_p$ then no out of bounds access is performed.

In order to underline the fact that such functions are often user-redefined, consider Program 4.2, showing the `strcpy` function as defined in Qmail. Our analyzer was able to infer the same results as for Program 4.1. The design of our string analysis on top of a C analyzer already handling most of the C language features provides non negligible advantages: we only have to provide operators for a small subset of the language, the capabilities of the analyzer do not depend upon the manner in which strings are accessed.

4.1.2 Outline of the chapter

Section 4.2 describes the subset of `C` we wish to analyze, Section 4.3 defines a low-level `C` abstraction upon which our analyzer is based, Section 4.4 details the String abstract domain. Finally Section 4.5 gives an overview of related works, while Section 4.6 concludes.

4.2 Syntax and concrete semantics

In this section we recall some of the definitions of the cell concrete semantics, which is a concrete semantics for the C language originally defined in [Min06a]. The descriptions and examples of the cell semantics and the cell abstraction made in this section and the following are sufficient for the understanding of subsequent abstraction definitions. We refer to [Min06a, Min13] for a more complete presentation.

Syntax. We will thereafter call `C--` the language defined in Figure 4.1 and denote as $\mathcal{V}$ the set of all potential variables. The description of Figure 4.1 omits some classical statements but makes precise some low-level features of the language. Note moreover that *int-types* are denoted by their signededness (**s** for signed integers, **u** for unsigned integers) and their length in bits, instead of `char` or `unsigned long`. This transformation is made before the analysis, and depends on the platform. Moreover, in order to simplify the presentation we will consider strings as arrays of **u8** (or **unsigned char**), results can be easily extended to arrays of **s8** (**signed char**).

Cells. Our `C`-like language features a rich type system. In a classic way, we will present the semantics of operations on scalar data-types: integers of various size (this can be easily extending to floats) and pointers, and reduce structured data-types, such as arrays, structs and unions, malloced blocks, to collections of scalar objects, we call *cells*. A simple solution would be to

$$
\begin{array}{lll}
\mathit{int\text{-}type} & \triangleq & \mathbf{s8} \mid \mathbf{s16} \mid \mathbf{s32} \mid \mathbf{s64} \\
 & & \mid \mathbf{u8} \mid \mathbf{u16} \mid \mathbf{u32} \mid \mathbf{u64} \\
\mathit{scalar\text{-}type} & \triangleq & \mathit{int\text{-}type} \mid \mathbf{ptr} \\
\mathit{type} & \triangleq & \mathit{scalar\text{-}type} \\
 & & \mid \mathit{type}[n] \quad n \in \mathbb{N} \\
 & & \mid \mathbf{struct}\{u_0 : \mathit{type}, \dots, u_{n-1} : \mathit{type}\} \\
 & & \mid \mathbf{union}\{u_0 : \mathit{type}, \dots, u_{n-1} : \mathit{type}\}
\end{array}
\qquad
\begin{array}{lll}
\mathit{lval} & \triangleq & *_{\mathit{scalar\text{-}type}}\mathit{expr} \mid v \in \mathcal{V} \\
\mathit{expr} & \triangleq & \mathtt{cst} \quad \mathtt{cst} \in \mathbb{N} \\
 & & \mid \&\mathit{lval} \\
 & & \mid \mathit{expr} \diamond \mathit{expr} \quad \diamond \in \{+, \leqslant, \dots\} \\
\mathit{stmt} & \triangleq & v = \mathtt{malloc}(e) \\
 & & \quad v \in \mathcal{V}, e \in \mathit{expr} \\
 & & \mid \mathit{type}\ v \quad v \in \mathcal{V} \\
 & & \mid \mathit{lval} = \mathit{expr} \\
 & & \mid \dots
\end{array}
$$

Figure 4.1: The syntax of the `C--` subset of `C`.

use the type of a structured variable and decompose it statically into such collections; writable memory zones (left-values) thus become access paths. Unfortunately, this static view does not hold for programs that abuse the type system and access some block of memory with various types, which is possible (and even common) in `C` using union types and pointer casts (consider for example Program 4.2). One solution would be to model the memory as arrays of bytes or even bits, and synthesize non-byte access (for instance, reading a 16-bit integer `a` would be expressed as `a[0]+256*a[1]`), but such a complex modeling would put a great strain on numeric abstract domains and cause huge precision losses. We thus rely on previous work [Min06a], that proposes to model memory blocks as collections of (possibly multi-byte) scalar cells, that are inferred and maintained dynamically during the analysis, according to the memory access pattern effectively employed by the program at run-time. For our purpose, we can assume that all memory accesses have the form $*_\tau e$, where τ is a scalar type and e is a pointer expression using pointer arithmetic at the byte level (this reduction can be performed statically as a pre-processing).

Remark 4.1. In addition to the definitions of Figure 4.1, we assume that we are given a function $\mathit{typeof} \in (\mathcal{V} \rightarrow \mathit{type})$. The type of a variable is given by its declaration in a `C--` program. Moreover we assume given a sizeof function from type to $\mathbb{N}$ that gives the size in bytes of each type (e.g. $\mathit{sizeof}(\mathbf{s32}) = 4$).

A cell denotes an addressable group of bytes to store a scalar value, it is represented by a base variable (v), an integer coding for the offset of the cell (o), and the type of the cell (t). Therefore we define the following set of cells:

$$\mathcal{C}\mathit{ell} \triangleq \{\langle v, o, t\rangle \mid v \in \mathcal{V},\ t \in \mathit{scalar\text{-}type},\ 0 \leqslant o \leqslant \mathit{sizeof}(\mathit{typeof}(v)) - \mathit{sizeof}(t)\}$$

By construction $\mathcal{C}\mathit{ell}$ represents the set of all addressable memory locations. The abstract states we will build contain a subset of those cells. Cells might denote overlapping portions of the memory. In such cases the underlying state satisfies every constraint implied by a cell: cells are understood conjunctively. Therefore removing cells induces a loss of information. A key aspect of [Min06a] we reuse is that new cells from $\mathcal{C}\mathit{ell}$ are added to the current environment dynamically to account for the access patterns encountered during the analysis, in a flow-sensitive way. As we do not rely on static type information, which can be misleading in C, we can handle union types, type-punning, and untyped allocated blocks transparently.

Concrete semantics. We will not detail here the complete concrete semantics of the `C--` language, however we give a definition of the set of concrete environments using cells, noted $\mathcal{E}$. An environment is a set of cells C and a function ρ mapping each cell to a value. A value can be either a numerical value or a pointer. A pointer is represented by: the base variable towards which it points and its offset. The set of pointer $\mathcal{P}\mathit{tr}$ is augmented with two special values: the

NULL pointer and the **invalid** pointer : $\mathcal{P}tr \triangleq (\mathcal{V} \times \mathbb{Z}) \cup \{\textbf{NULL}, \textbf{invalid}\}$

$$\mathcal{E} \triangleq \bigcup_{C \subseteq \mathcal{C}ell} \{\langle C, \rho \rangle \mid \rho \in R \triangleq C \to (\mathbb{N} \cup \mathcal{P}tr)\}$$

4.3 Background – Cell abstract domain

Let us consider the Cell abstraction [Min06a], an abstract domain able to abstract the semantics of C programs manipulating pointers. This abstraction associates to each cell with pointer type the set of variables it may point to; and associates to each cell a dimension in a numerical abstract domain. This dimension provides constraints on the content of cells with integer types, on the offset of the pointer for cells with pointer types. This abstraction is relational for numerical values (offsets of pointers and integer values), but is not relational for the variables pointed to by pointers. This abstract domain comes with an abstract interpreter that can successfully analyze C programs with no recursion and no dynamic memory allocation. The abstraction we propose here is built upon the cell abstract domain, it extends this domain so as to handle dynamic allocations and higher level string manipulations.

Pointers bases. When $C \subseteq \mathcal{C}ell$ is a subset of cells, we define $\overline{C}$ to be the set of cells denoting pointers : $\overline{C} \triangleq \{\langle v, o, t \rangle \in C \mid t = \textbf{ptr}\}$. Upon this we define $\mathcal{P}_C = \overline{C} \to \wp(\mathcal{V} \cup \{\textbf{NULL}, \textbf{invalid}\})$. $\mathcal{P}_C$ represents the possible memory locations pointed to by cells representing pointers (note that $\mathcal{P}_C$ only accounts for the base variable that is pointed to and not for the offset).

Numerical domain. We recall that for any finite set $\mathcal{W}$, we assume given a numerical abstract domain $\mathcal{N}^{\mathcal{W}}$. In addition to the hypothesis from Chapter 2, we assume that there exists a function $\textbf{range} \in \mathcal{W} \times \mathcal{N}^{\mathcal{W}} \to \overline{\mathbb{Z}}^2$ such that: for any $v \in \mathcal{W}$ and for any $N^\sharp \in \mathcal{N}^{\mathcal{W}}$, $\textbf{range}(x, N^\sharp)$ yields an interval of $\mathbb{Z}$ containing all concrete values associated to variable x in $N^\sharp$. In the interval domain this function simply returns the interval associated to the variable x, in the polyhedra domain, this returns the projection of the polyhedron onto the one dimensional space generated by variable x.

For any subset $C \subseteq \mathcal{C}ell$ we can therefore rely on a numerical abstraction $\mathcal{N}^C$ abstracting $\wp(C \to \mathbb{Z})$. We give the numerical domain of our abstraction a double role:

- For a cell containing a pointer, the variable (from the numerical domain) assigned to this cell codes for possible offsets of the pointer (thus paired with information from $\mathcal{P}_C$ we will describe completely the pointer contained in the cell)
- For other cells (containing e.g. a **u8**, a **s32**) the variable (from the numerical domain) codes for values contained in the cell.

Abstract states. We define the abstract domain $\mathcal{D}^\sharp_m$ with concretization $\gamma_m \in \mathcal{D}^\sharp_m \to \mathcal{E}$ as:

$$\mathcal{D}^\sharp_m \triangleq \{\langle C, N^\sharp, P \rangle \mid C \subseteq \mathcal{C}ell,\ N^\sharp \in \mathcal{N}^C,\ P \in \mathcal{P}_C\}$$

$$\gamma_m\langle C, N^\sharp, P \rangle \triangleq \langle C, \{\rho' \in R, \exists \rho \in \gamma^C(N^\sharp),\ \forall c = \langle v, o, t \rangle \in C, \left\{\begin{array}{ll} \rho'(c) = \rho(c) & \text{if } t \neq \textbf{ptr} \\ \rho'(c) = \langle p, \rho(c) \rangle & \text{if } t = \textbf{ptr} \wedge p \in P(c) \cap \mathcal{V} \\ \rho'(c) = p & \text{if } t = \textbf{ptr} \wedge p \in \{\textbf{NULL}, \textbf{invalid}\} \end{array}\right. \}\rangle$$

Example 4.1. Consider Program 4.2. At program point ● we have the abstract state: $S^\sharp = \langle \{\langle a, 0, \textbf{u64} \rangle\}, \{\langle a, 0, \textbf{u64} \rangle = 2^{32} + 2\}, \emptyset \rangle$. Moreover the next statement requires to read cells $\{\langle a, 0, \textbf{u32} \rangle\}$ and $\{\langle a, 4, \textbf{u32} \rangle\}$. $S^\sharp$ is equivalent to: $\langle \{\langle a, 0, \textbf{u64} \rangle, \langle a, 0, \textbf{u32} \rangle, \langle a, 4, \textbf{u32} \rangle\}, \{\langle a, 0, \textbf{u64} \rangle = 2^{32} + 2, \langle a, 0, \textbf{u32} \rangle = 2\ (= (2^{32} + 2) \mod 2^{32})), \langle a, 4, \textbf{u32} \rangle = 1\ (= (2^{32} + 2)\ / 2^{32})\}, \emptyset \rangle$.

```
unsigned long a = 0x100000002; ●
unsigned int x = *((unsigned int *) &a) + *((unsigned int *) &a + 1);
```

Figure 4.2: Pair of **u32** as **u64**

```
int a = 1;
int* p = &a;
*p = *p + 1;
```

Program 4.3: Dereferencing

Abstract operators and abstract transformers. Abstract operators (join, meet, widening) are defined by first unifying the operands, and then performing the operation in the underlying unified numerical domain and pointer map. The unification operator transforms two abstract elements into abstract elements with the same set of cells. This is done by by adding, in each element, the cells that exist only in the other element. We do not give here the definition of all the abstract transformers operating on our abstract states (we refer to [Min06a, Min13]), however the following example emphasizes how an abstract state is modified by expressions and statements of the `C` language. In particular, we note that when cells are available, most expressions are treated as expressions on a language where cells are the variables. When cells mentioned in the expressions are not available, they are added to the set of cells of the abstract state, by collecting information available in the overlapping cells, such as joining two byte-cells to synthesize the initial value of a new **u16**-cell at the same position or splitting a **u64** cell into two **u32** cells, as done in Example 4.1. The addition of cells required by the unification operator also uses this cell *materialization*, hence the set abstract operators also rely on cell additions.

Henceforth, in order to clarify the presentation, **a** denotes $\langle \mathsf{a}, 0, \tau \rangle$ when τ is the declared type of variable `a`.

Example 4.2. Consider Program 4.3, starting from $\top = \langle \emptyset, \emptyset, \emptyset \rangle$. The first statement requires the existence of the cell $\mathbf{a} = \langle \mathsf{a}, 0, \mathbf{s32} \rangle$. The set of cells constrained by our abstract state is dynamically updated to mention **a**, yielding: $\langle \{\mathbf{a}\}, \top^{\{\mathbf{a}\}}, \emptyset \rangle$, then we rewrite the statement in the following manner: `a = 1`. We execute this statement in the underlying numerical domain, and get: $\langle \{\mathbf{a}\}, \{\mathbf{a} = 1\}, \emptyset \rangle$. The second statement adds a new cell $\mathbf{p} = \langle \mathsf{p}, 0, \mathbf{ptr} \rangle$ and an element to the pointer map: $\langle \{\mathbf{a}, \mathbf{p}\}, \{\mathbf{a} = 1, \mathbf{p} = 0\}, \{\mathbf{p} \mapsto \{\mathsf{a}\}\} \rangle$. Note that $\mathbf{p} = 0$ codes for the value of the offset of pointer `p`. Finally the expression `*p` of the third statement is evaluated by following the P component of the abstract state, therefore the statement is transformed into `a = a + 1`. Thus yielding: $\langle \{\mathbf{a}, \mathbf{p}\}, \{\mathbf{a} = 2, \mathbf{p} = 0\}, \{\mathbf{p} \mapsto \{\mathsf{a}\}\} \rangle$.

4.4 String abstract domain

4.4.1 Domain definition

The introductory example shows that describing a string by a set of cells (one cell per character of the string) was usually not necessary to prove the absence of buffer overrun in string manipulations. Therefore we propose to add to our existing low-level abstraction of `C`-- an abstraction of strings that sums up all of its characters into two variables, one coding for the length of the string and the other for the allocated size of the buffer in which it is contained. We recall that the length of the string is defined to be the index of the first `'\0'` character, when there is no such character we define, by convention, the length of the string as the allocated size of the buffer containing the string. Memory blocks will therefore be abstracted either by the cell abstract domain or by the string abstract domain, whenever a memory block contains strings and integers (as is the case in a `struct`), we leave the task of representing this region to the cell abstract domain. In order to simplify the presentation we assume given a set of memory locations $\mathfrak{V}$ for which we will use a string summary, however this set can be dynamically modified and reductions could be proposed in order to store information on some memory locations in both the String domain and the Cell domain. We assume that for each memory location $s \in \mathfrak{V}$, we are given two variables denoted s_l and s_a. Those variables code for the length and the allocated size of the string, they will be added to the numerical domain of the cell abstract domain so that we are able to describe relations between lengths of variables and offsets of pointers. In the following $\mathfrak{V}^\star$ denotes $\bigcup_{s\in\mathfrak{V}}\{s_a, s_l\}$, this is the set of all numerical variables needed to describe strings in $\mathfrak{V}$.

Definition 4.1 (String abstract domain)**.** We define the String abstract domain to be:

$$\mathcal{S}^\sharp_m \triangleq \{\langle C, N^\sharp, P\rangle \mid C \subseteq \mathcal{C}ell \setminus \{\langle v, \_, \_\rangle \mid v \in \mathfrak{V}\},\ N^\sharp \in \mathcal{N}^{C\cup\mathfrak{V}^\star},\ P \in \mathcal{P}_C\}$$

This definition is the same as the cell abstract domain, with the difference that no cells are allocated for strings. However two numerical variables are added to the numerical domain for each string. These two variables encode the length and allocated size of the string.

Definition 4.2 (Order relation)**.** The string abstract domain is ordered by the same relation as the cell abstract domain: $\sqsubseteq_{S^\sharp_m} \triangleq \sqsubseteq_{D^\sharp_m}$.

We recall that $\sqsubseteq_{\mathcal{D}_m}$ will test the inclusion of the two numerical domains once cell sets have been unified, therefore our definition of $S^\sharp \sqsubseteq_{S^\sharp_m} S^{\sharp\prime}$ amounts to verifying that the constraints on the string variables (s_l and s_a) are stronger in the left member of the inequality.

4.4.2 Galois connection with the Cell abstract domain

The String abstract domain is an abstraction of the Cell abstract domain. Indeed we forget information that do not help us track the position of the first `'\0'` character. We define the Galois connection between the Cell domain and the String domain using two functions : **to_cell** and **from_cell**. The **to_cell**$(s, S^\sharp)$ function computes the range of s_l in the numeric abstract domain, for each possible length value we set the cells placed before (resp. at) the length to $[1; 255]$ (resp. 0), this yields an abstract element per possible value in the range, those are then joined. Conversely **from_cell**$(s, S^\sharp)$ computes the minimum length value (the index of the first cell whose range contains 0), and the maximum length value (the index of the first cell whose range is exactly $\{0\}$, it is the size of the buffer if no such index is found), finally those constraints are added to the numerical domain. If a string does not contain any `'\0'` character, we define its length to be the allocated size of the buffer it is contained in. The cell abstract domain provides an **add_cells**, which given a set of cells and an abstract element adds the set of cells to the abstract element, by collecting potential constraints.

Definition 4.3 (**to_cell** and **from_cell**)**.** In the following *st* denotes *sizeof* ∘ *typeof*.

$$
\begin{aligned}
&\textbf{to\_cell}(s, S^\sharp) = \\
&\qquad \textbf{let } \langle C, N^\sharp, P\rangle = \textbf{add\_cells}(\{\langle s, 0, \mathbf{u8}\rangle, \dots, \langle s, \mathit{st}(s) - 1, \mathbf{u8}\rangle\}, S^\sharp) \textbf{ in} \\
&\qquad \textbf{let } [a; b] = \textbf{range}(s_l, N^\sharp) \cap [0; \mathit{st}(s) - 1] \textbf{ in} \\
&\qquad \textbf{let } (N^\sharp_i)_{i \in [a;b]} = \mathbb{S}^\sharp\llbracket \texttt{remove}(s_l)\rrbracket \circ \mathbb{S}^\sharp\llbracket \texttt{remove}(s_a)\rrbracket \circ \\
&\qquad\qquad\qquad \mathbb{S}^\sharp\llbracket \texttt{Assume}(\{\langle s, 0, \mathbf{u8}\rangle \neq 0, \quad \dots, \langle s, i-1, \mathbf{u8}\rangle \neq 0, \langle s, i, \mathbf{u8}\rangle = 0\})\rrbracket(N^\sharp) \textbf{ in} \\
&\qquad \bigsqcup_{j=0}^{b} \langle C', N^\sharp_j, P\rangle \\
&\textbf{from\_cell}(s, S^\sharp) = \\
&\qquad \textbf{let } \langle C, N^\sharp, P\rangle = \textbf{add\_cells}(\{\langle s, 0, \mathbf{u8}\rangle, \dots, \langle s, \mathit{st}(s) - 1, \mathbf{u8}\rangle\}, S^\sharp) \textbf{ in} \\
&\qquad \textbf{let } c_{\geqslant} = \min(\{i \mid 0 \in \textbf{range}(\langle s, i, \mathbf{u8}\rangle, N^\sharp)\} \cup \{\mathit{st}(s)\}) \textbf{ in} \\
&\qquad \textbf{let } c_{\leqslant} = \min(\{i \mid \{0\} = \textbf{range}(\langle s, i, \mathbf{u8}\rangle, N^\sharp)\} \cup \{\mathit{st}(s)\}) \textbf{ in} \\
&\qquad \textbf{let } C^\star = \{\langle s', i, \tau\rangle \in C \mid s \neq s'\} \textbf{ in} \\
&\qquad \textbf{let } N^{\sharp\star} = \mathbb{S}^\sharp\llbracket \texttt{Assume}(\{s_l \geqslant c_{\geqslant}, s_l \leqslant c_{\leqslant}, s_a = \mathit{st}(s)\})\rrbracket(N^\sharp) \textbf{ in} \\
&\qquad \langle C^\star, N^{\sharp\star}, P\rangle
\end{aligned}
$$

Definition 4.4 (Galois connection)**.** With $\mathfrak{V} = \{s_0, \dots, s_{n-1}\}$, we can define:

$$
\begin{aligned}
\gamma_{S^\sharp_m, D^\sharp_m}(S^\sharp) &= \textbf{to\_cell}(s_0, \dots, (\textbf{to\_cell}(s_{n-1}, S^\sharp)) \dots) \\
\alpha_{S^\sharp_m, D^\sharp_m}(S^\sharp) &= \textbf{from\_cell}(s_0, \dots, (\textbf{from\_cell}(s_{n-1}, S^\sharp)) \dots)
\end{aligned}
$$

Example 4.3. Consider the string abstract elements $\langle \emptyset, \{s_l = 2, s_a = 4\}, \emptyset\rangle$ when *sizeof*(*type*(s)) = 4. We have: $\gamma_{S^\sharp_m, D^\sharp_m} = \langle \{\langle s, 0, \mathbf{u8}\rangle, \langle s, 1, \mathbf{u8}\rangle, \langle s, 2, \mathbf{u8}\rangle\}, \{\langle s, 2, \mathbf{u8}\rangle = 0, \langle s, 0, \mathbf{u8}\rangle \neq 0, \langle s, 1, \mathbf{u8}\rangle \neq 0\}, \emptyset\}\rangle$. The corresponding concrete state is the set of states in which there is a memory location where the first two bytes are non zero bytes, the third byte is set to zero and the fourth byte is unconstrained.

Remark 4.2. The interest of the definition of $\gamma_{S^\sharp_m, D^\sharp_m}$ and $\alpha_{S^\sharp_m, D^\sharp_m}$ is twofold: it enables us to define the semantics of the String abstract domain, but we also note that both functions **to_cell** and **from_cell** are computable. Therefore the set of memory locations dealt with by each domain can easily evolve during the analysis. Moreover with $\gamma_{S^\sharp_m, D^\sharp_m}$ and $\alpha_{S^\sharp_m, D^\sharp_m}$ being both computable, we can define a reduction operator between the String abstract domain and the Cell abstract domain. For efficiency reasons we can remove some information from the Cell domain, knowing that information from the String domain can be brought back to the Cell domain. This situation is similar to a reduction between octagons and the, strictly less expressive, interval domain as proposed in [CCF+06b].

Now that we have defined the String abstract domain, let us define the operators and transformers on elements of this domain.

4.4.3 Operators

Definition 4.5 ($\sqcup_{S^\sharp_m}$, $\sqcap_{S^\sharp_m}$ and $\triangledown_{S^\sharp_m}$)**.** As for the definition of the $\sqsubseteq_{S^\sharp_m}$ operator, the join ($\sqcup_{S^\sharp_m}$), meet ($\sqcap_{S^\sharp_m}$) and widening ($\triangledown_{S^\sharp_m}$) of two abstract elements is defined by applying the according operator in the underlying numerical abstract domain (after the addition on both sides of potentially missing variables and the unification of the set of cells).

Example 4.4. Consider Program 4.1 of the introductory example where *typeof*$(\mathtt{p}) =$ *typeof*$(\mathtt{q}) =$ **ptr**, if $S_1^\sharp = \langle\{\mathbf{p},\mathbf{q}\},\{\mathbf{p}=0,\mathbf{q}=0,\mathtt{src}_l \geqslant 0,\mathtt{src}_a \geqslant \mathtt{src}_l,\mathtt{dest}_l \geqslant 0,\mathtt{dest}_a \geqslant \mathtt{dest}_l\},\{\mathbf{p}\mapsto\{\mathtt{dest}\},\mathbf{q}\mapsto\{\mathtt{src}\}\}\rangle$ is the abstract state from which we start the analysis then $S_2^\sharp = \langle\{\mathbf{p},\mathbf{q}\},\{\mathbf{p}=1,\mathbf{q}=1,\mathtt{src}_l \geqslant 1,\mathtt{src}_a \geqslant \mathtt{dest}_l,\mathtt{dest}_l \geqslant 1,\mathtt{dest}_a \geqslant 1,\mathtt{dest}_a \geqslant \mathtt{dest}_l\},\{\mathbf{p}\mapsto\{\mathtt{dest}\},\mathbf{q}\mapsto\{\mathtt{src}\}\}\rangle$ is the abstract state after one analysis of the body of the `while` loop (constraint $\mathtt{src}_a \geqslant \mathtt{dest}_l$ comes from the fact that we collect error free executions). Therefore our analyzer has to perform the join of those two abstract states before reanalyzing the body of the loop. $S_1^\sharp \sqcup_{S_m^\sharp} S_2^\sharp = \langle\{\mathbf{p},\mathbf{q}\},\{\mathbf{p}=\mathbf{q},\mathbf{p}\leqslant 1,\mathbf{p}\geqslant 0,\mathbf{p}\leqslant\mathtt{dest}_l,\mathtt{dest}_a\geqslant\mathtt{dest}_l,\mathbf{p}\leqslant\mathtt{src}_l,\mathbf{p}\leqslant\mathtt{src}_a\},\{\mathbf{p}\mapsto\{\mathtt{dest}\},\mathbf{q}\mapsto\{\mathtt{src}\}\}\rangle$. Applying a widening at first loop iteration would yield: $S_1^\sharp \sqcup_{S_m^\sharp} S_2^\sharp = \langle\{\mathbf{p},\mathbf{q}\},\{\mathbf{p}=\mathbf{q},\mathbf{p}\geqslant 0,\mathbf{p}\leqslant\mathtt{dest}_l,\mathtt{dest}_a\geqslant\mathtt{dest}_l,\mathbf{p}\leqslant\mathtt{src}_l,\mathbf{p}\leqslant\mathtt{src}_a\},\{\mathbf{p}\mapsto\{\mathtt{dest}\},\mathbf{q}\mapsto\{\mathtt{src}\}\}\rangle$

The state transformations induced on our abstract state by the semantics of the C language is mainly dealt with by the Cell abstraction.

Definition 4.6 ($\mathtt{@}[v,e]$)**.** To ease the presentation of the transformers[1], we add expressions of the form $\mathtt{@}[v,e]$ with $e \in$ *expr* and $v \in \mathcal{V}$ to the C-- language. Such expressions denote pointers to variable v, with offset e:

$$\mathtt{@}[v,e] \triangleq \&v + e$$

Example 4.5. We want to perform the analysis of the statement

$$\mathtt{stmt} \triangleq \mathtt{*}_{\mathtt{u8}}\mathtt{t\ =\ *}_{\mathtt{u8}}\mathtt{(p\ +\ *}_{\mathtt{s32}}\mathtt{(\&u\ +\ 2))}$$

(where *typeof*$(\mathtt{p}) =$ *typeof*$(\mathtt{t}) =$ **ptr**) in the following abstract state: $S^\sharp = \langle\{\mathbf{p},\langle \mathrm{u},2,\mathbf{s32}\rangle,\mathbf{t}\},N^\sharp,\{\mathbf{p}\mapsto s',\mathbf{t}\mapsto s\}\rangle$ where $N^\sharp$ is a numerical abstract state built from the set of constraints we do not need to explicit for this example. The Cell abstraction rewrites `stmt` into:

$$\begin{aligned}
\mathtt{stmt} &\rightsquigarrow \mathtt{*}_{\mathtt{u8}}\mathtt{@}[s,o_t] = \mathtt{*}_{\mathtt{u8}}(\mathtt{@}[s',o_p] + \mathtt{*}_{\mathtt{u32}}(\mathtt{@}[u,0] + 2))\\
&\rightsquigarrow \mathtt{*}_{\mathtt{u8}}\mathtt{@}[s,o_t] = \mathtt{*}_{\mathtt{u8}}(\mathtt{@}[s',o_p] + \mathtt{*}_{\mathtt{u32}}\mathtt{@}[u,2])\\
&\rightsquigarrow \mathtt{*}_{\mathtt{u8}}\mathtt{@}[s,o_t] = \mathtt{*}_{\mathtt{u8}}(\mathtt{@}[s',o_p + \mathtt{*}_{\mathtt{u32}}\mathtt{@}[u,2]])\\
&\rightsquigarrow \mathtt{*}_{\mathtt{u8}}\mathtt{@}[s,o_t] = \mathtt{*}_{\mathtt{u8}}(\mathtt{@}[s',o_p + \langle \mathrm{u},2,\mathtt{u32}\rangle]).
\end{aligned}$$

This example emphasizes that there are only two transformers that need to be defined, the rest being delegated to the underlying cell domain:

$$\begin{array}{ll}
\mathbb{S}^\sharp[\![\mathtt{*}_\tau\mathtt{@}[s,e_1] = e_2]\!](S^\sharp) & \text{where } s \in \mathfrak{V}, e_1 \in \textit{expr}, e_2 \in \textit{expr}, \tau \in \textit{scalar-type}\\
\mathbb{E}^\sharp[\![\mathtt{*}_\tau\mathtt{@}[s,e]]\!](S^\sharp) & \text{where } s \in \mathfrak{V}, e \in \textit{expr}, \tau \in \textit{scalar-type}
\end{array}$$

4.4.4 Abstract evaluation

Let us first consider the evaluation of the dereferencing of a pointer to a string. The analyzer we want to define performs partitioning on the abstract state during the evaluation of expressions, therefore evaluation results are pairs (evaluated expression × abstract state): $\mathbb{E}^\sharp[\![.]\!] \in S_m^\sharp \to \wp(\textit{exp} \times S_m^\sharp)$. This return set is understood disjunctively and greatly improves the precision of the analyzer. We recall from Chapter 3 that mopsa handles such evaluations. Five cases can be distinguished during the evaluation of $\mathtt{*}_\tau\mathtt{@}[s,e]$:

- **before**: $\tau =$ **u8** and $\mathtt{@}[s,e]$ points before the first `'\0'` character. In this case the evaluation can yield any character that is not `'\0'`.
- **at**: $\tau =$ **u8** and $\mathtt{@}[s,e]$ points at the first `'\0'` character. In this case the evaluation yields `'\0'`.

[1] and actually to ease the implementation as well.

function	tests on offset	evaluation
before	$0 \leqslant o \wedge o < l \wedge o < a$	$[1;255]$
at	$0 \leqslant o \wedge o = l \wedge o < a$	0
after	$0 \leqslant o \wedge o > l \wedge o < a$	$[0;255]$
eerror	$o + r > a \vee o < 0$	$\emptyset$

Figure 4.3: Evaluation of a dereferencing.

- **after**: $\tau =$ **u8** and `@[s,e]` points after the first `'\0'` character. In this case the evaluation can yield any character.
- **eerror**: `@[s,e]` points outside of the allocated memory. In such a case we generate an **out_of_bounds** error.
- $\tau \neq$ **u8**, in this case we over-approximate the evaluation by the range of the type τ.

Figure 4.3 summarizes these cases. In this table o is the offset of the pointer, l and a are the length and the allocated size of the string. The following definition of the **before** operator underlines how Figure 4.3 should be read.

$$\mathbf{before}(e, s, \langle C, N^\sharp, P\rangle) = \\ \{([1;255], \langle C, \mathbb{S}^\sharp[\![\mathtt{Assume}(\{0 \leqslant e, e < s_l, e < s_a\})]\!](N^\sharp), P\rangle\}$$

Definition 4.7 ($\mathbb{E}^\sharp[\![*_\tau\mathtt{@[s,e]}]\!](S^\sharp)$)**.** Using these functions, we can now define:

$$\mathbb{E}^\sharp[\![*_{\tau=\mathtt{u8}}\mathtt{@[s,e]}]\!](S^\sharp) = \bigcup_{(e',S^\sharp)\in\mathbb{E}^\sharp[\![e]\!]S^\sharp} (\mathbf{before}(e', s, S^\sharp) \cup \mathbf{at}(e', s, S^\sharp) \\ \cup\, \mathbf{after}(e', s, S^\sharp) \cup \mathbf{eerror}(e', 1, s, S^\sharp))$$

and

$$\mathbb{E}^\sharp[\![*_{\tau\neq\mathtt{u8}}\mathtt{@[s,e]}]\!](S^\sharp) = \{(\mathit{range}(\tau), S^\sharp)\} \cup \mathbf{eerror}(e', \mathit{st}(\tau), s, S^\sharp))$$

Example 4.6. Consider $S^\sharp = \langle\{\langle \mathsf{p}, 0, \mathbf{ptr}\rangle(\triangleq \mathbf{p})\}, \{\mathbf{p} \leqslant s_l, s_l < s_a\}, (\mathbf{p} \mapsto s)\rangle$. As $\mathbf{p}$ may point at, or before the first `'\0'` we get that:

$$\mathbb{E}^\sharp[\![*_{u8}\mathsf{p}]\!](S^\sharp) = \{(0, \langle\{\mathbf{p}\}, \{\mathbf{p} = s_l, s_l < s_a\}, (\mathbf{p} \mapsto s)\rangle), \\ ([1,255], \langle\{\mathbf{p}\}, \{\mathbf{p} < s_l, s_l < s_a\}, (\mathbf{p} \mapsto s)\rangle) \\ \}$$

4.4.5 Abstract transformations

In order to complete the definition of our abstract interpreter, we need to provide the abstract semantics of an assignment in a string $*_\tau$`@[s,`e_1`] = `e_2. We can distinguish 7 cases in such an assignment:

- **set0**: $\tau =$ **u8** and a character that appears before the first `'\0'` is assigned to `'\0'`, in which case we need to set the variable coding for the length of the string to its new value (the offset of the pointer to the string). This case is dealt with by the **set0** function.
- **setnon0**: $\tau =$ **u8** and the first `'\0'` is replaced with a non-`'\0'` character, in which case we need to set the variable coding for the length of the string to its new value (it can be anything greater than the offset of the pointer to the string). This case is dealt with by the **setnon0** function.

function	tests on offsets	tests on rhs	transformation
set0	$o \geqslant 0 \wedge o \leqslant l \wedge o < a$	$c = 0$	$l \leftarrow o$
setnon0	$o \geqslant 0 \wedge o = l \wedge o < a$	$c \neq 0$	$l \leftarrow [o+1; a]$
unchanged	$o \geqslant 0 \wedge o < l \wedge o < a$	$c \neq 0$	
unchanged	$o \geqslant 0 \wedge o > l \wedge o < a$	$\top$	
l_unchanged	$o \geqslant 0 \wedge o > l \wedge o + r \leqslant a$	$\top$	
forget	$o \geqslant 0 \wedge o \leqslant l \wedge o + r \leqslant a$	$\top$	$l \leftarrow [o; a]$
serror	$o + r > a \vee o < 0$	$\top$	**out_of_bounds**

Figure 4.4: Summary of transformations.

- **unchanged**: $\tau =$ **u8** and we are performing an assignment that does not change the position of the first `'\0'` character. Either because we are replacing a character placed before the first `'\0'` character by a non-`'\0'` character, or because we are assigning a character after the position of the first `'\0'` character. Both cases are dealt with by the **unchanged** function.
- **l_unchanged**: $\tau \neq$ **u8** and we are performing an assignment that does not change the position of the first `'\0'` character: the only modified characters are placed after the first `'\0'` character. This case is dealt with by the **l_unchanged** function.
- **forget**: $\tau \neq$ **u8** and the offset of the pointer is less than the length of the string, in this case the position of the first `'\0'` character is greater than the offset of the pointer. This case is dealt with by the **forget** function.
- **serror**: The write generates an out of bounds, in which cases we generate an **out_of_bounds** warning. This case is dealt with by the **serror** function.

Figure 4.4 summarizes these cases. In this table l and a denote respectively the length and the allocated size of string s, o denotes the offset of the pointer, c denotes the evaluated right-hand side of the assignment, and finally r denotes $\mathit{sizeof}(\tau)$. The following definition of operator **set0** underlines how Figure 4.4 should be read.

$$\begin{aligned}
&\mathbf{set0}(s, e_1, e_2, \langle C, N^\sharp, P\rangle) = \\
&\quad \mathbf{let}\ N_1^\sharp = \mathbb{S}^\sharp[\![\mathtt{Assume}(\{e_1 \geqslant 0, e_1 \leqslant s_l, e_1 < s_a, e_2 = 0\})]\!](N^\sharp)\ \mathbf{in} \\
&\quad \mathbf{let}\ N_2^\sharp = \mathbb{S}^\sharp[\![s_l \leftarrow e_1]\!](N_1^\sharp)\ \mathbf{in} \\
&\quad \langle C, N_2^\sharp, P\rangle
\end{aligned}$$

Definition 4.8 ($\mathbb{S}^\sharp[\![*_\tau\mathtt{@[}s \in \mathfrak{V}, e_1\mathtt{]=}e_2]\!](S^\sharp)$)**.** Using the 6 transformations aforementioned we can now define:

$$\begin{aligned}
\mathbb{S}^\sharp[\![*_{\tau=\mathtt{u8}}\mathtt{@[}s \in \mathfrak{V}, e_1\mathtt{]=}e_2]\!](S^\sharp) = \bigsqcup_{\substack{(e_1', S^{\sharp\prime}) \in \mathbb{E}^\sharp[\![e_1]\!](S^\sharp) \\ (e_2', S^{\sharp\prime\prime}) \in \mathbb{E}^\sharp[\![e_2]\!](S^{\sharp\prime})}} & \mathbf{set0}(s, e_1', e_2', S^{\sharp\prime\prime}) \sqcup \mathbf{serror}(s, e_1', 1, S^{\sharp\prime\prime}) \\
& \sqcup \mathbf{unchanged}(s, e_1', e_2', S^{\sharp\prime\prime}) \\
& \sqcup \mathbf{setnon0}(s, e_1', e_2', S^{\sharp\prime\prime})
\end{aligned}$$

and

$$\mathbb{S}^\sharp[\![\star_{\tau\neq\texttt{u8}}\texttt{@[}s \in \mathfrak{V}\texttt{,}e_1\texttt{]=}e_2]\!](S^\sharp) = \bigsqcup_{(e_1',S^{\sharp\prime})\in\mathbb{E}^\sharp[\![e_1]\!](S^\sharp)} \mathbf{1\_unchanged}(s, e_1', \mathit{sizeof}(\tau), S^{\sharp\prime}) \\ \sqcup\, \mathbf{forget}(s, e_1', \mathit{sizeof}(\tau), S^{\sharp\prime}) \\ \sqcup\, \mathbf{serror}(s, e_1', \mathit{sizeof}(\tau), S^{\sharp\prime})$$

Example 4.7. Going back to Example 4.5, we now assume that $N^\sharp$ is a numerical abstract state built from the constraint set: $\{\mathbf{t} < s_a, \mathbf{t} = s_l, \mathbf{p} + \langle \mathsf{u}, 2, \mathbf{s32}\rangle < s_l', s_l' < s_a'\}$. Moreover in the following $S^\sharp \wedge E$ denotes the abstract state $S^\sharp$ in which the numerical component has been extended with the constraints set E, and e denotes the expression $\mathbf{p} + \langle \mathsf{u}, 2, \mathbf{s32}\rangle$. $\mathbb{E}^\sharp[\![\texttt{*(@[}s'\texttt{,}e\texttt{])}]\!](S^\sharp) = \{([1;255], S^\sharp \wedge \{e \geqslant 0, e < s_l', e < s_a'\}), (0, S^\sharp \wedge \{e \geqslant 0, e = s_l', e < s_a'\}), ([0;255], S^\sharp \wedge \{e \geqslant 0, e > s_l', e < s_a'\})\}$. With a precise enough numerical domain (e.g. polyhedra), $S^\sharp \wedge \{e \geqslant 0, e = s_l', e < s_a'\}$, $S^\sharp \wedge \{e \geqslant 0, e > s_l', e < s_a'\}$ and $S^\sharp \wedge \{e < 0 \vee e \geqslant s_a'\}$ form empty partitions, meaning that in this example, they represent impossible cases. For similar reasons $\mathbb{S}^\sharp[\![\texttt{stmt}]\!](S^\sharp)$ will compute abstract elements that are reduced to $\perp$ for functions **set0**, **unchanged** and **serror**. Therefore: $\mathbb{S}^\sharp[\![\texttt{stmt}]\!](S^\sharp) = \langle\{\mathbf{p}, \langle \mathsf{u}, 2, \mathbf{s32}\rangle, \mathbf{t}\}, R^{\sharp\prime}, \{\mathbf{p} \mapsto s', \mathbf{t} \mapsto s\}\rangle$ with $R^{\sharp\prime} = \{\mathbf{t} < s_a, \mathbf{t} + 1 \leqslant s_l, \mathbf{p} + \langle \mathsf{u}, 2, \mathbf{s32}\rangle < s_l', s_l' < s_a'\}$. This assignment made our abstraction lose the position of the first `'\0'` character, as it wrote a non-`'\0'` character in its place.

4.4.6 String declaration

When encountering a local variable declaration (`u8 s[n]` with $n \in \mathbb{N}$ and $s \in \mathfrak{V}$) we can set the allocated size of the string to n, and set the length to the range $[0, n]$ as shown in the following example: $\mathbb{S}^\sharp[\![\texttt{u8 s[27]}]\!]\langle\emptyset, \emptyset, \emptyset\rangle = \langle\emptyset, \{s_l \geqslant 0, s_l \leqslant 27, s_a = 27\}, \emptyset\}\rangle$. The formal definition is straightforward:

$$\mathbb{S}^\sharp[\![\texttt{u8 s}\in \mathfrak{V}\texttt{[}n \in \mathbb{N}\texttt{]}]\!]\langle C, N^\sharp, P\rangle = \langle C, \mathbb{S}^\sharp[\![s_l \leftarrow [0, n]]\!] \circ \mathbb{S}^\sharp[\![s_a \leftarrow n]\!](N^\sharp), P\rangle$$

Example 4.8. $\mathbb{S}^\sharp[\![\texttt{u8 s[27]}]\!]\langle\emptyset, \emptyset, \emptyset\rangle = \langle\emptyset, \{s_l \geqslant 0, s_l \leqslant 27, s_a = 27\}, \emptyset\rangle$

Example 4.9. Consider again Program 4.1 from the introductory example, analyzed starting from an abstract state $S^\sharp = \langle\{\mathbf{p}, \mathbf{q}\}, \{\mathbf{p} = 0, \mathbf{q} = 0, 0 \leqslant s_l < s_a, 0 \leqslant s_l' < s_a'\}, \{\mathbf{p} \mapsto s, \mathbf{q} \mapsto s'\}\rangle$. Note that the input state contains the information that **p** and **q** do not alias. The numerical invariant (the rest of the abstract state is not modified by the analysis) found at the beginning of line 2 is: $\{-\mathbf{p} + \mathbf{q} = 0, s_l' \geqslant \mathbf{p} + 1, s_a' - s_l' \geqslant 0, \mathbf{p} \geqslant 0, s_l \geqslant \mathbf{p}\}$. An **out_of_bounds** error is generated at line 3, indeed in the starting abstract state, no hypothesis is made on the relation between s_l' and s_a therefore there might be a buffer overrun at line 3. Finally the numerical invariant discovered at the end of line 6 is: $\{s_l' = s_l, \mathbf{q} = s_l, \mathbf{p} = s_l, s_a' \geqslant s_l + 1, s_l \geqslant 0, s_a \geqslant s_l + 1\}$, thus showing that we were able to infer that the two strings pointed to by p and q have the same size at the end of the analysis.

Remark 4.3. When a string gets out of the scope, its allocated numerical variables are removed from the numerical domain. The handling of pointers pointing towards the string, at the time it gets out of scope, is handled, as for other memory zones, by the C analyzer upon which we built the string abstraction.

4.4.7 Dynamic memory allocation

As mentioned in Figure 4.1, we allow dynamic memory allocations. The Cell abstract domain as presented in Section 4.3 is not able to handle those. To model dyamic memory allocation, we consider a finite set $\mathcal{A}$ of heap addresses, derived from the allocation site using recency

```
void aux1(char** x,int e) {
  ●*x = malloc(e);
}
void aux2(char** x,int e) {
  ⬟*x = malloc(e);
}
int main() {
  char* x;
  aux1(&x,10); aux1(&x,20);
  aux1(&x,30); aux2(&x,40);
  *x = '\0';
}
```

Program 4.4: Dynamic memory allocations

abstraction [BR06]: for each allocation site $\mathfrak{a}$, one abstract address, $\mathfrak{a}^s$, is used to model the last block allocated at $\mathfrak{a}$, and another one, $\mathfrak{a}^w$, to summarize the blocks allocated previously at $\mathfrak{a}$. While we perform weak updates on the later, we can perform strong updates on the former, which ensures a gain in precision. Indeed, in C allocated block are uninitialized, their value must therefore be set to $\top$. If an uninitialized block is merged with all previously allocated blocks we have to modify it via weak updates, therefore the $\top$ value will never be modified. To solve this, recency abstraction keeps separated the last allocated block from the others, allowing strong updates on the newly allocated memory zone.

Example 4.10. Consider now Program 4.4, and assume that $\mathfrak{a}^s$ and $\mathfrak{a}^w$ (resp. $\mathfrak{b}^s$ and $\mathfrak{b}^w$) are addresses for which we perform strong and weak update at program point ● (resp. ⬟). Starting from $\top$ the analysis of the body of function `main`, we get: $\langle\{\mathbf{x}\}, \{0 \leqslant \mathfrak{a}^w_{\mathfrak{l}} \leqslant \mathfrak{a}^w_{\mathfrak{a}}, 10 \leqslant \mathfrak{a}^w_{\mathfrak{a}} \leqslant 20, 0 \leqslant \mathfrak{a}^s_{\mathfrak{l}} \leqslant \mathfrak{a}^s_{\mathfrak{a}}, \mathfrak{a}^s_{\mathfrak{a}} = 30, \mathfrak{b}^s_{\mathfrak{a}} = 40, \mathfrak{b}^s_{\mathfrak{l}} = 0, \mathbf{x} = 0\}, \{\mathbf{x} \mapsto \mathfrak{b}^s\}\rangle$ This state gives us that `x` points to a memory location starting from a `'\0'` character. We also note that information about the two first allocations made at program point $\mathfrak{a}$ have been collapsed into the $\mathfrak{a}^w$ address.

Experimental results. The abstraction presented in this chapter was implemented in Mopsa and used in a C analyzer. This abstraction was coupled with the modular analysis presented in the following chapter (Chapter 5). For this reason, experimental results on the String abstraction will be provided after the presentation of the modular analyzer from Chapter 5.

4.5 Related works

One popular technique to avoid buffer overflows is dynamic analysis. There is a long history of such techniques (see [WK03] for some examples). These methods induce an overhead cost and do not prevent program failures. By contrast we employ static analysis. Strings are arrays of characters, therefore analysis methods proposed in [CCL11] and [Cou03] could be used to design static analyzers handling strings, here however we rather tend to transfer string accesses to pointer manipulations in order to use the expressiveness of the Cell abstraction. The three following works are the closest to ours and all follow the idea introduced in [WFBA00] to track the length of strings. Dor et al. [DRS01] tackled the problem by rewriting string manipulating statements into statements over a numerical variable language, however this transformation induced the usage of a number of variables quadratic in the number of strings present in the analysis, in order to account for pointer aliasing, our predicates give constraints on memory zones and not on pointers, removing the need for pointer aliasing handling. Simon and King [SK02] proposed an analyzer for a sub-`C` language manipulating strings, and allowing dynamic memory allocation, but some pointer manipulations could not be handled. They improved their results in [Sim08], the string domain presented here is a combination of results from [Sim08] and the cell abstract domain, moreover we provided a way to dynamically balance strings dealt with by the string abstract domain and by the cell domain. Additionally string length and allocated size are bound to pointers (whereas we bind them to the actual memory location containing the string), and this approach seems to prevent the modular integration of this domain in a full `C` language analyzer. Allamigeon et al. [AGH06] also proposed an analyzer that keeps track of the position of the first `'\0'` character, however their analysis is non-relational, can not handle arbitrary pointer cast, and uses static information on string length, therefore preventing the domain reusability for dynamically allocated strings. Note moreover that both [AGH06] and [DRS01] defined special abstract transformations for `strcpy`, whereas we provide an abstraction that is able to precisely analyze different versions of `strcpy` .

4.6 Conclusion

To conclude this chapter, let us discuss two of the future works on the string abstract domain. We assumed given a static partitioning between the memory regions that should be handled by the string abstraction and those that should be handled by the cell abstract domain. This can be improved in two ways. Firstly we could choose dynamically which abstraction should handle which memory regions. Secondly we could use both abstractions and design a reduction operator between them so that information from each domain can be used by the other. These two improvements can be made using the **to_cell** and **from_cell** functions defined here.

Moreover, the string abstract domain can only handle precisely strings that are stored in their own memory region. Indeed if two strings are inlined in a `struct` the abstraction will fail to precisely represent the length of the second one. Our abstraction should be extended in future works so as to be able to handle such cases.

We proposed an abstract domain able to tackle `C` strings of parametric size, built as an add-on to an existing domain [Min06a] capable of dealing with most of the features of the `C` language. We have shown how our analyzer can be tuned dynamically by choosing whether the String or Cell domain should deal with certain memory regions or by changing the underlying numerical domain, so as to adjust its precision (even though we did not provide heuristics as to when one should be used instead of the other).

We are now given a full C abstract interpreter augmented with a predicate abstraction targeting out-of-memory accesses in C strings. We will show, in the next chapter, how this analyzer is lifted to a modular analysis framework.

Chapter 5

Modular analysis

5.1 Introduction

5.1.1 Motivations

In Chapter 4 we designed a predicate abstraction enabling a high level reasoning about C strings. Our goal in this chapter is to lift this string abstraction to a modular framework. Consider again the `strcpy` function in `C`, shown in Program 4.1. In a real life `C` project, this function is often user-redefined (as shown by the `strcpy` function from Qmail in Program 4.2) and called several times during the analysis. We will call *summary* of a statements a description of its functional semantics that can be applied on an input. The implementation of a modular static analyzer able to infer, prove and reuse summaries on such functions without losing precision would yield a scalable analyzer able to prove the absence of buffer overruns in `C` projects manipulating strings. Indeed, an analyzer computing invariants by induction on the syntax of programs requires abstract transformers for function calls. A straightforward way to achieve this, provided that there is no recursivity, is to analyze the body of the function at each call site. Therefore a way to improve scalability is to design modular analyzers able to reuse previous analysis results so that reanalysis is not always needed (as emphasized in [CC02]). As an example, in a project containing an incrementation function, we want to be able to express and to infer that $\forall x, \texttt{incr}(x) = x + 1$. Once this relation discovered, no further analysis of the body of `incr` is required. This technique can be applied on any statement, not just on the bodies of functions (e.g. in [JM18] this is applied to `while` statements when the number of nested loops is high).

When performing modular analysis two classical families of analyses can be distinguished: top-down analyses and bottom-up analyses.

- A top-down approach starts from the root of the program, mimicking a classical intra-procedural analysis by inlining. However for some statements in the program, summaries are created once the first analysis of this statement is finished. Such summaries will usually use as preconditions the constraints satisfied by the input state encountered during the top-down analysis. This has the advantage that summaries target only the useful part of the abstract semantics as we have information on the context (e.g. the state at call site in the case of function summaries). The main drawback is that whenever calling contexts vary slightly from one another, the summary can not be used and new analyses will be required.
- By opposition, a bottom-up analysis starts from the leaf statements of the program (the ones for which we want a summary to be inferred) and propagates up towards the root. This technique has the huge advantage that it is not context sensitive, therefore, in order to be sound, the summaries created for the leaf statements are the most general (as no information on the context is provided) they can therefore be reused for all other occurrences of the statement, ensuring that the body of a function is analyzed at most once. The main drawback of this approach is also the fact that it is not context sensitive. Indeed

as all possible input contexts need to be considered (even those useless for the analysis of the complete program) this will leave us with two choices: (1) Either we try to perform a precise analysis. In the context of a C program manipulating pointers, a precise analysis will need to partition its input with respect to the different possible aliasing patterns, thus inducing a exponential blowup. (2) Or we perform an imprecise analysis, merging all aliasing patterns together. Such an analysis will be very imprecise, especially for our target applications which are programs manipulating strings.

As our primary goal is to be precise and achieve scalability we felt that the bottom-up approach would result in too many spurious computations and decided to modify the classical top-down approach so as to ensure that the scope of application of inferred summaries will stabilize, ensuring that the number of times a statement is reanalyzed is bounded. In order to achieve this we will use the widening mechanism, already provided by every abstract domain. The idea of the widening operator is to perform a "generalization" based on a finite number of observations. We feel that this generalization process can be used when considering the different possible call contexts of a function.

Abstract interpretation is always sound and inferred invariants describe an over approximation of the reachable set of states. Therefore the use of input/output relations discovered on statements must yield an over-approximation. Nonetheless we do not want to give up too much precision to achieve scalability. This was done by using classical techniques to express input/output relations on numerical variables as performed in [CC02], partitioning these relations according to symbolic conditions in the abstract state as proposed by Bourdoncle [Bou92a], and generalizing them using widening operations.

As a result example, consider again the `strcpy` function from Program 4.1 (in the previous chapter). Enabling modular analysis would yield that $\mathfrak{l}_{\texttt{src}} = \mathfrak{l}'_{\texttt{src}}$ and $\mathfrak{l}_{\texttt{dest}} = \mathfrak{l}'_{\texttt{src}}$ where primed variables (resp. unprimed variables) denote the state at the beginning (resp. at the end) of the analysis of the body of the function. Therefore these two relations state that the length of `src` was not modified by the call to `strcpy`, while the length of `dest` is now that of `src`.

5.1.2 Outline of the chapter

In order to present our modular analysis framework, we start in Section 5.2 by providing introductory remarks on modular analyses. Section 5.3 will then present how relational domains can be used to represent Input/Output summaries of numerical abstract transformers. Section 5.4 contains the main results of this chapter as it presents our modular analyzer extending to strings. Section 5.5 provides some preliminary experimental evaluations of our results. Section 5.6 contains a discussion about other works on modularity, and Section 5.7 concludes.

5.2 General remarks

We perform a static analysis in the context of abstract interpretation. Therefore we assume given a representation for the semantics of `C` program.

$$(\mathcal{D}, \subseteq) \xleftarrow{\gamma} (\mathcal{D}^\sharp, \sqsubseteq)$$

We recall that we are given a sound abstract interpreter for this language, that is a $\mathbb{S}^\sharp[\![\texttt{stmt}]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ function for every $\texttt{stmt} \in$ *stmt*, such that:

$$\forall S^\sharp,\ \texttt{stmt},\ \mathbb{S}[\![\texttt{stmt}]\!](\gamma(S^\sharp)) \subseteq \gamma(\mathbb{S}^\sharp[\![\texttt{stmt}]\!](S^\sharp)) \tag{5.1}$$

where $\mathbb{S}[\![\texttt{stmt}]\!] \in \mathcal{D} \to \wp(S)$ denotes the concrete semantics of the language.

$\mathbb{S}^\sharp[\![]\!]$ is usually defined inductively on the syntax of the program (as shown in Chapter 2), therefore performing the analysis of a sequence of instructions requires the analysis of every

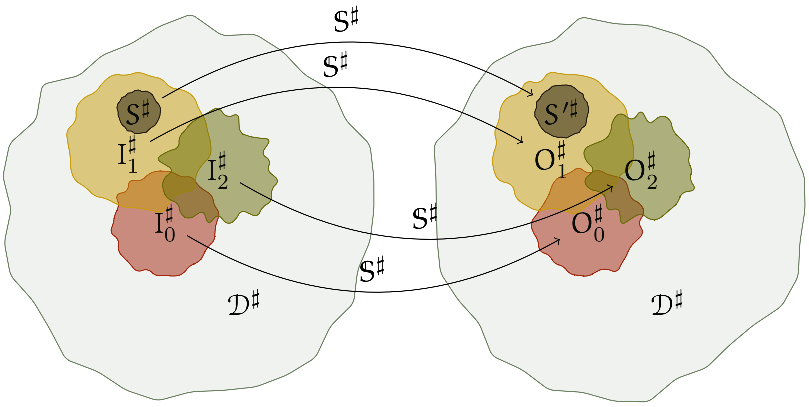


Figure 5.1: $\mathbb{T}_2^\sharp = [I_0^\sharp \mapsto O_0^\sharp] \circledast [I_1^\sharp \mapsto O_1^\sharp] \circledast [I_2^\sharp \mapsto O_2^\sharp]$

atom that composes the statement. As highlighted by the introductory remarks to this chapter, we would like to be able to infer, for every statement `stmt`, a partial function that, given an input abstract state, can produce an output abstract state that is an over approximation of the abstract state we would have obtained by performing the analysis of `stmt`.

Definition 5.1 (Summary function)**.** We call a *summary function* relative to $\mathbb{S}^\sharp$, and denote $\mathbb{T}^\sharp$ a partial function from *stmt* $\times \mathcal{D}^\sharp$ to $\mathcal{D}^\sharp$ such that:

$$\forall S_1^\sharp,\ S_2^\sharp,\ \texttt{stmt},\ S_1^\sharp \sqsubseteq S_2^\sharp \Rightarrow \left\{ \begin{array}{rl} \text{either:} & \mathbb{S}^\sharp[\![\texttt{stmt}]\!](S_1^\sharp) \sqsubseteq \mathbb{T}^\sharp(\texttt{stmt}, S_2^\sharp) \\ \text{or:} & \mathbb{T}^\sharp(\texttt{stmt}, S_2^\sharp) \text{ is undefined} \end{array} \right.$$

This ensures that if the summary function is undefined on some context $S_1^\sharp$, but is defined on some context $S_2^\sharp$ containing $S_1^\sharp$ ($S_1^\sharp \sqsubseteq S_2^\sharp$), then we can soundly return the image of $S_2^\sharp$ by the summary function. This is put forth in the following proposition.

Proposition 5.1. *If $\mathbb{T}^\sharp$ is a summary function relative to $\mathbb{S}^\sharp$, the analyzer $\overline{\mathbb{S}^\sharp}$ defined by:*

$$\overline{\mathbb{S}^\sharp}[\![\texttt{stmt}]\!](S^\sharp) \triangleq \left\{ \begin{array}{ll} \mathbb{T}^\sharp(\texttt{stmt}, S_2^\sharp) & \textit{if } \exists S_2^\sharp, S^\sharp \sqsubseteq S_2^\sharp \wedge \mathbb{T}^\sharp(\texttt{stmt}, S_2^\sharp) \textit{ is defined} \\ \mathbb{S}^\sharp[\![\texttt{stmt}]\!](S^\sharp) & \textit{otherwise} \end{array} \right.$$

is sound in the sense of Equation 5.1

Definition 5.2 ($\circledast$)**.** We define $\circledast$ operating on summary functions:

$$\mathbb{T}_0^\sharp \circledast \mathbb{T}_1^\sharp \triangleq \lambda(\texttt{stmt}, S^\sharp). \left\{ \begin{array}{cc} \mathbb{T}_0^\sharp(\texttt{stmt}, S^\sharp) & \text{if } \mathbb{T}_0^\sharp(\texttt{stmt}, S^\sharp) \text{ is defined} \\ \mathbb{T}_1^\sharp(\texttt{stmt}, S^\sharp) & \text{else if } \mathbb{T}_1^\sharp(\texttt{stmt}, S^\sharp) \text{ is defined} \\ \texttt{undefined} & \text{otherwise} \end{array} \right.$$

This operator enables us to build up summary functions by extension. Indeed if it is undefined on some state or statement, we can extend it with another summary function. We call $\mathbb{T}^\sharp$ an *extension* of $\mathbb{T}_1^\sharp$ when there exists $\mathbb{T}_2^\sharp$ such that $\mathbb{T}^\sharp = \mathbb{T}_1^\sharp \circledast \mathbb{T}_2^\sharp$.

Example 5.1.

- $\mathbb{T}_1^\sharp \triangleq \lambda(\texttt{stmt}, S^\sharp),\ \mathbb{S}^\sharp[\![\texttt{stmt}]\!](S^\sharp)$ is a summary function relative to $\mathbb{S}^\sharp$. This is the most precise summary function relative to $\mathbb{S}^\sharp$. However its computation requires running the analysis on `stmt`. For a function analysis it means always inlining the body of the callee.

- Given $\mathtt{stmt} \in \mathit{stmt}$, $n \in \mathbb{N}$, $(I^\sharp_j, O^\sharp_j)_{j\in[\![0,n-1]\!]} \in (\mathcal{D}^\sharp \times \mathcal{D}^\sharp)^n$ such that $\forall j \in [\![0; n-1]\!]$, $\mathbb{S}^\sharp[\![\mathtt{stmt}]\!](I^\sharp_j) \sqsubseteq O^\sharp_j$, let us define

  $$\mathbf{S}^\sharp_j = [I^\sharp_j \mapsto O^\sharp_j] \triangleq \lambda(\mathtt{stmt}', S^\sharp), \begin{cases} O^\sharp_j & \text{if } \mathtt{stmt}' = \mathtt{stmt} \wedge S^\sharp \sqsubseteq I^\sharp_j \\ \mathtt{undefined} & \text{otherwise} \end{cases}$$

  then $\mathbb{T}^\sharp_2 \triangleq \circledast_{i=0}^{n-1} \mathbf{S}^\sharp_j$ is a summary function relative to $\mathbb{S}^\sharp$. In this case we store every input/output relations we get for reuse.

In Example 5.1, function $\mathbb{T}^\sharp_2$ represents the case where we already performed the analysis of some statement on some inputs and kept a list of the input/output relations corresponding to these inputs. The function can be represented as the list $(I^\sharp_j, O^\sharp_j)_{j\in[\![0,n-1]\!]}$ of such input/output pairs and the computation of the function is reduced to a look-up in this list. Moreover in the framework of abstract interpretation where we only try to compute over-approximations of the set of accessible states, we can reuse a relation $(I^\sharp, O^\sharp)$ for states $S^\sharp$ such that $S^\sharp \sqsubseteq I^\sharp$. However this usage might come with a loss of information, indeed the actual computation of $\mathbb{S}^\sharp[\![\mathtt{stmt}]\!]$ might have produced an abstract element $S'^\sharp \sqsubset O^\sharp$. As an example, consider Figure 5.1 representing a summary function with 3 input/output relations stored. Using the $(I^\sharp_1 \mapsto O^\sharp_1)$ relation instead of reanalyzing the statement for the input state $S^\sharp$ (that is such that $S^\sharp \sqsubseteq I^\sharp_1$) leads to a loss of information as we will only be able to use $O^\sharp_1$. Dually we can remove constraints on $I^\sharp$ before the analysis of the body of the function in order to improve the reusability of the input/output relation obtained, the drawback being that the corresponding output abstract value will be greater thus losing precision. Furthermore computing and storing new $(I^\sharp, O^\sharp)$ relations whenever no existing summary could be used can cause the computation of input/output relations that will never be reused, hence the importance of generalizing $I^\sharp$ in the direction of newly discovered call contexts, so as to tailor summaries to actual call sites abstract values.

Remark 5.1. Consider the statement $\mathtt{stmt} = \mathtt{x = x + 1}$, and assume our abstract domain to be the interval domain. For every input state of the form $\{x \mapsto [\alpha, \beta]\}$, the output state will be of the form $\{x \mapsto [\alpha+1, \beta+1]\}$. A summary function $\mathbb{T}^\sharp$ defined on $\{\mathtt{stmt}\} \times \mathcal{S}^\sharp_m$ in the manner of Example 5.1 (with a finite list of input/output relations) will never yield an analyzer able to express that for every input $[\gamma, \delta]$ the output is $[\gamma+1, \delta+1]$. Indeed the interval domain would produce a set of input/output relations $\{[\alpha_i, \beta_i] \mapsto [\alpha_i+1, \beta_i+1]\}$, and for an input $[\gamma, \delta] \subsetneq [\alpha_0, \beta_0]$ we could only use as output abstract state $[\alpha_0+1, \beta_0+1]$, thus losing information compared to $[\gamma+1, \delta+1]$. Moreover if we only allow a finite number of summaries to be created, and if the number of potentially encountered inputs is unbounded we will have to generate and use a summary returning the $\top$ interval.

5.3 Using relational domains

Relational domains are able to express relations of the form $y = x + 1$. Such a relation can grasp the semantics of stmt from the previous remark. We use the relational aspect of the numerical domain to express relations not only between the values of variables, but also between their values and their input values. This idea was introduced in [CC77a] (also used in e.g. [CC02, Suz19]). Consider two sets of variables $\mathcal{W} = \{x, y\}$ and $\mathcal{W}' = \{x', y'\}$ and the abstract element: $N^\sharp = \{x = y', y = x'\}$. Moreover assume at input that $\{x = 3, y = 5\}$, then using the meet provided by the numerical domain in order to instantiate $N^\sharp$ with input constraints: $\{x = 3, y = 5\}_{|\{x',y',\ x,y\}} \sqcap \{x = y', y = x'\} = \{x = 3, y = 5, x' = 5, y' = 3\}$, and finally $\{x = 3, y = 5, x' = 5, y' = 3\}_{|\{x,y\}} = \{x' = 5, y' = 3\}$. This example emphasized how relational domains are used to express precise input/output relations between numerical variables.

Definition 5.3. Let $\mathcal{W}, \mathcal{W}'$ be sets of variables such that $\mathcal{W} \cap \mathcal{W}' = \emptyset$. Let $N^\sharp \in \mathcal{N}^{\mathcal{W} \cup \mathcal{W}'}$ a numerical abstract state. We define the function $\Lambda(N^\sharp) \in \mathcal{N}^{\mathcal{W}'} \to \mathcal{N}^{\mathcal{W}}$ deriving form $N^\sharp$ by:

$$\Lambda_{\mathcal{W}}(N^\sharp) \triangleq \lambda X^\sharp.\ (X^\sharp_{|\mathcal{W} \cup \mathcal{W}'} \sqcap^{\mathcal{W} \cup \mathcal{W}'} N^\sharp)_{|\mathcal{W}}$$

Example 5.2. Consider $\mathcal{W} = \{x\}$, $\mathcal{W}' = \{x'\}$, the abstract state $N^\sharp = \{x = x' + 1\}$. Consider moreover the following subset of $\mathcal{N}^{\mathcal{W}'}$: $S = \{\{x' = \alpha\} \mid \alpha \in \mathbb{Q}\}$. Then the restriction of $\Lambda_{\mathcal{W}}(N^\sharp)$ to S is the function that associates every abstract state of the form $\{x = \alpha + 1\}$ to the abstract state of the form $\{x' = \alpha\}$.

Remark 5.2. We only defined how we associate a function to a numerical abstract element containing primed variables. This function is total, it is defined over $\mathcal{N}^{\mathcal{W}'}$. In order to use this function in a modular analyzer, its definition set will be restricted to a subset of $\mathcal{N}^{\mathcal{W}'}$ over which the function can soundly be used. Example 5.4, in the following, will provide an illustration of this problem.

5.4 Building the summary functions

5.4.1 Summaries

Based on the remarks from the two previous subsections, themselves based on related works and state of the art results, we define the summary of a statement to be a set of input/transfer relations. A pair input/transfer has the semantics that: whenever a abstract state $S^\sharp$ satisfies the input condition, the application of the transfer relation on $S^\sharp$ is a sound post-condition of the statement. To describe input conditions we will use another abstract element, the notion of satisfaction will then be conveyed by the abstract comparison operator. Knowing whether $S^\sharp$ satisfies some input condition $I^\sharp$, or not, is tested by $S^\sharp \sqsubseteq I^\sharp$. When this holds this ensures that all concrete states represented by $S^\sharp$ are contained in the set of concrete states represented by $I^\sharp$. If the summary was inferred with input hypothesis $I^\sharp$, this ensures that all concrete states from $S^\sharp$ have been considered when building the according transfer relation. Moreover by opposition with the classical input/output summaries, we here have a transfer function, relating the input abstract states with the output abstract states. The importance (for precision) of this relationality was emphasized in the previous section.

When trying to apply the ideas put forth in the previous sections to the String analyzer defined in Chapter 4 we are faced with the following problem: we do not have an out-of-the-box mechanism that enables the lifting of the string abstraction to a relational abstraction as we have for purely numerical abstractions. The string abstraction is composed of: a numerical component (which we can lift to a relational framework) and two symbolic components (the set of cells and the pointer map). An input/output transfer function for the string domain would be an element of the form: $(\langle C, N^\sharp, P\rangle, \langle C', N^{\sharp\prime}, P'\rangle)$, that is a pair of abstract elements. The meaning of such a pair is: if an abstract element satisfies the left element of the pair then the right element is a sound post condition. As we have shown a way to express numerical input/output relations, we factor the two numerical component $N^\sharp$ and $N^{\sharp\prime}$ into a numerical element expressing such a relation.

Definition 5.4 (transfer relation)**.** We define *transfer relations* for the string abstraction (we recall that $\mathfrak{V}$ denotes the subset of memory zones handled by the string abstract domain), denoted $\overline{\mathcal{S}^\sharp_m}$, to be quintuplets of the form:

$$\overline{\mathcal{S}^\sharp_m} \triangleq \{\langle C, P, N^\sharp, C', P'\rangle \mid C, C' \subseteq \mathcal{C}ell \setminus \{\langle V, \_, \_\rangle \mid V \in \mathfrak{V}\},\ N^\sharp \in \mathcal{N},\ P \in \mathcal{P}_C,\ P' \in \mathcal{P}_{C'}\}$$

The support of the numerical component $N^\sharp$ is $C \cup C' \cup \mathfrak{V}^\star \cup \mathfrak{V}^{\star\prime}$:

- cells from C and C';
- string numerical variables (length and allocated size) as well as a primed version of those string variables.

A transfer relation therefore contains: the two symbolic components of the input (the cell set and the pointer map), the same two symbolic components of the output, and a relational transformation from the input variables to the output variables. We now need to define how this transfer relation is applied to an abstract element from the string domain. Given an input abstract element $\mathsf{C}_i, \mathsf{N}^\sharp_i, \mathsf{P}_i$ and a transfer relation $\langle \mathsf{C}, \mathsf{P}, \mathsf{N}^\sharp, \mathsf{C}', \mathsf{P}' \rangle$ we would like the output abstract element to be $\langle \mathsf{C}', \Lambda(\mathsf{N}^\sharp)(\mathsf{N}^\sharp_i), \mathsf{P}' \rangle$, that is: we use the output symbolic component and apply the numerical transformation to the input. However the input numerical component might not be defined on the support expected by the function $\Lambda(\mathsf{N}^\sharp)$. Indeed recall that in the cell domain (which is used as the basis for the string domain) the numerical domain provides constraints on a set of cells that evolve dynamically during the analysis. Once again we rely on the unification operator provided by the underlying cell domain to transform the numerical relation and the input numerical component so that they are compatible.

Definition 5.5 (Semantic of a transfer relation)**.** Given a transfer relation $\mathbf{T} = \langle \mathsf{C}, \mathsf{P}, \mathsf{N}^\sharp, \mathsf{C}', \mathsf{P}' \rangle$ (in $\overline{\mathcal{S}^\sharp_m}$) we define:

$$\begin{aligned} [\![\mathbf{T}]\!] \triangleq \lambda X^\sharp \in \mathcal{S}^\sharp_m.\ \ & \textbf{let}\ \langle \_, \mathsf{N}^\sharp_i, \_ \rangle, \langle \_, \mathsf{N}^\sharp_r, \_ \rangle = \textbf{unify}(X^\sharp, \langle \mathsf{C}, \mathsf{N}^\sharp, \mathsf{P} \rangle)\ \textbf{in} \\ & \langle \mathsf{C}', \Lambda(\mathsf{N}^\sharp_r)(\mathsf{N}^\sharp_i), \mathsf{P}' \rangle \end{aligned}$$

This definition provides the semantics attached to a transfer relation. We now can define the summaries that will be inferred for C statements. Following the idea introduced before, summaries will be sets of pairs containing: an input constraint and a transfer relation. This can be seen as a partitioning of transfer relations according to the input abstract element.

Example 5.3. Continuing on Example 4.1. Consider the following transfer relation: $\langle \{ \langle \mathsf{a}, 0, \textbf{u32} \rangle (\triangleq \mathbf{a}_0), \langle \mathsf{a}, 4, \textbf{u32} \rangle (\triangleq \mathbf{a}_1) \}, \emptyset, \{ \mathbf{a}_0 = \mathbf{a}'_0, \mathbf{a}_1 = \mathbf{a}'_1 + 1 \}, \{ \mathbf{a}_0, \mathbf{a}_1 \}, \emptyset \rangle$. We want to apply this relation on input state $\langle \{ \langle \mathsf{a}, 0, \textbf{u64} \rangle (= \mathbf{a}) \}, \{ \mathbf{a} = 2^{32} + 2 \}, \emptyset \rangle$, the unification yields: $\mathsf{N}^\sharp_i = \{ \mathbf{a} = 2^{32} + 2, \mathbf{a}_0 = 2, \mathbf{a}_1 = 1 \}$ and $\mathsf{N}^\sharp_r = \{ \mathbf{a}_0 = \mathbf{a}'_0, \mathbf{a}_1 = \mathbf{a}'_1 + 1 \}$ with $\mathbf{a}$ unconstrained. Applying the numerical relation yields the output state: $\langle \{ \mathbf{a}_0, \mathbf{a}_1 \}, \{ \mathbf{a}_0 = 2, \mathbf{a}_1 = 2 \}, \emptyset \rangle$.

Definition 5.6 (Summaries)**.** We define the set of *statement summaries* as:

$$\mathcal{C} \triangleq \wp(\mathcal{S}^\sharp_m \times \overline{\mathcal{S}^\sharp_m})$$

Note that summaries contain an input condition, in addition to the already existing symbolic input found in the transfer relation. This is required for soundness. As transfer relations are computed dynamically during the analysis of the targeted statement, the computation of this transfer relation is done by running the abstract interpreter on the body of the statement. Hence the projection of the numerical component from the transfer relation onto the input variables might be relaxed compared to its initial value. It would therefore be unsound to use this projection as input test for the use of the transfer function. The following example emphasizes this problem.

Example 5.4. Consider Program 5.2, where we will only consider numerical components. Assume that `a` points to some integer `x`, assume moreover that this program is called in context $\{\mathtt{x} \geqslant 1\}$. In order to build an input/output numerical relation we add a primed variable `x'` to denote the initial value of `x`, and we equate it to `x` initially: $\mathsf{N}^\sharp_0 \triangleq \{\mathtt{x} = \mathtt{x'}, \mathtt{x} \geqslant 1\}$. As $\mathtt{x} \geqslant 1$ we only analyze the then branch and perform: $\mathbb{S}^\sharp[\![\mathtt{x} \leftarrow \mathtt{x} + 1]\!](\mathsf{N}^\sharp_0)$. As we made no hypothesis

```
void strcat(char* dest, char* src)
{
  int i; int j;
  for (i=0; dest[i]!='\0';i++) ;
  for (j=0; src[j]!='\0';j++)
    dest[i+j] = src[j];
  dest[i+j] = '\0';
}
```

Program 5.1: `strcat`

```
void extend(int* a) {
  if (*a >= 0) {
    (*a)++;
  } else {
    (*a)--;
  }
}
```

Program 5.2: Filters illustration

on the precision of $\mathbb{S}^\sharp[\![.]\!]$, this might yield the (sound) post-condition: $N_1^\sharp \triangleq \{x = x' + 1\}$. The constraint $x' \geqslant 1$ has been lost. We therefore obtain as the numerical input/output relation for this program that $\{x = x' + 1\}$. Projecting $N_1^\sharp$ onto the set of initial variable $\{x'\}$ we get $\top$. We see however that this can not be used as a sound input condition for function `extend` as `extend(-1)` returns -2.

5.4.2 Inferring and proving summaries

In the following all abstract operators $\sqsubseteq$, $\sqcup$, $\triangledown$ refers to those of the string abstract domain, defined in Chapter 4. We recall that the computation of these operators boils down to the application of the corresponding numerical operator on the underlying numerical domain, once cell sets have been unified. We feel that two analyses starting from different aliasing patterns should be kept separated in order to improve precision. Indeed, analyzing `strcat(p,q)` (see Figure 5.1) without any hypothesis on the possible aliasing of `p` and `q` would result in a huge loss of precision and in false alarms being raised at every call (`p` and `q` might be aliased, which would raise a segmentation fault). Performing an analysis for every possible aliasing scheme would result in a combinatorial blow up, moreover we might perform analysis for aliasing schemes that will never occur at any call site. For these reasons we will only perform analyses on demand. Our choice of summaries from Definition 5.6 shows that analysis will be partitioned. The decision to extend a partition or to build a new one will be based mostly on the ”symbolic” part of the abstract domain. The heuristic we chose was to separate abstract states with different aliasing patterns, but also those where the unification of the cell or string sets would induce major differences in the numerical domain set. The summary function is extended on demand, meaning that when the analyzer encounters a function call, it tries to use an existing relation and if none can be found it builds a new relation or it generalizes an existing one. Generalization of a relation is done in the following way: assume known a relation with input $I^\sharp$ and an abstract state $S^\sharp$, such that $S^\sharp \not\sqsubseteq I^\sharp$. If the analyzer deems that $S^\sharp$ and $I^\sharp$ should be in the same relation (e.g. because they have the same aliasing pattern), we perform a new analysis of the function starting from $I^\sharp \triangledown (S^\sharp \sqcup I^\sharp)$, that is a generalization, of $I^\sharp$, by the mean of the widening operator. This ensures that, given an aliasing, a function will be analyzed only a bounded number of times and that the input of the obtained relation is tailored to the actual values at call site. Algorithm 5.1 provides a definition of this extension technique. It is used on statements for which we want to infer summaries. The set of summaries is empty initially, and as Algorithm 5.1 returns the new summary for the statement (in addition to a sound post-condition), the next call needs to be given the previously returned summary. Algorithm 5.1 uses three undefined functions:

- **identity_relation**: this functions takes as input an abstract state and return the identity transfer relation. This is obtained by duplicating the symbolic component of the abstract state, duplicating and “priming” variables in the numerical relation, and finally adding

Algorithm 5.1: try_and_extend algorithm

```
Input  : a statement stmt,
         a summary for statement stmt: $C = \{(I^\sharp_1, \mathbf{T}_1), \dots, (I^\sharp_n, \mathbf{T}_n)\}$,
         an integer threshold $m$,
         an input state $S^\sharp$
Output: a new summary, and a postcondition
if $\exists (I^\sharp_k, \boldsymbol{T}_k) \in C, S^\sharp \sqsubseteq I^\sharp_k$ then
    return $(C, [\![\boldsymbol{T}_k]\!](S^\sharp))$;
else if $n < m$ then
    $S^\sharp_2 \leftarrow \mathbb{S}^\sharp[\![\texttt{stmt}]\!](\textbf{identity\_relation}(S^\sharp))$;
    $(\mathbf{T}_{n+1}, O^\sharp) \leftarrow \textbf{extract\_relation}(S^\sharp_2)$;
    return $(C \cup (S^\sharp, \boldsymbol{T}_{n+1}), O^\sharp)$;
else
    $I^\sharp_k = \textbf{find\_closest}(S^\sharp, C)$;
    $S^\sharp_2 \leftarrow \mathbb{S}^\sharp[\![\texttt{stmt}]\!](\textbf{identity\_relation}(I^\sharp_k \triangledown S^\sharp))$;
    $(\mathbf{T}_{n+1}, O^\sharp) \leftarrow \textbf{extract\_relation}(S^\sharp_2)$;
    return $(C \setminus (I^\sharp_k, \boldsymbol{T}_k) \cup (S^\sharp, \boldsymbol{T}_{n+1}), O^\sharp)$;
end
return res;
```

Algorithm 5.2: identity_relation algorithm

```
Input  : an abstract state $\langle C, N^\sharp, P \rangle$
Output: The identity relation
res $\leftarrow N^\sharp$;
foreach $x \in \textit{def}(N^\sharp)$ do
    res $\leftarrow \mathbb{S}^\sharp[\![\texttt{Assume}(x = x')]\!] \circ \mathbb{S}^\sharp[\![\texttt{add}(x')]\!](\texttt{res})$ ;
endfch
return $\langle C, P, \texttt{res}, C, P \rangle$;
```

equality constraints between primed and unprimed variables. **identity_relation** is defined in Algorithm 5.2.

- **extract_relation**: this function takes as input a transfer function and extracts from it: the transfer relation and the output state (this is obtained by removing the input variables from the numerical environment). **extract_relation** is defined in Algorithm 5.3.
- **find_closest**: this function takes as input an abstract state $S^\sharp$ and a summary C, it is heuristically based as it looks for an input condition from C that is the most similar to $S^\sharp$. We chose a measure of similarity based on aliasing. Indeed, recall that in the cell abstraction, the pointer map component of an abstract element associates to each cell with pointer type an over-approximation of the set of bases that may be pointed to by the cell. **find_closest**$(S^\sharp, C)$ tries to find an element $(I^\sharp, \mathbf{T})$ of C, the pointer map component of $I^\sharp$ being identical to the pointer map component of $S^\sharp$. If no such element can be found, it tries to look for one with as few differences as possible.

Note the we assumed in Algorithm 5.1 that $\mathbb{S}^\sharp[\![\texttt{stmt}]\!]$ can be executed on transfer relations in addition to usual abstract elements. This is implemented by considering the output component of the transfer relation as an abstract element and applying the usual transfer relations on this abstract element. Therefore building numerical relations does not require a transformation of the intra-procedural iterator. Indeed variables are added to the numerical domain with equality constraints between primed and unprimed variables. Analysis is then performed as if primed variables were not present in the numerical domain and they are removed after storing the summary.

Algorithm 5.3: extract_relation algorithm

```
Input  : an abstract relation $\langle C, P, N^\sharp, C', P' \rangle$
Output: The same abstract relation and the output state
res ← $N^\sharp$;
foreach primed variable $x' \in def(N^\sharp)$ do
    res ← $\mathbb{S}^\sharp[\![$remove(x)$]\!]$(res) ;
endfch
return $\langle C, P, N^\sharp, C', P' \rangle, \langle C', \texttt{res}, P' \rangle$;
```

Example 5.5. Consider the statement `s32 x = a + 1` (with *typeof*(a) = **s32**), from input state: $\langle \{\mathbf{a}'\}, \{\mathbf{a}' = \mathbf{a}\}, \emptyset \rangle$, $\mathbf{a}'$ has been added to denote the initial value of **a**. The output state is then $\langle \{\mathbf{a}', \mathbf{x}'\}, \{\mathbf{a}' = \mathbf{a}, \mathbf{x}' = \mathbf{a} + 1, \}, \emptyset \rangle$. From this we deduce the relation: let $I^\sharp_\alpha$ be some input state, if the set of cells of $I^\sharp_\alpha$ is precisely $\{\mathbf{a}\}$ and if the numerical domain of $I^\sharp_\alpha$ satisfies the condition $\mathbf{a} = \alpha$ and if the pointer map is empty, then a possible output state is $\langle \{\mathbf{a}, \mathbf{x}\}, \{\mathbf{a} = \alpha, \mathbf{x} = \alpha + 1\}, \emptyset \rangle$.

Example 5.6. Consider now the function `strcat` of Figure 5.1. The modular analysis of this function yields a relation stating that:

if `dest` points (at offset 0) to some memory location s, with length s_l and allocated size s_a, and if `src` points (at offset 0) to some memory location t with equivalent length and allocated size definition and $t \neq s$

then $\{s_l' = t_l + s_l, s_a' = s_a, t_a' = t_a, t_l' = t_l, t_l' \geqslant 0, t_l' \leqslant t_a' - 1, t_l' \leqslant s_l', s_l' \leqslant s_a' - 1\}$

Therefore thanks to the $s_l' = t_l + s_l$ relation, if another call to `strcat` is performed in a state where $s_l = \alpha$ and $t_l = \beta$ for some α and β, our analyzer, can conclude (without reanalysis) that the length of the string t_l at the end of the analysis is $\alpha + \beta$.

5.4.3 Improvements

The use of a relational domain for the description of summaries improves their precision (by comparison with a simple non relational input/output list as presented in Example 5.1). Not only do we want our summaries to be precise, but we want them to be reusable. For this reason several improvements were designed for the **try_and_extend** algorithm:

Framing. We loosen the input state by removing some memory blocks (meaning we leave out constraints on these regions) from the input state. This plays the role of the framing rule in separation logic. In order to be able to soundly restore the constraints on these memory blocks after the function analysis, they need to be memory blocks that we know will not be used by the function. As a first framing rule we can, for example, remove every memory region that can not be accessed by pointer dereferencing from one of the argument of the function or from global variables.

Example 5.7. Let us assume that a call `incr(p)` is made in the input context:

$$\langle \{ \langle \mathsf{p}, 0, \mathbf{ptr} \rangle (\triangleq \mathbf{p}), \langle \mathsf{b}, 0, \mathbf{u8} \rangle (\triangleq \mathbf{b}), \langle \mathsf{a}, 0, \mathbf{u8} \rangle (\triangleq \mathbf{a}) \}, N^\sharp, \{ \mathbf{p} \mapsto \{\mathsf{a}\} \} \rangle$$

The memory zone $\langle \mathsf{b}, 0, \mathbf{u8} \rangle$ can not be accessed during the analysis of `incr(p)`. Therefore the cell is removed from the cell set before analysis and the according variable is removed from the numerical component. We can thus perform the modular analysis of statement `incr(p)` starting from $S^\sharp = \langle \{\mathbf{p}, \mathbf{a}\}, N^\sharp_{|\{\mathbf{p}, \mathbf{a}\}}, \{\mathbf{p} \mapsto \{\mathsf{a}\}\} \rangle$. If we find a transfer relation **T** (line 2 of Algorithm 5.1) that can be applied on $\underline{S}^\sharp$, we modify the numerical component of the transfer relation **T** by assuming that **b** = **b'** and apply the transfer relation. Note that in the special case where the underlying

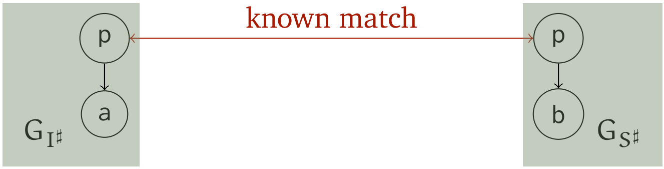


Figure 5.2: graph isomorphism for variable renaming

function name	Loc
`fprint_pascal_string`	29
`null_terminate`	11
`space_terminate`	12
`extendfilename`	17
`remove_newline`	18
`insert_long`	21
`join`	44

Table 5.1: Functions from web2c analyzed

numerical domain is non relational, this amounts to removing the interval associated to **b** from the numerical domain and restoring it after the analysis. Doing the same in the relational case would lose the relationality between variables in the frame and those not in the frame.

In Chapter 7 we propose a more sophisticated framing mechanism.

Quantification. When a summary is created, some memory blocks are quantified universally, therefore when trying to apply a summary we try to unify the memory blocks from the actual input state with those of the summary input state. Continuing on the example of the incrementation function, assume that we have discovered a precise transfer relation applicable under input condition $I^\sharp = \langle \{\mathbf{p}, \mathbf{a}\}, \{\mathbf{p} = 0, \mathbf{a} \geqslant 0\}, (\mathbf{p} \mapsto \{\mathbf{a}\}) \rangle$ and a new call is made to `incr` with input state $S^\sharp = \langle \{\mathbf{p}, \mathbf{c}\}, \{\mathbf{p} = 0, \mathbf{c} = 0\}, (\mathbf{p} \mapsto \{\mathbf{a}\}) \rangle$. We can not reuse the discovered transfer relation because $S^\sharp \not\sqsubseteq I^\sharp$. However, if `a` is not a global variable it plays the same role as `c` in the body of function `incr`. For this reason we look for a renaming σ such that $S^\sharp[\sigma] \sqsubseteq I^\sharp$. A necessary condition for this to hold is to ensure that the pointer map component of the cell abstraction satisfies a point-wise inclusion. We build the graphs associated with the pointer maps component of each abstract element (the input condition and the input state), as shown in Figure 5.2, and try to find a matching of the two graphs that preserves transitions (a graph isomorphism) and matches together the arguments of the function (known match from Figure 5.2). Every matching gives us a renaming σ that we can then apply to the abstract element. Finally we look for a matching σ that satisfies $S^\sharp[\sigma] \sqsubseteq I^\sharp$. Here we can see that the renaming $\sigma = (\mathtt{a} \mapsto \mathtt{c})$ applied to $S^\sharp$ will indeed yield an abstract element satisfying $I^\sharp$. This will allow us to use the relation associated with filter $I^\sharp$ for input $S^\sharp$.

5.5 Preliminary experimental evaluation

The String abstract domain from Chapter 4 was added to the library of existing domains in Mopsa, and a new inter-procedural iterator was added to implement the modular analysis presented in this chapter. The current analyzer is in development, it is able to analyze all `C` code fragments presented here, but can not tackle complete realist `C` projects yet. To test our modular string analysis, we thus considered the examples and benchmarks used in previous works on string analysis [AGH06], [DRS01].

```
1   char* insert_long(char* cp)
2   {
3     char tbuf[BUFSIZ];
4     int i;
5     for (i=0;&buf[i]<cp;++i)
6       tbuf[i] = buf[i];
7     strcpy(&tbuf[i],"(long)");
8     strcpy(&tbuf[i + 6], cp);
9     strcpy(buf, tbuf);
10    return cp + 6;
11  }
```

Program 5.3: `insert_long` from web2c

```
typedef struct {
  char* f;
} s;
char buf[10];

void init(s* x) {
  x[1].f = buf;
}
int main () {
  s a[2][2];
  s* ptr = (s*) &(a[1]);
  init(ptr);
  ptr = (s*) &(a[0]);
  strcpy(a[1][1].f,"strcpy ok”);
  strcpy(a[1][1].f,"strcpy not ok”);
}
```

Program 5.4: Program from [AGH06]

In related works, Allamigeon et al. mentioned in [AGH06], Section 5, that the most difficult example they had to deal with were calls to `strcpy` performed on string placed in a structure, itself placed in a matrix, and accessed via pointer manipulations (see Program 5.4). This example was successfully analyzed with the version of `strcpy` defined in Program 4.1 and with an alternate implementation found in Qmail (see Program 4.2), the second case was more complex and required the use of partitioning. Our ability to easily deal with the pointer manipulations from Program 5.4 comes from the use of the Cell domain to deal with low-level features of `C`. Indeed our abstraction is not dependant upon the way strings are accessed as all string accesses are transformed to pointer manipulations.

We are able to tackle most of the programs from web2c mentioned in [DRS01] (7 out of 9, programs that could not be analyzed are due to the fact that we do not have yet implemented all the features of the `C` language, see Table 5.1). Note moreover that both [AGH06] and [DRS01] defined special abstract transformations for `strcpy`, whereas we perform a modular analysis of the function. The number of lines of codes from Table 5.1 does not take into account library functions as we tested with several different implementations of `strcpy`, `strcat` functions. The precision of these analyses (number of errors and false alarms) is similar to that of [DRS01] and the execution time of the analyzer was always below 2 seconds. We do not provide a comparison between the running times of the analyzer with and without the modular iterator. Indeed, we first focused on improving the framing mechanism so as to reduce the analysis time of the modular iterator. As an example consider Program 5.3, starting the analysis under the conditions that: `cp` points to `buf`, a buffer of size `BUFSIZ` before the first `'\0'` character, produces alarms at line 9 and 10. Indeed under such hypothesis `strcpy` tries to write outside of `tbuf`.

5.6 Related works

The goal of the modular analysis iterator defined here is to provide a new inter-procedural iterator. When trying to see the program as a control flow graph, inter-procedural analysis diverges from intra-procedural in the fact that not all paths should be considered when computing the set of reachable states at a given program point, at the risk of a huge loss of precision. Indeed return values from all possible inputs of a function are propagated to every call site. Sharir and Pnueli [SP78] provide two approaches to this problem: the functional approach and the call string approach. The call string approach tags the inferred data with a stack a strings to denote the calling context in which it was discovered. Our work can be described in the functional approach. Reps et al.[RHS95] translated this data-flow problem into graph reachability problems in the special case where the set of potentially inferred facts is finite.

Cousot and Cousot mentioned in [CC02] the importance of performing modular analyses and described several methods to design them. An efficient way is to use user-provided contracts as in [FL10]. Our goal was to infer contracts, therefore works closest to ours would be the input/output inference performed by Bourdoncle in [Bou92a], however this method was limited to non-relational (interval) domains, unlike our method, which is thus more expressive (see Remark 5.1). In [CC02], numerical relations are used to represent the semantics of a set of statements, however this is limited to numerical programs whereas we extend the method to consider both numbers and pointers, including pointer arithmetic.

The semantics of a statement is a function from $\mathcal{D}$ to $\mathcal{D}$, in the definition of a modular abstract interpreter we would like to abstract this semantics. This amounts to the definition of an abstract domain $(\mathcal{D} \times \mathcal{D})^\sharp$. Several techniques [CC94] can be devised to obtain such an abstraction. Using sets of input/output pairs amounts to choosing as abstraction the set $\mathcal{D}^\sharp \times \mathcal{D}^\sharp$ which is non relational. In [JGR05] a relational abstraction is proposed, it requires however $\mathcal{D}^\sharp$ to be finite.

The analyzer proposed by Sotin and Jeannet in [SJ11] is able to infer input/output relations of the form proposed in Section 5.4. Nevertheless they consider a subset of `C` that does not contain pointer arithmetic, union types nor pointer casts.

Müller-Olm and Seidl [MS07] and Sharma and Reps [SR17] proposed domains specialized in the discovery of numerical input/output relations on statements, in both cases the relations discovery is performed during the analysis of the statement by a special domain.

In Section 5.4 we mentioned that we implemented a mechanism to infer framing in order to improve analysis reusability, framing mechanisms are fundamentals in tools based on separation logic such as Smallfoot [BCO05] or Infer [CDD$^+$15b].

We mentioned in the introductory remarks that two families of modular analysis can be designed: the top-down and bottom-up approaches. Several works tried to mix these two techniques to get the better of both world [NKH04, ZMNY14].

In more recent works, Kranz et al.[KS18] define an inter-procedural iterator based on Heyting completions. In [BH19] the authors provide a bottom-up inter-procedural analyzer targeting numerical properties. They choose an approach that is dual to ours as they build their summary partitioning by successive refinements whereas we enlarge ours.

5.7 Conclusion

Upon the String abstraction from Chapter 4, which is an abstraction able to handle real life C code, with an added precision for the length of C buffer, we defined an inter-procedural iterator designed to increase statement analysis reusability without having to lose precision. We provided preliminary experimental results by testing our modular analysis on examples from the literature.

These preliminary experimental results were used as a measure of the expressiveness capabilities and reusability of the contracts inferred by our inter-procedural iterator. The expressiveness of the contracts seems sufficient for a precise analysis, however the reusability should be improved for the iterator to scale. The generalization mechanism introduced by the use of contracts widening ensures that the number of reanalysis of a given statement will be bounded thus ensuring reusability ultimately. This can be bettered by an improved framing mechanism. At this point framing is based purely on removing memory zones that are not reachable in the state at call site, however reachability does not imply that the memory zone is useful to the analysis (consider for example the set of global variables in a C program). In order to improve the framing mechanism we decided to design a pre-analyzer. Its goal is to discover which memory zones might be read and/or modified by the statement. This distinction is important as, if we know that a memory zone is read but unmodified by a statement, it is kept in the input state, but its output value can be refined by the input value when the analysis was too imprecise.

A first pre-analysis was designed (not presented in this thesis), the memory was represented as a graph, which nodes were to be folded during the actual analysis at call site. However we were not able to design a precise enough widening operator for the analysis of looping statements as the graph structure did not provide enough regularities.

In order to add regularities, we decided to represent the set of potentially accessed memory zones as a set of C pointer manipulating expressions. This led to the development of the tree abstraction presented in Chapter 7.

Chapter 6

Numerical Abstraction

6.1 Introduction

As emphasized in Chapter 2 and by their extensive use in Chapters 4, 5 and 7, static analysis by abstract interpretation relies heavily on the use of numerical abstractions. Classical numerical abstractions represent sets of maps over a finite set of variables $\mathcal{V}$ to $\mathbb{I}$, where $\mathbb{I} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$. In this introduction, we take a look at some C and Python programs and show how classical numerical domains could not be used for their analysis. We emphasize that this comes from two severe restrictions of these domains which are that definition sets of represented numerical maps must be (1) homogeneous and (2) finite and set at analysis start up.

6.1.1 Homogeneous definition sets

In Chapter 5, we emphasized that several analyses of the body of a function could be done at once by joining the different states at call sites. Indeed consider Program 6.1, in order to analyze the body of `f()` (resp. `g()`) we need to provide abstract transformers for statement ● (resp. ●). In a classical top-down analysis abstract post-conditions of function calls are computed by inlining the body of the function (as was described in Chapter 5) and analyzing it. However in the example of Program 6.1 we see that performing only one analysis with input `*x = 0` would be precise enough to provide post-conditions for both calls. However in order to provide an input/output relation (for function `incr`) that is usable both for ● and for ●, its input condition needs to be satisfied by environments on both one and two variables. As has been shown in Chapter 5, input conditions are described as an abstract element, therefore we need to define a representation for sets of numerical environments with heterogeneous definition sets.

Some features in programming languages like C or Python can produce sets of maps defined on different supports (called *heterogeneous* sets) at a given program point, e.g. an `if` statement in a C program where only one branch dynamically allocates memory. As another example, consider Program 6.2 written in Python. At program point ■, all concrete numerical environments are defined on `x` and `y`. They can be abstracted using a numerical domain such as intervals or polyhedra. However `x` is changed into a string under the condition that `x` $\leqslant$ `y`. Therefore some of the reachable numerical environments at program point ● are defined on $\{$`x`, `y`$\}$, while others only on $\{$`y`$\}$. Usual numerical domains do not provide a way to represent such sets of environments, hence several techniques have been devised to answer this problem. A first approach would be to partition the set of reachable states according to their support, here this would yields the two polyhedra: $\{0 \leqslant y, y \leqslant 10\}$ and $\{0 \leqslant y, y < x, x \leqslant 10\}$. Albeit very precise, this technique is costly as it might induce an exponential blow-up. Another method to represent such environments would be to remove `x` from the abstract environment to denote that it is not numerically defined. Using the interval domain, the abstract environments that need to be merged at the end of the first `if` statement would be $(\mathtt{y} \mapsto [0;10])$ and $(\mathtt{x} \mapsto [0;10], \mathtt{y} \mapsto [0;10])$. It is then possible

```
void incr(int* x) {
  *x = *x + 1
}

int f() {
  int a = 0;
  ●incr(&a);
}

int g() {
  int b = 0;
  int c = 42;
  ★incr(&b);
}
```

Program 6.1: Uncomparable input states

```
■# 0<=x<=10 and 0<=y<=10
if x<=y:
    x = "hello"
●
if x==y:
    ★
```

Program 6.2: Python heterogeneous environments

```
if (...) {
  s1 = malloc(n);
  s1[x] = '\0';●
} else {
  s2 = malloc(n);
  s2[x] = '\0';★
}
```

Program 6.3: String initialization

to slightly modify the usual interval join operation: so that the invariant inferred for program point ● is $(\mathtt{x} \mapsto [0;10], \mathtt{y} \mapsto [0;10])$:

$$m_1 \sqcup m_2 = \lambda x \in \mathbf{def}(m_1) \cup \mathbf{def}(m_2). \begin{cases} m_1(x) \sqcup_I m_2(x) & \text{if } x \in \mathbf{def}(m_1) \cap \mathbf{def}(m_2) \\ m_1(x) & \text{otherwise if } x \in \mathbf{def}(m_1) \\ m_2(x) & \text{otherwise} \end{cases}$$

where $\sqcup_I$ is the interval join operator. Indeed if the numerical value of x is read, we know that it is in the range $[0;10]$. This technique has the major drawback that it can only be used with non relational domains, here we used intervals and could therefore not prove that ★ is unreachable. Finally, we could add a spurious binding to x when it is undefined [All08]. However setting x to any value in the interval $[0;10]$ at the end of the `else` branch induces a precision loss preventing us from proving that ★ is unreachable. In order to get this result we would need to choose for x a value not in its original interval. Choosing a spurious value for x will necessarily induce a precision loss, moreover guessing the value "minimizing" the precision loss is hard. The abstraction presented in this chapter is relational and does not require adding spurious values, it is therefore able to prove that ★ is unreachable. Indeed the invariant inferred at ● is: if x and y are numerically defined then $\mathtt{x} > \mathtt{y}$. Note that we do not need partitioning to infer this invariant.

Even though it is not directly related to a language feature, one might need to represent sets of maps defined over heterogeneous supports in order to build higher level abstractions. These abstractions are parameterized by an underlying numerical domain in which they store variables [Cou03]. For example consider Prog. 6.3 in which two strings are conditionally allocated. Reusing the abstraction from Chapter 4, the numerical invariant at program point ● (resp. ★) will be $\{\mathtt{s1}_l \leqslant \mathtt{x}\}$ (resp. $\{\mathtt{s2}_l \leqslant \mathtt{x}\}$), where $\mathtt{s1}_l$ (resp. $\mathtt{s2}_l$) denotes the position of the first `'\0'` in `s1` (resp. `s2`). Classical homogeneous numerical domains cannot provide a precise postcondition for the `if` statement as the two environments to be joined are not defined on the same set of variables, the only sound result would thus be $\top$ whereas we would like to obtain

```
void* memset(void* b, int c, size_t len) {
    char* p = (char*)b;
    for (size_t i = 0; i != len; ++i) {
        p[i] = c;
    }
    return b;
}
```

Program 6.4: Memset

$\{\mathtt{s1}_{\mathfrak{l}} \leqslant \mathtt{x} \wedge \mathtt{s2}_{\mathfrak{l}} \leqslant \mathtt{x}\}$ as a postcondition, with the meaning that whenever $\mathtt{s1}_{\mathfrak{l}}$ and x are defined, they satisfy $\mathtt{s1}_{\mathfrak{l}} \leqslant \mathtt{x}$, similarly for s2.

As mentioned in the outline of this thesis, the goal of Chapter 7 will be to set up abstractions for sets of trees with numerically-labeled leaves as we want to represent sets of expressions, encoding access paths in data structures. Sets considered may contain trees with different number of numerically-labeled leaves. Our abstraction will rely on numerical domains to store constraints on leaves. Therefore Chapter 7 will assume the existence of numerical domains over heterogeneous definition sets.

6.1.2 Fixed finite definition set

Another restriction of classical numerical domains is that they can only represent sets of maps over of given finite definition sets. Program 6.4 is a definition of the memset function which, given a pointer b, a constant c and an integer len, sets b[0], b[1], ..., $\mathtt{b[len} - \mathtt{1]}$ to the value c. Using a numerical domain (e.g. the octagons) to precisely represent the post-condition of memset we need an abstract element with definition set containing b[0], b[1], ..., $\mathtt{b[len} - \mathtt{1]}$. This is not possible with classical numerical domains when len is a parameter of this analysis. Solutions based on variable summarization [GDD$^+$04] have been devised to answer this problem, we will show in this chapter that this method is compatible with the representation of heterogeneous environments.

6.1.3 Outline of the chapter.

The remainder of the chapter will be organized as follow: Section 6.2 will define the concrete semantics we want to abstract. Section 6.3 will describe state of the art techniques usually employed to circumvent restrictions presented in this introduction. There will also be mentions of their limitations and why we chose to devise new techniques to answer these restrictions. Section 6.4 will give the core definitions and results of this chapter while emphasizing on one of the main drawbacks of our work: the handling of numerical domains over integers. This is addressed in the following Section 6.5. Section 6.6 then provides an extension of the numerical abstraction of Section 6.4, this extension provides a precision/efficiency trade-off. Finally Section 6.7 focuses on the lifting of classical concretization to represent environments with potentially unbounded definition set by means of summarizing variables.

6.2 Concrete semantics

In this section we define the concrete semantics we are interested in abstracting. This semantics is defined over the concrete world: $\mathfrak{M} \triangleq \wp(\mathcal{V} \nrightarrow \mathbb{I})$, the set of all sets of partial maps from $\mathcal{V}$ to $\mathbb{I}$. $\mathfrak{M}$ is ordered by the inclusion relation $\subseteq$. Once our concrete semantics introduced, we show

```
⓪a = 0
①if ?:
    ②a = 1
    ③b = 1④
else:
    ⑤c = 2⑥

⑦if b <= 5:
    ⑧a = 2
⑨
```

Program 6.5: Heterogeneous environments Python

$$S_{\textcircled{0}} = \{()\} \tag{6.1}$$

$$S_{\textcircled{1}} = \mathbb{S}\llbracket \mathtt{a} \leftarrow \mathtt{0} \rrbracket_{\mathbb{Z}} \circ \mathbb{S}\llbracket \mathtt{add(a)} \rrbracket_{\mathbb{Z}}(S_{\textcircled{0}}) \tag{6.2}$$

$$S_{\textcircled{2}} = \mathbb{S}\llbracket \mathtt{Assume(?)} \rrbracket(S_{\textcircled{1}}) \tag{6.3}$$

$$S_{\textcircled{3}} = \mathbb{S}\llbracket \mathtt{a} \leftarrow \mathtt{1} \rrbracket_{\mathbb{Z}} \circ \mathbb{S}\llbracket \mathtt{add(a)} \rrbracket_{\mathbb{Z}}(S_{\textcircled{2}}) \tag{6.4}$$

$$S_{\textcircled{4}} = \mathbb{S}\llbracket \mathtt{b} \leftarrow \mathtt{1} \rrbracket_{\mathbb{Z}} \circ \mathbb{S}\llbracket \mathtt{add(b)} \rrbracket_{\mathbb{Z}}(S_{\textcircled{3}}) \tag{6.5}$$

$$S_{\textcircled{5}} = \mathbb{S}\llbracket \mathtt{Assume(?)} \rrbracket_{\mathbb{Z}}(S_{\textcircled{1}}) \tag{6.6}$$

$$S_{\textcircled{6}} = \mathbb{S}\llbracket \mathtt{c} \leftarrow \mathtt{1} \rrbracket_{\mathbb{Z}} \circ \mathbb{S}\llbracket \mathtt{add(c)} \rrbracket_{\mathbb{Z}}(S_{\textcircled{5}}) \tag{6.7}$$

$$S_{\textcircled{7}} = S_{\textcircled{6}} \cup S_{\textcircled{4}} \tag{6.8}$$

$$S_{\textcircled{8}} = \mathbb{S}\llbracket \mathtt{Assume(b} \leqslant \mathtt{5)} \rrbracket_{\mathbb{Z}}(S_{\textcircled{7}}) \tag{6.9}$$

$$S_{\textcircled{9}} = \mathbb{S}\llbracket \mathtt{a} \leftarrow \mathtt{2} \rrbracket_{\mathbb{Z}} \circ \mathbb{S}\llbracket \mathtt{add(a)} \rrbracket_{\mathbb{Z}}(S_{\textcircled{8}}) \tag{6.10}$$

that our definitions are expressive enough to model the behaviour of the examples introduced in Section 6.1.

6.2.1 Definition

Our concrete semantics will be defined only by the few following concrete transformers:

- $\mathbb{S}\llbracket \mathtt{Assume(c)} \rrbracket_{\mathbb{I}}$, where $c \in \mathit{expr}$
- $\mathbb{S}\llbracket v \leftarrow e \rrbracket_{\mathbb{I}}$, where $e \in \mathit{expr}$ and $v \in \mathcal{V}$
- $\mathbb{S}\llbracket \mathtt{remove}(v) \rrbracket_{\mathbb{I}}$, where $v \in \mathcal{V}$
- $\mathbb{S}\llbracket \mathtt{add}(v) \rrbracket_{\mathbb{I}}$, where $v \in \mathcal{V}$

we recall from Chapter 2 that $\mathbb{S}\llbracket \mathtt{stmt} \rrbracket_{\mathbb{I}}^{\mathcal{W}} \in \wp(\mathcal{W} \to \mathbb{I}) \to \wp(\mathcal{W} \to \mathbb{I})$ designates the concrete transformers operating on the lattice $\wp(\mathcal{W} \to \mathbb{I})$. For presentation purposes, whenever $R \in \mathfrak{M}$ and $\mathcal{W} \in \wp(\mathcal{V})$ we denote $R_{\mathcal{W}} \triangleq \{\rho \in R \mid \mathbf{def}(\rho) = \mathcal{W}\}$.

Definition 6.1 (Concrete semantics on $(\mathfrak{M}, \subseteq)$)**.** In the following $R \in \mathfrak{M}$:

$$\begin{aligned}
\mathbb{S}\llbracket \mathtt{Assume(c)} \rrbracket_{\mathbb{I}}(R) &= \bigcup_{\mathbf{fv}(c) \subseteq \mathcal{W} \subseteq \mathcal{V}} \mathbb{S}\llbracket \mathtt{Assume(c)} \rrbracket_{\mathbb{I}}^{\mathcal{W}}(R_{\mathcal{W}}) \\
\mathbb{S}\llbracket x \leftarrow e \rrbracket_{\mathbb{I}}(R) &= \bigcup_{(\{x\} \cup \mathbf{fv}(e)) \subseteq \mathcal{W} \subseteq \mathcal{V}} \mathbb{S}\llbracket x \leftarrow e \rrbracket_{\mathbb{I}}^{\mathcal{W}}(R_{\mathcal{W}}) \\
\mathbb{S}\llbracket \mathtt{remove}(v) \rrbracket_{\mathbb{I}}(R) &= \bigcup_{\mathcal{W} \subseteq \mathcal{V}} \{\rho_{|\mathcal{W} \setminus \{v\}} \mid \rho \in R_{\mathcal{W}}\} \\
\mathbb{S}\llbracket \mathtt{add}(v) \rrbracket_{\mathbb{I}}(R) &= \bigcup_{\mathcal{W} \subseteq \mathcal{V} \setminus \{v\}} \{\rho[v \mapsto i] \mid \rho \in R_{\mathcal{W}}, i \in \mathbb{I}\} \cup \bigcup_{\{v\} \subseteq \mathcal{W} \subseteq \mathcal{V}} R_{\mathcal{W}}
\end{aligned}$$

Remark 6.1. Please note that in Definition 6.1, environments undefined on variables appearing in a statement do not have a post-condition for the considered statement. We will show in the following Section 6.2.2 that this definition models the behavior of programs that manipulate heterogeneous environments.

6.2.2 Example of use

Program 6.5 is a small Python code snippet manipulating integer variables. Equations 6.1 to 6.10 provide the definitions of the set of reachable states at each program point, from ⓪ to

⑨. Using definitions from Definition 6.1, we solve this system and get the following:

$$S_{\textcircled{1}} = \mathbb{S}\llbracket \mathsf{a} \leftarrow \mathtt{0} \rrbracket_{\mathbb{Z}} \left(\bigcup_{\mathcal{W}\subseteq\mathcal{V}\setminus\{a\}} \{\rho[\mathsf{a} \mapsto \mathsf{i}] \mid \rho \in (\{()\})_{\mathcal{W}}, i \in \mathbb{Z}\} \cup \bigcup_{\{v\}\subseteq\mathcal{W}\subseteq\mathcal{V}} (\{()\})_{\mathcal{W}} \right) \tag{6.11}$$

$$= \mathbb{S}\llbracket \mathsf{a} \leftarrow \mathtt{0} \rrbracket_{\mathbb{Z}} \left(\{()[\mathsf{a} \mapsto \mathsf{i}] \mid i \in \mathbb{Z}\} \cup \emptyset\right) \tag{6.12}$$

$$= \mathbb{S}\llbracket \mathsf{a} \leftarrow \mathtt{0} \rrbracket_{\mathbb{Z}} \left(\{(\mathsf{a} \mapsto i) \mid i \in \mathbb{Z}\}\right) \tag{6.13}$$

$$= \bigcup_{(\{\mathsf{a}\}\cup\mathbf{fv}(\mathtt{0}))\subseteq\mathcal{W}\subseteq\mathcal{V}} \mathbb{S}\llbracket \mathsf{a} \leftarrow \mathtt{0} \rrbracket_{\mathbb{Z}}^{\mathcal{W}}(\{(\mathsf{a} \mapsto i) \mid i \in \mathbb{Z}\}_{\mathcal{W}}) \tag{6.14}$$

$$= \mathbb{S}\llbracket \mathsf{a} \leftarrow \mathtt{0} \rrbracket_{\mathbb{Z}}^{\{\mathsf{a}\}}(\{(\mathsf{a} \mapsto i) \mid i \in \mathbb{Z}\}) \tag{6.15}$$

$$= \{(\mathsf{a} \mapsto 0)\} \tag{6.16}$$

$$S_{\textcircled{2}} = S_{\textcircled{1}} \tag{6.17}$$

$$S_{\textcircled{3}} = \mathbb{S}\llbracket \mathsf{a} \leftarrow \mathtt{1} \rrbracket_{\mathbb{Z}} \circ \mathbb{S}\llbracket \mathsf{add(a)} \rrbracket_{\mathbb{Z}}(S_{\textcircled{2}}) \tag{6.18}$$

$$= \mathbb{S}\llbracket \mathsf{a} \leftarrow \mathtt{1} \rrbracket_{\mathbb{Z}} \left(\bigcup_{\mathcal{W}\subseteq\mathcal{V}\setminus\{\mathsf{a}\}} \{\rho[\mathsf{a} \mapsto \mathsf{i}] \mid \rho \in (\{(\mathsf{a} \mapsto 0)\})_{\mathcal{W}}, i \in \mathbb{Z}\} \cup \bigcup_{\{\mathsf{a}\}\subseteq\mathcal{W}\subseteq\mathcal{V}} (\{(\mathsf{a} \mapsto 0)\})_{\mathcal{W}} \right) \tag{6.19}$$

$$= \mathbb{S}\llbracket \mathsf{a} \leftarrow \mathtt{1} \rrbracket_{\mathbb{Z}} \left(\emptyset \cup \{(\mathsf{a} \mapsto 0)\}\right) \tag{6.20}$$

$$= \{(\mathsf{a} \mapsto 1)\} \tag{6.21}$$

$$S_{\textcircled{4}} = \mathbb{S}\llbracket \mathsf{b} \leftarrow \mathtt{1} \rrbracket_{\mathbb{Z}} \left(\bigcup_{\mathcal{W}\subseteq\mathcal{V}\setminus\{\mathsf{b}\}} \{\rho[\mathsf{b} \mapsto \mathsf{i}] \mid \rho \in (\{(\mathsf{a} \mapsto 1)\})_{\mathcal{W}}, i \in \mathbb{Z}\} \cup \bigcup_{\{\mathsf{b}\}\subseteq\mathcal{W}\subseteq\mathcal{V}} (\{(\mathsf{a} \mapsto 1)\})_{\mathcal{W}} \right) \tag{6.22}$$

$$= \mathbb{S}\llbracket \mathsf{b} \leftarrow \mathtt{1} \rrbracket_{\mathbb{Z}} \left(\{(\mathsf{a} \mapsto 1)[\mathsf{b} \mapsto \mathsf{i}] \mid i \in \mathbb{Z}\} \cup \emptyset\right) \tag{6.23}$$

$$= \{(\mathsf{a} \mapsto 1, \mathsf{b} \mapsto 1)\} \tag{6.24}$$

$$S_{\textcircled{6}} = \{(\mathsf{a} \mapsto 1, \mathsf{c} \mapsto 2)\} \tag{6.25}$$

$$S_{\textcircled{7}} = \{(\mathsf{a} \mapsto 1, \mathsf{b} \mapsto 1), (\mathsf{a} \mapsto 1, \mathsf{c} \mapsto 2)\} \tag{6.26}$$

$$S_{\textcircled{8}} = \mathbb{S}\llbracket \mathtt{Assume}(\mathsf{b} \leqslant 0) \rrbracket_{\mathbb{Z}}(S_{\textcircled{7}}) \tag{6.27}$$

$$= \bigcup_{\mathbf{fv}(\mathtt{b\ <=\ 5})\subseteq\mathcal{W}\subseteq\mathcal{V}} \mathbb{S}\llbracket \mathtt{Assume}(\mathsf{b} \leqslant 0) \rrbracket_{\mathbb{Z}}^{\mathcal{W}}(\{(\mathsf{a} \mapsto 1, \mathsf{b} \mapsto 1), (\mathsf{a} \mapsto 1, \mathsf{c} \mapsto 2)\}_{\mathcal{W}}) \tag{6.28}$$

$$= \mathbb{S}\llbracket \mathtt{Assume}(\mathsf{b} \leqslant 0) \rrbracket_{\mathbb{Z}}^{\{\mathsf{a},\mathsf{b}\}}(\{(\mathsf{a} \mapsto 1, \mathsf{b} \mapsto 1)\}) \tag{6.29}$$

$$= \{(\mathsf{a} \mapsto 1, \mathsf{b} \mapsto 1)\} \tag{6.30}$$

$$S_{\textcircled{9}} = \{(\mathsf{a} \mapsto 2, \mathsf{b} \mapsto 1)\} \tag{6.31}$$

This example emphasizes that our choice of semantics enables the description of Python behavior. Indeed Eq. 6.28 and 6.29 show that only environments containing variables appearing in expressions are kept. Moreover Eq. 6.18 to 6.21 underline the fact that $\mathsf{add}(v)$ only adds an unconstrained variable v in environment that do not yet contain v. This ensures that every variable assignment can be statically enriched with an $\mathsf{add}(v)$, without modifying the environments that do contain v.

In the following section we describe existing works that enable the representation of $\mathfrak{M}$.

6.3 State of the art

6.3.1 Partitioning

The most used technique for the representation of sets of numerical maps with heterogeneous supports is the use of partitionings [RM07]. Upon one of the numerical abstraction from Section 2.4 we build the following partitioning domain :

Definition 6.2 (Partitioning domain)**.** The partitioning domain is defined to be: $\mathcal{P}^\sharp = \{\top\} \cup \{m \in \Pi_{\mathcal{W}:\wp_f(\mathcal{V})} \mathcal{N}^{\mathcal{W}} \mid \mathbf{def}(m) \text{ is finite}\}$ with concretization:

$$\gamma_{\mathcal{P}^\sharp}(m) = \begin{cases} \mathfrak{M} & \text{if } m = \top \\ \bigcup_{\mathcal{W} \in \mathbf{def}(m)} \gamma^{\mathcal{W}}(m(\mathcal{W})) & \text{otherwise} \end{cases}$$

Moreover when every $\mathcal{N}^{\mathcal{W}}$ enjoys a Galois connection with the concrete (the abstraction function is then denoted: $\alpha^{\mathcal{W}}$) we can define an abstraction function:

$$\alpha_{\mathcal{P}^\sharp}(P) = \begin{cases} \biguplus_{\mathcal{W} \in \{\mathbf{def}(\rho) \mid \rho \in P\}} (\mathcal{W} \mapsto \alpha^{\mathcal{W}}(\{\rho \in P \mid \mathbf{def}(\rho) = \mathcal{W}\})) & \text{if } |\{\mathbf{def}(\rho) \mid \rho \in P\}| < \infty \\ \top & \text{otherwise} \end{cases}$$

Example 6.1. Consider the set of partial maps: $P = \{\begin{pmatrix} x \mapsto \alpha \\ y \mapsto \alpha + 1 \end{pmatrix} \mid \alpha \in \mathbb{R}\} \cup \{(x \mapsto \beta) \mid \beta \in \{0, 2\}\}$. We build a partitioning domain upon the octagon domain (which enjoys an abstraction function). Octagons are represented as conjunctions of linear constraints. This yields: $\alpha_{\mathcal{P}^\sharp}(P) = \begin{pmatrix} \{x\} \mapsto \{0 \leqslant x \wedge x \leqslant 2\} \\ \{x, y\} \mapsto \{y = x + 1\} \end{pmatrix}$. This can then be concretized to: $\gamma_{\mathcal{P}^\sharp}(\alpha_{\mathcal{P}^\sharp}(P)) = \{\begin{pmatrix} x \mapsto \alpha \\ y \mapsto \alpha + 1 \end{pmatrix} \mid \alpha \in \mathbb{R}\} \cup \{(x \mapsto \beta) \mid \beta \in [0, 2]\}$. We can see that a precision loss occurred due to the convexity of octagons.

Definition 6.3 (Order relation)**.** On the partitioning domain we define the following relation:

$$\begin{aligned} \sqsubseteq_{\mathcal{P}^\sharp} \overset{\Delta}{=} \{(m_1, m_2) \mid m_2 = \top \\ \vee (m_1 \neq \top \wedge \forall \mathcal{W} \in \mathbf{def}(m_1), \mathcal{W} \in \mathbf{def}(m_2) \wedge m_1(\mathcal{W}) \sqsubseteq^{\mathcal{W}} m_2(\mathcal{W})\} \end{aligned}$$

Example 6.2. Reusing elements from Example 6.1:
$\big(\{x, y\} \mapsto \{y = 1, x = 0\}\big) \sqsubseteq \begin{pmatrix} \{x\} \mapsto \{0 \leqslant x \wedge x \leqslant 2\} \\ \{x, y\} \mapsto \{y = x + 1\} \end{pmatrix}$

Proposition 6.1 (Galois connection)**.** *If for every $\mathcal{W} \in \wp(\mathcal{V})$ we have a Galois connection: $(\wp(\mathcal{W} \to \mathbb{I}), \subseteq) \xrightleftharpoons[\alpha^{\mathcal{W}}]{\gamma^{\mathcal{W}}} (N_{\mathcal{W}}, \sqsubseteq^{\mathcal{W}})$, then we have the following Galois connection: $(\mathfrak{M}, \subseteq) \xrightleftharpoons[\alpha_{\mathcal{P}^\sharp}]{\gamma_{\mathcal{P}^\sharp}} (\mathcal{P}^\sharp, \sqsubseteq_{\mathcal{P}^\sharp})$*

See proof on page 147.

Definition of abstract operators. Now that we have defined the abstract domain as well as its concretization, we provide abstract operators that are sound relative to this concretization. Please note that the concretization function is not only defined as a disjunction over sets of variables on which the partitioning is defined, but this disjunction is disjoint, indeed $\mathcal{W} \neq \mathcal{W}' \Rightarrow \gamma^{\mathcal{W}}(\dots) \cap \gamma^{\mathcal{W}'}(\dots) = \emptyset$. This disjointedness enables a straightforward definition of most abstract operators as the application of the operator on every underlying numerical element. As an example consider the two following definitions:

Definition 6.4 (Join operator)**.**

$$m_1 \sqcup m_2 \overset{\Delta}{=} \biguplus_{\mathcal{W} \in \mathbf{def}(m_1) \cup \mathbf{def}(m_2)} \left(\mathcal{W} \mapsto \begin{cases} m_1(\mathcal{W}) \sqcup^{\mathcal{W}} m_2(\mathcal{W}) & \text{if } \mathcal{W} \in \mathbf{def}(m_1) \cap \mathbf{def}(m_2) \\ m_1(\mathcal{W}) & \text{otherwise if } \mathcal{W} \in \mathbf{def}(m_1) \\ m_2(\mathcal{W}) & \text{otherwise if } \mathcal{W} \in \mathbf{def}(m_2) \end{cases} \right)$$

Definition 6.5 (Abstract transfer functions)**.**

$$\mathbb{S}^\sharp[\![\mathtt{stmt}]\!](m_1) \overset{\Delta}{=} \biguplus_{\mathcal{W} \in \mathbf{def}(m_1)} \Big(\mathcal{W} \mapsto \mathbb{S}^\sharp[\![\mathtt{stmt}]\!]^{\mathcal{W}}(m_1(\mathcal{W})) \Big)$$

As emphasized by the two previous definitions, the cost of performing one abstract operation on an abstract element m is the sum of the cost of performing the operation on every abstract element in the image of m. Let us assume that we want to represent a concrete environment containing n mandatory variables and p optional variables. Consider moreover the worst case scenario where we need to abstract environments containing all possible subsets of the optional variables, we get a map with an image of size 2^p. Therefore the cost of computing an abstract operation is exponential in the number of optional variables represented. Please note moreover that in the case of polyhedra, the cost of performing one operation on the underlying polyhedron is already exponential in the number of variables.

6.3.2 Encoding of optional variables via *avatars*

Liu and Rival in [LR15] provides an abstraction able to handle programs manipulating optional numerical variables. Their abstraction lifts any numerical domain built as a conjunction of constraints (which is the case for intervals, octagons, polyhedra). We present here – in a simplified way – the idea of their representation: for every optional variable $\mathtt{x}$, two variables $\mathtt{x}^+$ and $\mathtt{x}^-$, called *avatars* are allocated in the underlying numerical domain. If $\mathtt{x}^+$ and $\mathtt{x}^-$ can be concretized to the same value α then $\mathtt{x}$ is concretized to `None` or α, otherwise it is concretized to `None`. We recall here the definition of this concretization function, $\mathbb{X}$ denotes the set of optional variables, $\mathbb{X}^+$ (resp. $\mathbb{X}^-$) is the set of variables x^+ (resp. x^-) when $\mathtt{x} \in \mathbb{X}$, finally $\mathbb{Y}$ denotes the set of non-optional variables.

$$\begin{aligned}\gamma(N^\sharp) \triangleq \{\rho \mid \exists \tau \in \gamma^{\mathbb{Y}\cup\mathbb{X}^+\cup\mathbb{X}^-}(N^\sharp), \forall y \in \mathbb{Y}, \rho(Y) = \tau(Y) \wedge \\ \forall x \in \mathbb{X}, (\rho(x) = \tau(x^+) = \tau(x^-) \vee \rho(x) = \texttt{None})\}\end{aligned}$$

As an example, consider Program 6.6, assuming the abstraction is built upon the polyhedron domain, the abstraction computed for program point ● (at line 5) is $\{\mathtt{x}^+ = 1, \mathtt{x}^- = 0, \mathtt{y} = 0\}$, the concretization of which is $\{(\mathtt{x} \mapsto \texttt{None}, \mathtt{y} \mapsto 0)\}$, as $\mathtt{x}^+$ and $\mathtt{x}^-$ could not be concretized to the same value. At program point ★ (at line 7), the computed abstraction is $\{y = \mathtt{x}^+, y = \mathtt{x}^-\}$, which concretization is: $\{(\mathtt{x} \mapsto \texttt{None}, \mathtt{y} \mapsto \alpha) \mid \alpha \in \mathbb{Z}\} \cup \{(\mathtt{x} \mapsto \alpha, \mathtt{y} \mapsto \alpha) \mid \alpha \in \mathbb{Z}\}$. The abstraction provided in this chapter diverges from the one introduced by Liu and Rival on the following points:

- As emphasized by the previous example, their abstraction cannot express that an optional variable is never `None` at some program point
- They are able to express some relations between the “support” (in their case: the set of variables not mapped to `None`) of the represented maps with the numerical values associated to variables. This can not be done in the abstraction presented thereafter
- In their case variables must be defined as optional or non-optional. Setting all variables to optional to represent Python-like behaviors, would induce a huge overhead as the number of variables would be doubled.

6.3.3 A folk technique for non relational domains

We now present a classical technique used in analyzers where the underlying domain is non relational. To simplify, assume that the abstract element is a map m, associating to each variable v a set of potential concrete value $m(v)$. Consider now the three following concretization functions:

$$\begin{aligned}\gamma_{\supseteq}(m) &= \{\rho \in \mathfrak{M} \mid \mathbf{def}(\rho) \supseteq \mathbf{def}(m) \wedge \forall v \in \mathbf{def}(m),\ \rho(v) \in m(v)\}\\ \gamma_{=}(m) &= \{\rho \in \mathfrak{M} \mid \mathbf{def}(\rho) = \mathbf{def}(m) \wedge \forall v \in \mathbf{def}(m),\ \rho(v) \in m(v)\}\\ \gamma_{\subseteq}(m) &= \{\rho \in \mathfrak{M} \mid \mathbf{def}(\rho) \subseteq \mathbf{def}(m) \wedge \forall v \in \mathbf{def}(m),\ \rho(v) \in m(v)\}\end{aligned}$$

```
int option x;
int y = 0;

if (?) {
  x = None; ●
} else {
  y = x; ⬟
}
```

Program 6.6: Optional variable manipulations

$\gamma_\supseteq$, $\gamma_=$ and $\gamma_\subseteq$ are three extensions of the classical interval concretization. $\gamma_\supseteq$ does not provide any constraints on variables to which no interval was associated. On the contrary, $\gamma_=$ enforces environments to be defined precisely on the definition set of the interval. Finally $\gamma_\subseteq$ allows environments to be defined only on a subset of the support of the interval map. Contrary to $\gamma_=$, this allows the representation of heterogeneous maps, and contrary to $\gamma_\supseteq$ provides constraints on all variable present in the support.

We will now illustrate the shortcomings of $\gamma_\supseteq$ and $\gamma_=$. Let us consider program 6.7, written in Python. The top-down analysis of this program yields, at program point ●, the abstractions $S^\sharp_{\bullet,=} = S^\sharp_{\bullet,\supseteq} = S^\sharp_{\bullet,\subseteq} = (x \mapsto [1;1])$. At program point ⬟ we get: $S^\sharp_{\text{⬟},=} = S^\sharp_{\text{⬟},\supseteq} = S^\sharp_{\text{⬟},\subseteq} = (x \mapsto [0;0], y \mapsto [2;2])$ Let us now consider the possible post-conditions for the inner-most `if` statement for each semantics.

- For the analysis with semantics $\gamma_\supseteq$: The post-condition needs to be an over-approximation of $\gamma_\supseteq(S^\sharp_{\bullet,\supseteq}) \cup \gamma_\supseteq(S^\sharp_{\text{⬟},\supseteq}) = \{\rho \mid \exists \rho', \rho = (x \mapsto 1) \uplus \rho' \vee \rho = (x \mapsto 1, y \mapsto 2) \uplus \rho'\} = \{(x \mapsto 1, \dots)\}$. Therefore the strongest possible post-conditions is $(x \mapsto [1;1])$. We lost all information on y.
- For the analysis with semantics $\gamma_=$: We need to provide an over-approximation of the concrete state $\gamma_=(S^\sharp_{\bullet,=}) \cup \gamma_=(S^\sharp_{\text{⬟},=}) = \{(x \mapsto 1), (x \mapsto 0, y \mapsto 2)\}$, however we have seen that $\gamma_=$ can only represent precisely sets of maps with homogeneous supports. For soundness reasons, the only possible post-conditions is therefore $\top$. Loosing information on both x and y.
- For the analysis with semantics $\gamma_\subseteq$: We need to provide an over-approximation of the concrete state $\gamma_\subseteq(S^\sharp_{\bullet,\subseteq}) \cup \gamma_\subseteq(S^\sharp_{\text{⬟},\subseteq}) = P \triangleq \{(), (y \mapsto 2), (x \mapsto 1), (x \mapsto 0), (x \mapsto 0, y \mapsto 2)\}$. This can be represented by: $(x \mapsto [0;1], y \mapsto [2;2])$, with concretization $P \cup \{(x \mapsto 1, y \mapsto 2)\}$. Note that the analysis induced over-approximations but was more precise than the two other semantics.

The abstract union operation for the $\gamma_\subseteq$ semantics comes very naturally from the usual join of intervals: when joining two abstract environments m and m', (1) if a variable v is bound in m and in m', it is bound in the result with the join of $m(v)$ and $m'(v)$; (2) if a variable v is bound in only one of the input (*w.l.o.g.* say m), it is bound in the result with $m(v)$; (3) finally, if it is unbound in both operands, it is unbound in the result.

For all the aforementioned reasons $\gamma_\subseteq$ has been used as the canonical extensions of intervals to heterogeneous environments.

We used this technique as a starting point for the definition of a new abstraction used for the representation of heterogeneous maps. Let us therefore use the opportunity of the presentation of this non-relational approach to introduce our new abstraction. This abstraction will be not only more precise than $\gamma_\subseteq$, but it will also generalize the method to relational domains. Indeed the join operation provided in the previous paragraph can be trivially generalized to any

```
x = 0
if (?):
    if (?):
        x = 1 ●
    else:
        y = 2 ⬟
else:
    if (?):
        x = 3
    else:
        z = 4
```

Program 6.7: Heterogeneous Python environments

non-relational domain, however as we will see, lifting to non-relational domains is not straightforward.

Support abstraction. Even though abstraction $\gamma_\subseteq$ grossly over-approximates the set of accepted supports, we have shown that it was able to provide interesting numerical constraints. Our first improvement will be to enrich the interval environment with an abstraction for supports that provides more information that an upper-bound. The set of supports is a set of sets of variables, that is a set of elements from $\wp(\mathcal{V})$. Therefore we choose a simple lower-bound/upper-bound abstraction. As an example, the abstraction for program point ● will be: $S^\sharp_{●} = \langle (\mathsf{x} \mapsto [1;1]), \{\mathsf{x}\}, \{\mathsf{x}\}\rangle$, and for program point ⬟: $S^\sharp_{⬟} = \langle (\mathsf{x} \mapsto [0;0], \mathsf{y} \mapsto [2;2]), \{\mathsf{x},\mathsf{y}\}, \{\mathsf{x},\mathsf{y}\}\rangle$, the join of which will be $X^\sharp = \langle (\mathsf{x} \mapsto [0;1], \mathsf{y} \mapsto [2;2]), \{\mathsf{x}\}, \{\mathsf{x},\mathsf{y}\}\rangle$. In $X^\sharp$, $\{\mathsf{x}\}$ is a lower-bound of the set of supports of the represented maps, whereas $\{\mathsf{x},\mathsf{y}\}$ is an upper-bound. Hence variable x is present in every environment, and y is optional. The concretization of $X^\sharp$ will be: $\{(\mathsf{x} \mapsto 0, \mathsf{y} \mapsto 2), (\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2), (\mathsf{x} \mapsto 0), (\mathsf{x} \mapsto 1)\}$. This is more precise than the classical approach. The classical approach $\gamma_\subseteq$ is the special case where the lower-bound is always set to $\emptyset$ and the upper-bound to the definition set of the interval map of the abstract element. The analysis of the other inner-most `if` would yield the abstraction $(\mathsf{x} \mapsto [0;3], \mathsf{z} \mapsto [5;5])$, hence the post-condition of the entire Program 6.7 would yield: $Y^\sharp = \langle (\mathsf{x} \mapsto [0;3], \mathsf{y} \mapsto [2;2], \mathsf{z} \mapsto [4;4]), \{\mathsf{x}\}, \{\mathsf{x},\mathsf{y},\mathsf{z}\}\rangle$. Indeed when computing a join we will unite the upper-bounds and intersect the lower-bounds. Once more, note that this induces a support over-approximation, indeed the concretization of $Y^\sharp$ contains the environment $(\mathsf{x} \mapsto 0, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 4)$, even though no concrete environment reachable in Program 6.7 is defined on $\{\mathsf{x},\mathsf{y},\mathsf{z}\}$.

6.4 The Clip abstraction

6.4.1 Abstraction definition

In order to abstract sets of maps with heterogeneous definition sets, we start by abstracting the potential definition set. We choose a simple lower-bound/upper-bound abstraction ($\mathfrak{l}$ and $\mathfrak{u}$ in the following definition). Moreover we need to abstract the potential mappings given a definition set: this is done using a classical numerical domain. Contrary to partitioning, we will use only one numerical abstract element, defined on the upper-bound $\mathfrak{u}$, to represent all environments (instead of one abstract element for each definition set). We also add a $\top$ element, used in

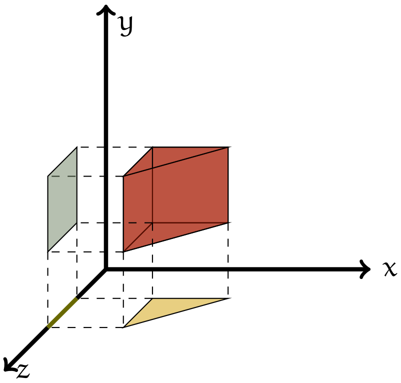


Figure 6.1: Projections between $\{z\}$ and $\{x, y, z\}$

the case where the number of different supports that need to be represented is infinite. As the abstract numerical element component of our abstraction is given meaning across projections on all variable sets in an interval from the variable powerset lattice, the abstraction defined in this section will be referred hereinafter as the Clip abstraction (CLosed under Interval Projection).

Definition 6.6 (Abstract elements)**.** We define the following set: $\mathfrak{M}^\sharp \triangleq \{\langle N^\sharp, l, u\rangle \mid l, u \in \wp_f(\mathcal{V}) \wedge l \subseteq u \wedge N^\sharp \in \mathcal{N}^u \wedge N^\sharp \neq \bot^u\} \cup \{\top, \bot\}$. An element of $\mathfrak{M}^\sharp$ is therefore: either $\top$, $\bot$ or a triple $\langle N^\sharp, l, u\rangle$ where l and u are finite sets of variables such that $N^\sharp$ is defined over u.

Definition 6.7 (Concretization function)**.** Abstract elements from $\mathfrak{M}^\sharp$ are concretized to $\mathfrak{M}$ thanks to the following concretization function:

$$\gamma(\bot) = \emptyset \quad \gamma(\top) = \mathfrak{M}$$
$$\gamma(\langle N^\sharp, l, u\rangle) = \{\rho \in \mathcal{W} \to \mathbb{I} \mid l \subseteq \mathcal{W} \subseteq u \wedge \rho \in \gamma^u(N^\sharp)_{|\mathcal{W}}\}$$

Example 6.3. As an example consider $\gamma(\langle\{x = y, x \leqslant 3, z = 0\}, \{x\}, \{x, y, z\}\rangle) = \{(x \mapsto a) \mid a \leqslant 3\} \cup \{(x \mapsto a, y \mapsto a) \mid a \leqslant 3\} \cup \{(x \mapsto a, z \mapsto 0) \mid a \leqslant 3\} \cup \{(x \mapsto a, y \mapsto a, z \mapsto 0) \mid a \leqslant 3\}$. As intended, the resulting set of maps contains maps with different definition sets.

Example 6.4. Figure 6.1 depicts the octagon (in ●): $N^\sharp = \{x + z \leqslant 3, x \geqslant 1, z \geqslant 1\}$. The concretization of the abstract element $\langle N^\sharp, \{z\}, \{x, y, z\}\rangle$ is the union: ● $\cup$ ● $\cup$ ● $\cup$ ●.

The concretization function from Definition 6.7 defines the set of represented maps as the union of several projections of some numerical abstract element. Therefore the definition of the following set abstract operators will need the ability to perform the test: given $N^\sharp \in \mathcal{N}^u$ and $N^{\sharp\prime} \in \mathcal{N}^{u'}$, with $u \subseteq u'$ does $\gamma^u(N^\sharp) \subseteq \gamma^{u'}(N^{\sharp\prime})_{|u}$. Indeed $\gamma(\langle N^\sharp, l, u\rangle) \subseteq \gamma(\langle N^{\sharp\prime}, l', u'\rangle)$ as soon as $l' \subseteq l \subseteq u \subseteq u'$ and $\gamma^u(N^\sharp) \subseteq \gamma^{u'}(N^{\sharp\prime})_{|u}$. As this test is not necessarily computable, we will need to define an abstract operator called *proj-inclusion* (for projection and inclusion), denoted $|\sqsubseteq$. We will say that the operator is sound whenever $\forall u \subseteq u'$, $\forall N^\sharp \in \mathcal{N}^u, N^{\sharp\prime} \in \mathcal{N}^{u'}, N^\sharp \mathrel{|\sqsubseteq} N^{\sharp\prime} \Rightarrow \gamma^u(N^\sharp) \subseteq \gamma^{u'}(N^{\sharp\prime})_{|u}$. The following subsection provides the definition of such an abstract operator.

6.4.2 Composing inclusion and projection

We assume in this section that the projection operator of the underlying domain is exact, it is the case for intervals and octagons over $\mathbb{Z}$ or $\mathbb{Q}$ and for polyhedra over $\mathbb{Q}$. The soundness of the projection operator ensures that it is always true that $\gamma^{\mathcal{W}}(N^\sharp)_{|\mathcal{W}'} \subseteq \gamma^{\mathcal{W}'}(N^\sharp_{|\mathcal{W}'})$, the exactness of the projection operator is stronger and ensures that $\forall \mathcal{W}, \mathcal{W}' \in \wp_f(\mathcal{V}), \forall N^\sharp \in \mathcal{N}^{\mathcal{W}}, \gamma^{\mathcal{W}}(N^\sharp)_{|\mathcal{W}'} = \gamma^{\mathcal{W}'}(N^\sharp_{|\mathcal{W}'})$. The following definition and proposition motivates this assumption.

Definition 6.8 (Proj-incl operator)**.** Given $N^\sharp \in \mathcal{N}^u$ and $N^{\sharp\prime} \in \mathcal{N}^{u'}$, with $u \subseteq u'$, we define the following proj-incl operator: $N^\sharp \mathrel{|\sqsubseteq} N^{\sharp\prime} \triangleq N^\sharp \sqsubseteq^u N^{\sharp\prime}_{|u}$.

Proposition 6.2 (Soundness of $|\sqsubseteq$)**.** *Whenever the abstract projection operator $\Box_{|\Box}$ is exact, the proj-incl operator from Definition 6.8 is sound.*

See proof on page 147.

Remark 6.2. Let us note that the precision of the operators defined in the rest of the Clip abstraction depends greatly upon the precision of the $|\sqsubseteq$ operator. The version defined in Definition 6.8 albeit precise is not sound as soon as the underlying projection operator is not exact, see examples in Section 6.5. The alternative definition: $N^\sharp \mathrel{|\sqsubseteq} N^{\sharp\prime} \triangleq N^\sharp_{|u'} \sqsubseteq^{u'} N^{\sharp\prime}$ is always sound[1], however the Clip abstraction will not be able to retrieve constraints on optional variables.

Using this newly defined $|\sqsubseteq$ operator we can move on to the definition of the abstract set operators of the Clip abstraction.

6.4.3 Operator definitions

Abstract inclusion test

We start by providing a comparison operator on $\mathfrak{M}^\sharp$. As we ensured that, in a triple $\langle N^\sharp, l, u\rangle$, $N^\sharp \neq \bot^u$ we have $\forall l \subseteq \mathcal{W} \subseteq u, \gamma^u(N^\sharp)_{|\mathcal{W}} \neq \emptyset$. Hence a necessary condition for $\gamma(\langle N^\sharp, l, u\rangle) \subseteq \gamma(\langle N^{\sharp\prime}, l', u'\rangle)$ is that $l' \subseteq l \subseteq u \subseteq u'$. The numerical inclusion can then be tested with the $|\sqsubseteq$ operator from previous section.

Definition 6.9 (Order relation $\sqsubseteq$)**.** On $\mathfrak{M}^\sharp$ we define the following comparison operator:

$$\langle N^\sharp, l, u\rangle \sqsubseteq \langle N^{\sharp\prime}, l', u'\rangle \overset{\Delta}{\Leftrightarrow} l' \subseteq l \subseteq u \subseteq u' \wedge N^\sharp \mathrel{|\sqsubseteq} N^{\sharp\prime}$$

this comparison is trivially extended to $\top$ (resp. $\bot$) as being the largest (resp. smallest) element in $\mathfrak{M}^\sharp$.

Proposition 6.3 (Soundness of $\sqsubseteq$)**.** *$\sqsubseteq$ is a sound abstraction of $\subseteq$. This means that $X^\sharp \sqsubseteq Y^\sharp \Rightarrow \gamma(X^\sharp) \subseteq \gamma(Y^\sharp)$. Moreover whenever the comparison operator of the underlying domain is complete, $\sqsubseteq$ is complete as well, which means that $X^\sharp \sqsubseteq Y^\sharp \Leftarrow \gamma(X^\sharp) \subseteq \gamma(Y^\sharp)$.*

See proof on page 147.

Example 6.5. Consider the following abstract element $X^\sharp \triangleq \langle\{x = y, x \leqslant 3, z = 0\}, \{x\}, \{x, y, z\}\rangle$, introduced in Example 6.3. Consider moreover $Y^\sharp \triangleq \langle\{x \leqslant y, y \leqslant 2\}, \{x, y\}, \{x, y\}\rangle$ representing the concrete set of environments: $\{(x \mapsto \alpha, y \mapsto \beta) \mid \alpha \leqslant \beta \wedge \beta \leqslant 2\}$. We have $\{x\} \subseteq \{x, y\} \subseteq \{x, y\} \subseteq \{x, y, z\}$. Moreover $\{x = y, x \leqslant 3, z = 0\}_{|\{x,y\}} = \{x = y, x \leqslant 3\}$ and as $\{x \leqslant y, y \leqslant 2\} \sqsubseteq^{\{x,y\}} \{x = y, x \leqslant 3\}$ we have $Y^\sharp \sqsubseteq X^\sharp$.

[1] The main difference with the definition from Definition 6.8 is the fact that the abstract operation pre inclusion test is applied on the left operand, which can be approximated while preserving the soundness (but loosing precision) of the overall operator. Whereas over-approximating the right operand is unsound before an inclusion test, more details are provided in Section 6.5

Algorithm 6.1: strengthening operator

```
Input  : $N^{\sharp\prime}$, C: a set of constraints, $N_1^\sharp \in N_u$: a soundness threshold on environment $u$,
         $N_2^\sharp \in N_v$: a soundness threshold on environment $v$
Output: res an abstract element over-approximating $N_1^\sharp$ on $u$ and $N_2^\sharp$ on $v$
res ← $N^{\sharp\prime}$;
foreach $c \in C$ do
    test ← $\mathbb{S}^\sharp[\![\texttt{Assume(c)}]\!]^{u \cup v}$(res);
    if $N_1^\sharp \mathrel{|\sqsubseteq} \texttt{test} \wedge N_2^\sharp \mathrel{|\sqsubseteq} \texttt{test}$ then
        res ← test;
    end
return res;
```

Abstract union operator

We now need to define a join operator $\sqcup$ that abstracts the classic set operator $\cup$. The possible environments resulting from the union of two abstract element $\langle \_, l, u \rangle$ and $\langle \_, l', u' \rangle$ are environments defined at least on $l \cap l'$ and at most on $u \cup u'$, therefore the result from this join operation will be of the form $\langle \_, l \cap l', u \cup u' \rangle$, note however that this might induce an over-approximation of the set of represented environments. Section 6.6 is dedicated to the definition of an abstraction aiming at removing this over-approximation.

The numerical component can not be computed by simply applying the join operator from the underlying domain on both abstract component as they might have different definition sets. A first sound, naive solution would be to extend their respective definition set and to perform the abstract operation on the resulting elements: $N^\sharp_{1\ |u \cup u'} \sqcup^{u \cup u'} N^\sharp_{2\ |u \cup u'}$. However consider $U_1^\sharp = \langle \{x = y\}(= N_1^\sharp), \{x, y\}, \{x, y\} \rangle$ and $U_2^\sharp = \langle \{x = z\}(= N_2^\sharp), \{x, z\}, \{x, z\} \rangle$, where the underlying domain is the octagon domain over $\mathbb{Q}$ where elements are represented as a set of linear constraints (e.g. $\{x = y\}$). The concretization of $U_1^\sharp$ is $\{(x \mapsto \alpha, y \mapsto \alpha) \mid \alpha \in \mathbb{Q}\}$, the concretization of $U_2^\sharp$ is $\{(x \mapsto \alpha, y \mapsto \alpha) \mid \alpha \in \mathbb{Q}\}$. We have $N^\sharp_{1\ |\{x,y,z\}} = \{x = y\}$ and $N^\sharp_{2\ |\{x,y,z\}} = \{x = z\}$, hence $N^\sharp_{1\ |\{x,y,z\}} \sqcup^{\{x,y,z\}} N^\sharp_{2\ |\{x,y,z\}} = \top^{\{x,y,z\}}$. Therefore using this as the numerical component would yield the abstract element $\langle \top^{\{x,y,z\}}, \{x\}, \{x, y, z\} \rangle$ the concretization of which is the whole set of numerical maps defined on any set between $\{x\}$ and $\{x, y, z\}$. Consider now the abstract element in $\mathfrak{M}^\sharp$: $U_3^\sharp = \langle \{x = y, x = z\}(= N_3^\sharp), \{x\}, \{x, y, z\} \rangle$. The concretization of $U_3^\sharp$ over-approximates the union of the concretization of $U_1^\sharp$ and $U_2^\sharp$, and its numerical component is more precise than $\top$. We note that the numerical constraints appearing in $N_3^\sharp$ could be found in $N_1^\sharp$ or $N_2^\sharp$, therefore in order to remove the aforementioned imprecision we define a refined abstract union operator, denoted as $\boxed{\star}$, that uses constraints found in the inputs in order to refine its result. This is done using the **strenghtening** operator of Algorithm 6.1 which adds constraints from C that do not make the projection of $N^{\sharp\prime}$ to u (resp. v) lower than the threshold $N_1^\sharp$ (resp. $N_2^\sharp$).

Definition 6.10 ($\boxed{\star}$ operator). Let $N_1^\sharp \in N_u$, $N_2^\sharp \in N_v$ be two numerical environments, let $\mathfrak{c} = u \cap v$ we define:

$$
\begin{aligned}
N_1^\sharp \boxed{\star} N_2^\sharp = &\ \textbf{let } N^{\sharp\prime} = (N^\sharp_{1\ |\mathfrak{c}} \sqcup^{\mathfrak{c}} N^\sharp_{2\ |\mathfrak{c}})_{|u \cup v} \textbf{ in} \\
&\ \textbf{let } C = \textbf{constraints}(N_1^\sharp) \cup \textbf{constraints}(N_2^\sharp) \textbf{ in} \\
&\ \textbf{strengthening}(N^{\sharp\prime}, C, N_1^\sharp, N_2^\sharp)
\end{aligned}
$$

Remark 6.3.
- The relative precision of $\boxed{\star}$ depends upon the order of iteration over constraints $c \in C$ in Algorithm 6.1. Our implementation currently iterates in the order in which constraints are returned from the abstract domains. More clever heuristics will be considered in future work.

- No hypothesis on the set of constraints given to **strenghtening** is required to ensure soundness. We chose the set of constraints found in the arguments of the join as a first heuristic. The more constraints are given to **strengthening**, the more precise the result will be.
- $N_1^\sharp \boxed{\star} N_2^\sharp$ starts by performing the join over the domain $\mathfrak{c}$, the result is then strengthened. Other **strenghtening**$(N^{\sharp\prime}, N_1^\sharp \in N_u, N_2^\sharp \in N_v)$ operator could be defined, however in order to ensure soundness of $\boxed{\star}$, it must satisfy the two following constraints for every set of constraints C:

$$N_1^\sharp \sqsubseteq^u \textbf{strenghtening}(N^{\sharp\prime}, C, N_1^\sharp, N_2^\sharp)$$
$$N_2^\sharp \sqsubseteq^v \textbf{strenghtening}(N^{\sharp\prime}, C, N_1^\sharp, N_2^\sharp)$$

Example 6.6. Let us now consider the example introduced beforehand: $N_1^\sharp \boxed{\star} N_2^\sharp = \{x = y, y = z\} \in N_{\{x,y,z\}}$. Indeed using the notations of Definition 6.10: $\texttt{res} \triangleq N^{\sharp\prime} = \top \in N_{\{x,y,z\}}$, $C = \{x = y, y = z\}$, moreover $\mathbb{S}^\sharp[\![\texttt{Assume}(x = y)]\!]^{u \cup v}(\top) = \{x = y\}(\triangleq \texttt{test})$, $N_1^\sharp \sqsubseteq^{\{x,y\}} \{x = y\} = \texttt{test}_{|\{x,y\}}$ and $N_2^\sharp \sqsubseteq^{\{x,z\}} \top = \texttt{test}_{|\{x,z\}}$. Therefore constraint $x = y$ is added to $\texttt{res}$. At the next loop iteration: $\mathbb{S}^\sharp[\![\texttt{Assume}(x = z)]\!]^{u \cup v}(\{x = y\}) = \{x = y, x = z\}(\triangleq \texttt{test})$, $N_1^\sharp \sqsubseteq^{\{x,y\}} \{x = y\} = \texttt{test}_{|\{x,y\}}$ and $N_2^\sharp \sqsubseteq^{\{x,z\}} \{x = z\} = \texttt{test}_{|\{x,z\}}$. Therefore constraint $x = z$ is added to $\texttt{res}$.

Proposition 6.4 (Soundness of $\boxed{\star}$). *let* $N_1^\sharp \in N_u$ *and* $N_2^\sharp \in N_v$, *then* $\gamma^u(N_1^\sharp) \subseteq \gamma^{u \cup v}(N_1^\sharp \boxed{\star} N_2^\sharp)_{|u}$ *and* $\gamma^v(N_2^\sharp) \subseteq \gamma^{u \cup v}(N_1^\sharp \boxed{\star} N_2^\sharp)_{|v}$

See proof on page 148.

Using the $\boxed{\star}$ operator we can now define an operator abstracting the union.

Definition 6.11 (Union abstract operators: $\sqcup$). We define the following abstract set operator:

$$\langle N^\sharp, l, u \rangle \sqcup \langle N^{\sharp\prime}, l', u' \rangle \triangleq \langle N^\sharp \boxed{\star} N^{\sharp\prime}, l \cap l', u \cup u' \rangle$$

Example 6.7. Following Example 6.6, let us compute the abstract union of $U_1^\sharp = \langle \{x = y\}, \{x, y\}, \{x, y\} \rangle$ and $U_2^\sharp = \langle \{x = z\}, \{x, z\}, \{x, z\} \rangle$. As mentioned in Example 6.6 we have $\{x = z\} \boxed{\star} \{x = y\} = \{x = y = z\}$. Hence we have $U_1^\sharp \sqcup U_2^\sharp = \langle \{x = y = z\}, \{x\}, \{x, y, z\} \rangle$, the concretization of which is $\{(x \mapsto \alpha), (x \mapsto \alpha, y \mapsto \alpha), (x \mapsto \alpha, z \mapsto \alpha), (x \mapsto \alpha, y \mapsto \alpha, z \mapsto \alpha) \mid \alpha \in \mathbb{I}\}$. Note that the join induced an over-approximation over the definition set, but not over the numerical constraints, as intended.

Proposition 6.5 (Soundness of $\sqcup$). *The* $\sqcup$ *operator over-approximates the union:* $\forall U_1^\sharp, U_2^\sharp, \gamma(U_1^\sharp) \subseteq \gamma(U_1^\sharp \sqcup U_2^\sharp)$ *and* $\gamma(U_2^\sharp) \subseteq \gamma(U_1^\sharp \sqcup U_2^\sharp)$

See proof on page 148.

Remark 6.4. The $\boxed{\star}$ operator is costly: it computes a projection and an inclusion test for every constraints appearing in its inputs. We now present an alternative definition of this operator that reduces this overhead cost while retaining the same level of precision, this was not introduced as the $\boxed{\star}$ operator as it is more complicated and does not convey the idea as well.

Definition 6.12 ($\boxast_2$ operator)**.** Using the notation $\mathfrak{c} = u \cap v$, we define the following operator:

```
$N_1^\sharp \boxast_2 N_2^\sharp =$ if $N_1^\sharp = \bot^u$ then
        $N_2^\sharp$
    else if $N_2^\sharp = \bot^v$ then
        $N_1^\sharp$
    else
        let $N^{\sharp\prime} = (N_1^\sharp{}_{|\mathfrak{c}} \sqcup^{\mathfrak{c}} N_2^\sharp{}_{|\mathfrak{c}})_{|u \cup v} \sqcap^{u \cup v} (N_1^\sharp{}_{|u \setminus \mathfrak{c}})_{|u \cup v} \sqcap^{u \cup v} (N_2^\sharp{}_{|v \setminus \mathfrak{c}})_{|u \cup v}$ in
        let $C = \{c \in \mathbf{constraints}(N_1^\sharp) \mid \mathbf{fv}(c) \cap u \neq \emptyset \wedge \mathbf{fv}(c) \cap (u \setminus \mathfrak{c}) \neq \emptyset\}$
            $\cup \{c \in \mathbf{constraints}(N_2^\sharp) \mid \mathbf{fv}(c) \cap v \neq \emptyset \wedge \mathbf{fv}(c) \cap (v \setminus \mathfrak{c}) \neq \emptyset\}$ in
        $\mathbf{strengthening}(N^{\sharp\prime}, C, N_1^\sharp, N_2^\sharp)$
```

Proposition 6.6 (Soundness of $\boxast_2$)**.** *let* $N_1^\sharp \in N_u$ *and* $N_2^\sharp \in N_v$*, then* $\gamma^u(N_1^\sharp) \subseteq \gamma^{u \cup v}(N_1^\sharp \boxast_2 N_2^\sharp)_{|u}$ *and* $\gamma^v(N_2^\sharp) \subseteq \gamma^{u \cup v}(N_1^\sharp \boxast_2 N_2^\sharp)_{|v}$

See proof on page 149.

The idea behind $\boxast_2$ is the following: there are 3 kinds of constraints in $N_1^\sharp$: (1) those that relate only variables from $u \cap v$; (2) those that relate only variables from $u \setminus (u \cap v)$; (3) those that relate at least one variable from $u \cap v$ and one variable from $u \setminus (u \cap v)$. In the definition of $\boxast$, only constraints from 1 can be found in $N^{\sharp\prime}$, others must be salvaged using the **strenghtening** operator. In the definition of $\boxast_2$, $N^{\sharp\prime}$ contains not only constraints from 1, but also constraints from 2 thanks to the two $\sqcap$ computations. Finally we only have to try to add back constraints that contain at least one variable in $u \cap v$ and one in $u \setminus (u \cap v)$. In the special case where the underlying numerical domain is non relational, $\boxast_2$ amounts to: computing the point-wise join on variables shared by the two abstract elements, keeping the abstract value for every variable that does not appear in the other abstract element. This is what was usually done to handle heterogeneous environments for non relational numerical domains. Moreover consider the special case where the two abstract elements are defined over the same set of variables. We have $u = v = \mathfrak{c}$ in Definition 6.12, the definition falls back to the classical join: $\sqcup^u$, as the set C is empty, $u \setminus \mathfrak{c} = \emptyset$ and $v \setminus \mathfrak{c} = \emptyset$.

Abstract intersection operator

Let us now define an operator abstracting the intersection. Given two abstract elements $\langle N_1^\sharp, l_1, u_1\rangle$ and $\langle N_2^\sharp, l_2, u_2\rangle$, they represent maps, the definition sets of which are between l_1 and u_1, resp. l_2 and u_2. Therefore the maps that are represented by both abstract elements have definition sets greater than both l_1 and l_2 and lower than both u_1 and u_2. Our abstract operator, over-approximating the intersection will therefore yield an abstract element of the form $\langle \_, l_1 \cup l_2, u_1 \cap u_2\rangle$. Before proceeding to the definition, we insist on the reasons why the numerical component can not be $N_1^\sharp{}_{|u_1 \cap u_2} \sqcap^{u_1 \cap u_2} N_2^\sharp{}_{|u_1 \cap u_2}$, which is unsound, for the meet computation. Consider the two following abstract elements: $U_1^\sharp = \langle \{y = x, x \geqslant -2, x \leqslant 1\}, \{x\}, \{x, y\}\rangle$ and $U_2^\sharp = \langle \{y = -x, x \geqslant -1, x \leqslant 2\}, \{x\}, \{x, y\}\rangle$. Figure 6.2 provides an illustration of the concretization of these two abstract elements: $\gamma(U_1^\sharp) = \{(x \mapsto \alpha), (x \mapsto \alpha, y \mapsto \alpha) \mid \alpha \in [-1; 2]\}$ and $\gamma(U_2^\sharp) = \{(x \mapsto \alpha), (x \mapsto \alpha, y \mapsto -\alpha) \mid \alpha \in [-2; 1]\}$. The concrete intersection is therefore: $\{(x \mapsto \alpha) \mid \alpha \in [-1; 1]\} \cup \{(x \mapsto 0, y \mapsto 0)\}$. Therefore the result from our abstract operation must be a polyhedron defined on $\{x, y\}$ such that: (1) it contains the point $(0, 0)$ (2) its projection on $\{x\}$ contains the interval $[-1; 1]$. We see that computing the intersection of the two polyhedra on $\{x, y\}$ will yield the polyhedron $\{x = 0, y = 0\}$ which satisfies (1) but does not satisfy (2).

Figure 6.2: Introductory example for intersection

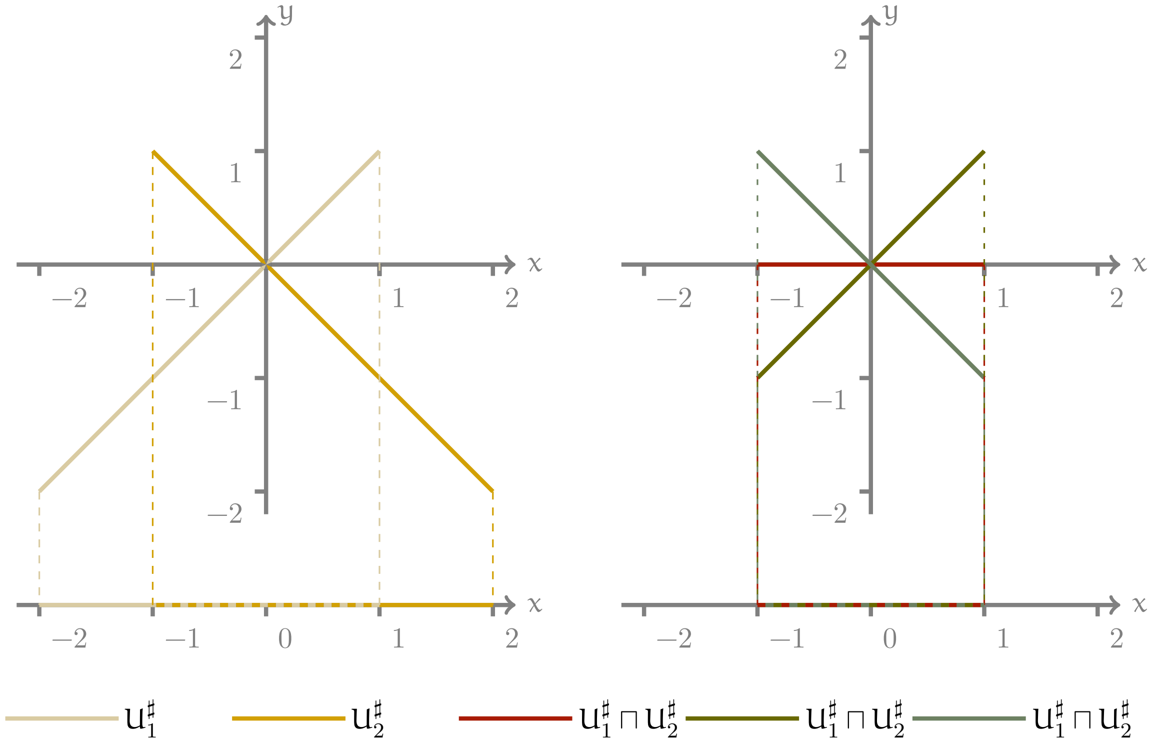


Figure 6.2 provides 3 polyhedra satisfying (1) and (2). These 3 polyhedra are incomparable and are minimal over-approximations of the concrete intersection.

Definition 6.13 (Meet operator $\sqcap$). Let us now define an abstract operator over-approximating the intersection operation:

$$\begin{aligned}\langle N_1^\sharp, l_1, u_1\rangle \sqcap \langle N_2^\sharp, l_2, u_2\rangle \triangleq & \textbf{if } \ l_1 \cup l_2 \not\subseteq u_1 \cap u_2 \textbf{ then}\\ & \quad \bot\\ & \textbf{else}\\ & \quad \langle {N_1^\sharp}_{|u_1 \cap u_2} \sqcap^{u_1 \cap u_2} ({N_2^\sharp}_{|l_1 \cup l_2})_{|u_1 \cap u_2}, l_1 \cup l_2, u_1 \cap u_2\rangle\end{aligned}$$

Example 6.8. Let us consider again the example from Figure 6.2. We have $U_1^\sharp = \langle \{y = x, x \geqslant -2, x \leqslant 1\}, \{x\}, \{x, y\}\rangle$ and $U_2^\sharp = \langle \{y = -x, x \geqslant -1, x \leqslant 2\}, \{x\}, \{x, y\}\rangle$. We have $\{x\} \cup \{x\} \subseteq \{x, y\} \cap \{x, y\}$, hence $U_1^\sharp \sqcap U_2^\sharp = \langle N^\sharp, \{x\}, \{x, y\}\rangle$, where $N^\sharp = \{y = x, x \geqslant -2, x \leqslant 1\} \sqcap^{\{x,y\}} (\{y = -x, x \geqslant -1, x \leqslant 2\}_{|\{x\}})_{|\{x,y\}} = \{y = x, x \geqslant -2, x \leqslant 1\} \sqcap^{\{x,y\}} \{x \geqslant -1, x \leqslant 2\} = \{y = x, x \geqslant -1, x \leqslant 1\}$. Finally Definition 6.13 is polyhedron ● from Figure 6.2. This example emphasizes that the idea behind the definition is the following: We can always keep constraints from both abstract elements that only relate variables that are present in both abstract elements. However $({N_1^\sharp}_{|l_1 \cup l_2} \sqcap^{l_1 \cup l_2} {N_2^\sharp}_{|l_1 \cup l_2})_{|u_1 \cap u_2}$ is very imprecise: in our example this would yield the polyhedron $\{x \geqslant -1, x \leqslant 1\}$ where y is unbounded. We have seen that the conjunction of constraints from $N_1^\sharp$ and $N_2^\sharp$ was unsound, however we can keep the constraints from one of the two abstract elements.

Remark 6.5 (Lack of Symmetry). Definition 6.13 is asymmetric: all constraints from $N_1^\sharp$ are kept, whereas only constraints on $l_1 \cup l_2$ are kept from $N_2^\sharp$. The symmetric definition can also be chosen. We do not provide here heuristics to guide this choice.

Proposition 6.7 (Soundness of the meet operator)**.** *The meet operator $\sqcap$ over-approximates the intersection.*

See proof on page 149.

Remark 6.6 (Homogeneous semantics). As for the abstract union, when the meet operator is used in the context of the representation of homogeneous sets of environments, it falls back to the computation of the classical meet operator. This ensures that we retain the precision and efficiency of this operator.

Finally as the abstract lattice is of infinite height we need to define a widening operator. As for the other operators our goal is to provide an operator which can be computed by one application of the underlying widening in the case of homogeneous environments. As no hypothesis is made on the set of variables, no structure can be used to define a clever widening for sequences of abstract elements. Therefore as soon as the environment diverges we will over-approximate by going to $\top$. This yields a huge loss of precision, however we feel that it is the role of the higher level abstraction storing variable in the numerical domain to ensure stabilization of the definition set before widening the numerical environment.

Definition 6.14 (Widening operator)**.** In order to ensure the stabilization of infinitely increasing chains in $\mathfrak{M}^\sharp$ we define the following widening operator:

$$\langle N_1^\sharp, l_1, u_1\rangle \triangledown \langle N_2^\sharp, l_2, u_2\rangle = \begin{cases} \langle N_1^\sharp \triangledown^{u_1} N^\sharp_{2\,|u}, l_1, u_1\rangle & \text{when } l_1 \subseteq l_2 \wedge u_2 \subseteq u_1 \\ \langle N_1^\sharp \boxed{\star} N_2^\sharp, l_2, u_1\rangle & \text{when } l_2 \subset l_1 \wedge u_2 \subseteq u_1 \\ \top & \text{otherwise} \end{cases}$$

Remark 6.7. This definition contains 3 cases:

- The environment is stable (in the sense that the definition set of environments represented by the second argument of the widening is contained in the set of environments represented by the first argument), in which case we fall back to the widening operator of the underlying domain.
- The environment is not stable but the growth that occurred can only occur finitely many times. This is the case where the lower bound of the definition set of represented environments is decreased. As it is a finite set, it must stabilize. In this case we soundly fall back to the $\boxed{\star}$ operator.
- Finally the environment is not stable and the growth that occurred may be indefinitely repeated. In such cases we soundly over-approximate to $\top$. The tree abstraction from the following chapter does not make use of this case as it ensures stabilization of the environment before resorting to widening operations.

Proposition 6.8 (Soundness and termination of the widening operator)**.** *The $\triangledown$ operator is sound:* $\forall U_1^\sharp, U_2^\sharp, \gamma(U_1^\sharp) \subseteq \gamma(U_1^\sharp \triangledown U_2^\sharp) \wedge \gamma(U_2^\sharp) \subseteq \gamma(U_1^\sharp \triangledown U_2^\sharp)$. *Moreover it ensures stabilization of infinite sequences:* $\forall (U_n^\sharp)_{n\in\mathbb{N}}, (V_n^\sharp)_{n\in\mathbb{N}}, V_0^\sharp = U_0^\sharp \wedge \forall n \geqslant 0, V_{n+1}^\sharp = V_n^\sharp \triangledown U_{n+1}^\sharp \Rightarrow \exists N \in \mathbb{N}, \forall k \geqslant N, V_k^\sharp = V_N^\sharp$.

See proof on page 149.

6.4.4 Abstract transformers

We recall that our concrete semantics is defined on the four statements:

- $\texttt{Assume}(c)$, where $c \in$ *expr*
- $v \leftarrow e$, where $e \in$ *expr* and $v \in \mathcal{V}$
- $\texttt{remove}(v)$, where $v \in \mathcal{V}$
- $\texttt{add}(v)$, where $v \in \mathcal{V}$

In this subsection we provide abstract transformers computing an over-approximation of the concrete semantics of these statements.

Definition 6.15 (Abstract transformers).

$$\mathbb{S}^\sharp[\![\texttt{Assume}(c)]\!](\langle N^\sharp, l, u\rangle) = \begin{cases} \langle \texttt{Assume}(c)^u(N^\sharp), l \cup \mathbf{fv}(c), u\rangle & \text{if } l \cup \mathbf{fv}(c) \subseteq u \\ \bot & \text{otherwise} \end{cases}$$

$$\mathbb{S}^\sharp[\![x \leftarrow e]\!](\langle N^\sharp, l, u\rangle) = \begin{cases} \langle \mathbb{S}^\sharp[\![x \leftarrow e]\!]^u(N^\sharp), l \cup \mathbf{fv}(e) \cup \{x\}, u\rangle & \text{if } l \cup \mathbf{fv}(e) \cup \{x\} \subseteq u \\ \bot & \text{otherwise} \end{cases}$$

$$\mathbb{S}^\sharp[\![\texttt{add}(v)]\!](\langle N^\sharp, l, u\rangle) = \begin{cases} \langle N^\sharp, l, u\rangle & \text{if } v \in l \\ \langle (N^\sharp_{|u \setminus \{v\}})_{|u \cup \{v\}}, l \cup \{v\}, u \cup \{v\}\rangle & \text{otherwise} \end{cases}$$

$$\mathbb{S}^\sharp[\![\texttt{remove}(v)]\!](\langle N^\sharp, l, u\rangle) = \langle N^\sharp_{|u \setminus \{v\}}, l \setminus \{v\}, u \setminus \{v\}\rangle$$

Remark 6.8.
- Recall that the concrete semantics of $\texttt{Assume}(c)$ (resp. $x \leftarrow e$) filters out environments that are undefined on c (resp. x and e), this will ensure the soundness of the operators from Definition 6.15.
- Please note that when trying to add a new variable v, if all environments represented by the abstract element are already defined on v, (case $v \in l$) the abstract element will not be modified. However in the case where some environments are defined on v and some are not (case $v \notin l$ and $v \in u$) all existing constraints on v will be lost.
- Note that, once more, when used with sets of homogeneous environments, the definition of these abstract transformers falls back to the application of the corresponding operator on the underlying domain.

Proposition 6.9 (Soundness of the abstract transformers). *All the abstract transformers from Definition 6.15 soundly abstract their concrete counterpart:* $\forall \texttt{stmt}, \mathbb{S}[\![\texttt{stmt}]\!](\gamma(U^\sharp)) \subseteq \gamma(\mathbb{S}^\sharp[\![\texttt{stmt}]\!](U^\sharp))$.

See proof on page 150.

6.5 The integer case

In the definition of the Clip abstraction, we made the hypothesis that the lifted domain provided an exact abstract projection operator. This was needed to ensure soundness of the abstract operator resulting from the lifting. However the polyhedral projection is not exact when given an integer semantics. Therefore this exactness requirement restriction is too coarse, indeed abstractions of sets of integer maps have often been used as a essential part of an analysis. This comes from the fact that: (1) not only variables with integer types are often used in the analyzed programs, (2) but also because maps with integer values can be used to design higher level abstractions (e.g. predicate abstractions). In these two use cases, expressing relations between variables is crucial for precision, which often leads to the use of integer polyhedra.

Notations. In this section we will use polyhedra over $\mathbb{Q}$ and polyhedra over $\mathbb{Z}$ defined over some variable set $u \subseteq \mathcal{V}$. These two abstract domains have the same set of abstract elements (polyhedra, this set is denoted as $\mathcal{N}_P^u$), but their concretization functions differ. The abstract operators, abstract transformers and concretization function for these abstract domains will be denoted $\square^u_{P_\mathbb{Q}}$ (resp. $\square^u_{P_\mathbb{Z}}$), the abstract universal/existential quantifications will be denoted as $\square_{|\mathbb{Q}\square}$ (resp. $\square_{|\mathbb{Z}\square}$). We recall that, given a set of variables $\mathcal{W}$, and a polyhedron P over $\mathcal{W}$, the integer concretization of P is defined as $\gamma^{\mathcal{W}}_{P_\mathbb{Z}}(P) = \gamma^{\mathcal{W}}_{P_\mathbb{Q}}(P) \cap \mathbb{Z}^{\mathcal{W}}$

The classical abstraction using integer polyhedra does not enjoy an exact projection operator.

Example 6.9. Consider the polyhedra $P = \{x = 2y\}$, defined over $\{x, y\}$. Its concretization is $\gamma_{P_{\mathbb{Z}}}^{\mathcal{W}}(P) = \{(x \mapsto 2\alpha, y \mapsto \alpha) \mid \alpha \in \mathbb{Q}\} \cap \mathbb{Z}^{\{x,y\}} = \{(x \mapsto 2\alpha, y \mapsto \alpha) \mid \alpha \in \mathbb{Z}\}$. The concrete projection of $\gamma_{P_{\mathbb{Z}}}^{\mathcal{W}}(P)$ over $\{x\}$ is $\gamma_{P_{\mathbb{Z}}}^{\mathcal{W}}(P)_{|\{x\}} = \{(x \mapsto 2\alpha) \mid \alpha \in \mathbb{Z}\} \sim 2\mathbb{Z}$. Note however that $2\mathbb{Z}$ can not be represented as the integer concretization of some polyhedron. The abstract projection therefore only yields the over-approximation $P_{|\mathbb{Z}_{\{x\}}} = \top_{P_{\mathbb{Z}}}^{\{x\}}$ which is concretized into: $\{(x \mapsto \alpha) \mid \alpha \in \mathbb{Z}\} \sim \mathbb{Z}$.

6.5.1 Losing soundness

We show how this imprecision induces unsound abstract operator when trying to lift integer polyhedra with the Clip abstraction. Let us assume that we lifted the above integer polyhedra abstraction and consider the two following Clip abstract elements $U_1^\sharp = \langle \{x = 2y\}, \{x, y\}, \{x, y\} \rangle$ and $U_2^\sharp = \langle \top_{P_{\mathbb{Z}}}^{\{x\}}, \{x\}, \{x\} \rangle$. These elements are concretized into $\gamma(\mathfrak{S}_1^\sharp) = \{(x \mapsto 2\alpha, y \mapsto \alpha) \mid \alpha \in \mathbb{Z}\}$ and $\gamma(\mathfrak{S}_2^\sharp) = \{(x \mapsto \alpha) \mid \alpha \in \mathbb{Z}\}$. Let us join these two abstract elements. This requires the computation of $\{x = 2y\} \boxast \top_{P_{\mathbb{Z}}}^{\{x\}}$:

- $\{x = 2y\} \sqcup_{P_{\mathbb{Z}}}^{\{x,y\}} \top_{P_{\mathbb{Z}}}^{\{x\}}{}_{|\mathbb{Z}_{\{x,y\}}} = \top_{P_{\mathbb{Z}}}^{\{x,y\}}$
- we then try to assume back the constraint $x = 2y$, yielding the polyhedron $\{x = 2y\}$ and check whether we violate the soundness conditions:
  - $\{x = 2y\} \sqsubseteq_{P_{\mathbb{Z}}}^{\{x,y\}} \{x = 2y\}$: this test holds
  - $\top_{P_{\mathbb{Z}}}^{\{x\}} \sqsubseteq_{P_{\mathbb{Z}}}^{\{x\}} \{x = 2y\}_{|\mathbb{Z}_{\{x\}}} = \top_{P_{\mathbb{Z}}}^{\{x\}}$ as emphasized in Example 6.9, hence it holds as well.

As constraint $x = 2y$ is assumed back, joining the abstract element $U_1^\sharp$ and $U_2^\sharp$ yields $U_3^\sharp \triangleq \langle \{x = 2y\}, \{x\}, \{x, y\} \rangle$. This is concretized as $\{\rho \in \gamma_{P_{\mathbb{Z}}}^{\{x,y\}}(\{x = 2y\})_{|\mathcal{W}} \mid \{x\} \subseteq \mathcal{W} \subseteq \{x, y\}\} = \gamma_{P_{\mathbb{Z}}}^{\{x,y\}}(\{x = 2y\})_{|\{x\}} \cup \gamma_{P_{\mathbb{Z}}}^{\{x,y\}}(\{x = 2y\})_{|\{x,y\}} = \{(x \mapsto 2\alpha) \mid \alpha \in \mathbb{Z}\} \cup \{(x \mapsto 2\alpha, y \mapsto \alpha) \mid \alpha \in \mathbb{Z}\}$. We see here that $\gamma(U_1^\sharp) \cup \gamma(U_2^\sharp) \not\subseteq \gamma(U_1^\sharp \sqcup U_2^\sharp)$, indeed $(x \mapsto 1) \in \gamma(U_2^\sharp)$ but $(x \mapsto 1) \notin \gamma(U_1^\sharp \sqcup U_2^\sharp)$.

This soundness loss can also be emphasized on the inclusion test. Indeed let us compare $U_2^\sharp$ and $U_3^\sharp$: we have $\{x\} \subseteq \{x\}$ and $\{x\} \subseteq \{x, y\}$, moreover $\top_{P_{\mathbb{Z}}}^{\{x\}} \sqsubseteq_{P_{\mathbb{Z}}}^{\{x\}} \{x = 2y\}_{|\mathbb{Z}_{\{x\}}} = \top_{P_{\mathbb{Z}}}^{\{x\}}$. Therefore the relation $U_2^\sharp \sqsubseteq U_3^\sharp$ holds, even though $\{x \mapsto 1\} \in \gamma(U_2^\sharp)$ but $(x \mapsto 1) \notin \gamma(U_3^\sharp)$.

The reason for this soundness loss comes from the fact that the inclusion tests performed on the underlying domain (both for the computation of the join operation and for the computation of the inclusion tests) are preceded by a projection. As the projection operator strictly over-approximates, the comparison might be performed with an abstract element representing an over-approximation of the original set of concrete elements, this yields an unsound comparison.

6.5.2 Losing parametricity

As losing soundness is not acceptable, we will instead provide a new concretization function for polyhedra over $\mathbb{Z}$, that is not a lift of the usual concretization function. Indeed by opposition to the Clip abstraction which provides a parametric way to lift numerical domains, the technique provided in this section can only be applied on the polyhedra domain with integer semantics.

Definition 6.16 (Abstract element and concretization $\gamma_{\mathbb{Z}}$). As for the Clip abstraction, abstract element are of the form $\langle N^\sharp, l, u \rangle$ where $N^\sharp$ is a polyhedron defined on u, l and u are finite sets of variables such that $l \subseteq u$. The set of such abstract elements is denoted $\mathfrak{M}_{\mathbb{Z}}^\sharp$. These abstract elements are concretized with the following function: $\gamma_{\mathbb{Z}}(\langle N^\sharp, l, u \rangle) = \{\rho \in \mathcal{W} \to \mathbb{Z} \mid l \subseteq \mathcal{W} \subseteq u \land \rho \in \gamma_{P_{\mathbb{Q}}}^{u}(N^\sharp)_{|\mathcal{W}} \cap \mathbb{Z}^{\mathcal{W}}\}$. We will refer to this abstraction as the Clip$_{\mathbb{Z}}$ abstraction.

Remark 6.9. Please note that parametricity is lost since the projection is placed between the rational concretization and the intersection with $\mathbb{Z}$, the new concretization function is not built on top of $\gamma_{P_{\mathbb{Z}}}$.

Example 6.10. Let us consider our example $U_1^\sharp = \langle \{x = 2y\}, \{x\}, \{x, y\} \rangle$. We have

$$\begin{aligned}\gamma_{\mathbb{Z}}(U_1^\sharp) &= \gamma_{P_{\mathbb{Q}}}^{\{x,y\}}(\{x = 2y\})_{|\{x\}} \cap \mathbb{Z}^{\{x\}} \cup \gamma_{P_{\mathbb{Q}}}^{\{x,y\}}(\{x = 2y\})_{|\{x,y\}} \cap \mathbb{Z}^{\{x,y\}} \\ &= \{(x \mapsto 2\alpha, y \mapsto \alpha) \mid \alpha \in \mathbb{Q}\}_{|\{x\}} \cap \mathbb{Z}^{\{x\}} \cup \{(x \mapsto 2\alpha, y \mapsto \alpha) \mid \alpha \in \mathbb{Q}\}_{|\{x,y\}} \cap \mathbb{Z}^{\{x,y\}} \\ &= \{(x \mapsto 2\alpha) \mid \alpha \in \mathbb{Q}\} \cap \mathbb{Z}^{\{x\}} \cup \{(x \mapsto 2\alpha, y \mapsto \alpha) \mid \alpha \in \mathbb{Q}\} \cap \mathbb{Z}^{\{x,y\}} \\ &= \{(x \mapsto \alpha) \mid \alpha \in \mathbb{Z}\} \cup \{(x \mapsto 2\alpha, y \mapsto \alpha) \mid \alpha \in \mathbb{Z}\}\end{aligned}$$

This detailed computation emphasizes that interchanging the projection and the intersection with $\mathbb{Z}$ ensures that the restriction of the concretization to every definition set is a convex in $\mathbb{Z}$, hence removing the problematic case of representing non convex shape as $2\mathbb{Z}$, which would be the case by applying the CLIP abstraction lifting to the polyhedra concretization function.

Having modified the concretization function, we can now follow the same definitions as for the CLIP abstraction parameterized by the polyhedra abstraction over $\mathbb{Q}$. However some implementations[2] enable the user to specify whether a variable is an integer or a real number, in the case where the variable is an integer some simplifications might be performed, after operators have been computed. As an example consider the polyhedron ● from Figure 6.3 where y is an optional integer and x a non optional real. The first row of the figure shows a sound simplification of the polyhedron, indeed the concretization of this polyhedron (shown in ●) before and after the simplifications was unchanged. However if we now consider the second row, it shows (in ●) the concretization defined in this section. We see that the simplification induces a reduction of the projection on x. The simplification, which amounted to the removal of spurious non integers values, is therefore unsound. Our choice of concretization function therefore prevents us from using simplifications performed on integer variables as soon as those are optional. Therefore all optional variables are stored as real numbers (loosing precision), however as soon as they are made non optional, they are made integers, and so we gain back the ability to perform simplifications. For that reason we see that once more we fall back to the precision and expressiveness of the underlying domain as soon as all variables are mandatory.

6.6 Bridging the gap with partitioning: the ℘-CLIP abstraction

In Section 6.3.1 we provided an in-depth presentation of the partitioning abstraction. It is the most precise abstraction among those presented at this point. Indeed it uses one numerical abstract element per represented support, whereas the abstraction we defined in Section 6.4 encompasses all supports. In this Section we propose to bridge the gap between these two extremes.

As an introductory example consider Program 6.8. Assuming that we are analyzing this program using the CLIP abstraction parameterized by integers, we get: $U^\sharp = \langle (x \mapsto [0;0], y \mapsto [0;0]), \emptyset, \{x, y\} \rangle$ at program point ● and $V^\sharp = \langle (y \mapsto [1;1]), \{x\}, \{x\} \rangle$ at program point ★. The concretization of $U^\sharp$ yields the set of environments: $U = \{(), (x \mapsto 0), (y \mapsto 0), (x \mapsto 0, y \mapsto 0)\}$, the concretization of $V^\sharp$ yields $V = \{(y \mapsto 1)\}$. $U^\sharp$ and $V^\sharp$ need to be joined to complete the analysis, this yields $W^\sharp = \langle (x \mapsto [0;0], y \mapsto [0;1]), \emptyset, \{x, y\} \rangle$. The concretization of $W^\sharp$ is $\{(), (x \mapsto 0), (y \mapsto 0), (x \mapsto 0, y \mapsto 0), (y \mapsto 1), (x \mapsto 0, y \mapsto 1)\}$. We see that the environment $(x \mapsto 0, y \mapsto 1)$ can not be found in $U \cup V$, and is therefore a join inaccuracy. This inaccuracy on environment $\{x, y\}$ was introduced when joining environments defined on $\{y\}$. The partitioning abstraction does not suffer from such inaccuracies as maps defined on distinct environments are represented by different numerical domains. Let us simulate this behavior on our abstraction. When asked to join $U^\sharp$ and $V^\sharp$, we observe that $U^\sharp$ represents environments defined on $\{\mathcal{W} \in$

[2] such as the one provided by the APRON library (see [JM09]), used in the MOPSA framework

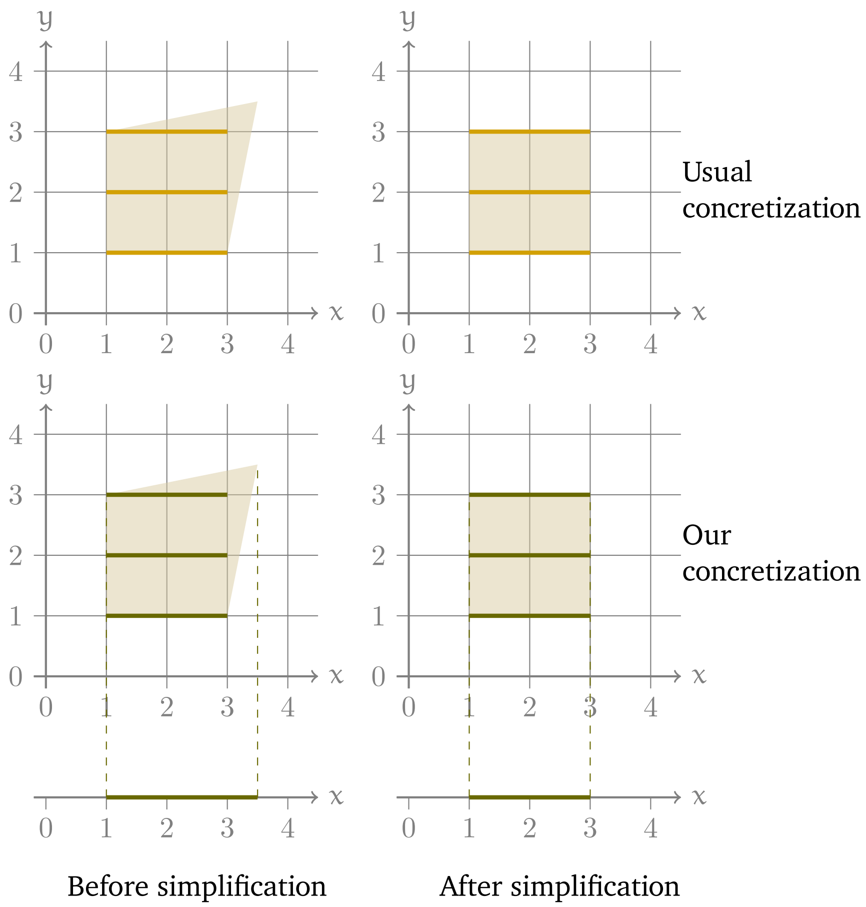


Figure 6.3: $\mathbb{Z}$ simplifications

```
if ?:
    if ?:
        x = 0
    else:
        y = 0
    ●
else:
    y = 1★
```

Program 6.8: Environment over-approximation example

$\wp_f(\mathcal{V}) \mid \emptyset \subseteq \mathcal{W} \subseteq \{x, y\}\} = \{\emptyset, \{x\}, \{y\}, \{x, y\}\}$ and $V^\sharp$ represents maps defined on $\{y\}$. Let us split $U^\sharp$ into a disjunction of abstract elements, each defined on disjoint sets of variables.

$$U^\sharp \rightarrow \left\{ \begin{array}{lll} U_1^\sharp = & \langle (x \mapsto 0, y \mapsto 0), & \{x, y\}, \quad \{x, y\}\rangle \\ U_2^\sharp = & \langle (x \mapsto 0), & \{\emptyset\}, \quad \{x\}\rangle \\ U_3^\sharp = & \langle (y \mapsto 0), & \{y\}, \quad \{y\}\rangle \end{array} \right\}$$

Note that $\gamma(U^\sharp) = \gamma(U_1^\sharp) \cup \gamma(U_2^\sharp) \cup \gamma(U_3^\sharp)$. This transformation is a step towards the partitioning domain. Note however that $U_2^\sharp$ still represents environments defined on $\{\}$ and environments defined on $\{x\}$, whereas in a partitioning abstraction each *monomial* (an element from the partition) is concretized to an homogeneous set of maps. Now that $U^\sharp$ is split we can perform a point-wise join as in the partitioning abstraction: $U_4^\sharp = U_3^\sharp \sqcup V^\sharp = \langle (y \mapsto [0;1]), \{y\}, \{y\}\rangle$. The result of the join operation is then expressed as the disjunction: $\{U_1^\sharp, U_2^\sharp, U_4^\sharp\}$. Please note that it is possible to apply operators point-wise as we ensured that the concretization is $U_1^\sharp$, $U_2^\sharp$ and $U_3^\sharp$ are pair-wise non intersecting, indeed they represent maps over distinct supports.

Power set lattice intervals. As we have seen, definitions from this section will rely heavily on the fact that monomials represent sets of maps defined on non intersecting supports. We have chosen to abstract sets of supports by their lower-bound/upper bound. We define an *Support Variable Interval (SVI)* of $\wp(\mathcal{V})$ to be a set I of support (so a set of sets of variables) such that: $\exists l \in \wp_f(\mathcal{V})$, $\exists u \in \wp_f(\mathcal{V})$, $I = \{\mathcal{W}, l \subseteq \mathcal{W} \subseteq u\}$. Whenever $l \in \wp_f(\mathcal{V})$ and $u \in \wp_f(\mathcal{V})$, $[l;u]$ denotes the SVI $\{\mathcal{W}, l \subseteq \mathcal{W} \subseteq u\}$, this might be empty. The intersection of two SVIs $[a;b]$ and $[c;d]$ is the SVI $[a \cup c; b \cap d]$, the union of two SVIs is not necessary a SVI. When two supports are in the same SVI I, their intersection and union are in I as well. Finally if two elements c and d are in the SVI $[a;b]$, the SVI $[c;d]$ is contained in the SVI $[a;b]$.

Running example. In this section, we will use the following two concrete sets as running examples.

$$P_1 \triangleq \{(), (x \mapsto 0), (y \mapsto \alpha), (x \mapsto \alpha, y \mapsto \alpha) \mid \alpha \leqslant 0\}$$
$$P_2 \triangleq \{(), (x \mapsto 1), (y \mapsto \alpha), (z \mapsto \beta, y \mapsto \alpha), (z \mapsto \beta) \mid \alpha \leqslant \beta \wedge \alpha \geqslant 0\}$$

6.6.1 Abstraction definition

An abstract element will be defined as a partitioning of several CLIP abstract elements, this abstraction will henceforth be referred to as the ℘-CLIP abstraction.

Definition 6.17 (℘-CLIP abstraction)**.** Let us define the set

$$\begin{aligned}\mathfrak{M}_\wp^\sharp = & \{\mathfrak{S}^\sharp \in \wp_f(\mathfrak{M}^\sharp) \mid \\ & \quad \forall U^\sharp \in \mathfrak{S}^\sharp,\ U^\sharp \neq \bot \\ & \quad \wedge U^\sharp = \top \Rightarrow \mathfrak{S}^\sharp = \{\top\} \\ & \quad \wedge \forall V^\sharp \neq U^\sharp \in \mathfrak{S}^\sharp, U^\sharp = \langle N^\sharp, l, u\rangle \wedge V^\sharp = \langle N^{\sharp\prime}, l', u'\rangle \Rightarrow [l;u] \cap [l';u'] = \emptyset \\ & \}\end{aligned}$$

As for partitioning, elements of $\mathfrak{M}_\wp^\sharp$ are sets of elements form $\mathfrak{M}^\sharp$, moreover in a partitioning abstraction we impose that at most one numerical abstract element is present per support, here we have the same restriction but for non intersecting sets of supports. Moreover the biggest element of our abstraction (for the order relation of Definition 6.22) will be $\{\top\}$ and its smallest element will be $\emptyset$.

Example 6.11 (Using the running example)**.** P_1, P_2 can be represented respectively by:

$$\mathfrak{S}_1^\sharp = \{\langle \{x = 0\}, \emptyset, \{x\}\rangle, \langle \{y \leqslant 0, x = y\}, \{y\}, \{x, y\}\rangle\}$$
$$\mathfrak{S}_2^\sharp = \{\langle \{x = 1\}, \{x\}, \{x\}\rangle, \langle \{y \geqslant 0, y \leqslant z\}, \emptyset, \{y, z\}\rangle\}$$

Definition 6.18 (Concretization function)**.** We recall that γ denotes the concretization function from $\mathfrak{M}^\sharp$ to $\mathfrak{M}$. Upon γ we define the concretization from $\mathfrak{M}^\sharp_\wp$ to $\mathfrak{M}$ by:

$$\gamma_\wp(\mathfrak{S}^\sharp) = \bigcup_{\mathfrak{U}^\sharp \in \mathfrak{S}^\sharp} \gamma(\mathfrak{U}^\sharp)$$

As was expected we have: $\gamma_\wp(\{\top\}) = \mathfrak{M}$ and $\gamma_\wp(\emptyset) = \emptyset$

Example 6.12 (Using the running example)**.** By definition, we have $\gamma_\wp(\mathfrak{S}^\sharp_1) = \gamma(\langle\{x = 0\}, \emptyset, \{x\}\rangle) \cup \gamma(\langle\{y \leqslant 0, x = y\}, \{y\}, \{x, y\}\rangle) = \{(), (x \mapsto 0)\} \cup \{(y \mapsto \beta), (x \mapsto \alpha, y \mapsto \beta) \mid \alpha = \beta \wedge \beta \leqslant 0\} = P_1$. Similarly we have $\gamma_\wp(\mathfrak{S}^\sharp_2) = P_2$.

Link with partitioning abstraction. If we impose in Definition 6.17 that $\forall \mathfrak{U}^\sharp \in \mathfrak{S}^\sharp, \mathfrak{U}^\sharp = \langle N^\sharp, l, u\rangle \Rightarrow l = u$, we fall back to the classical partitioning definition. Indeed we have seen in the previous section that whenever $\langle N^\sharp, l, u\rangle$ was such that $l = u$ the Clip abstraction behaves as its underlying numerical abstraction. On the contrary if we impose that $|\mathfrak{S}^\sharp| \leqslant 1$, we fall back to the Clip abstraction as our abstraction contains only one monomial and its concretization is that of the Clip abstraction. This shows that the $\wp$-Clip abstraction defined here can emulate the precise and costly partitioning abstraction as well as the, not so precise, Clip abstraction. Its behavior will be defined by the number of allowed monomials in the abstraction.

As two elements $\mathfrak{S}^\sharp_1$ and $\mathfrak{S}^\sharp_2$ from $\mathfrak{M}^\sharp_\wp$ might contain monomials defined on incomparable SVIs, the definition of operators requires abstract elements to be unified. This unification requires us to perform basic operations on SVIs. Note that results from these operations need to be expressed as a disjoint union of SVIs. We have seen that intersections can be easily computed, however SVI difference is more complicated and is the subject of the following subsection.

6.6.2 Set difference between SVIs

As mentioned before, the unification of two abstract elements requires the computation of the difference between two SVIs. When $I = [a; b]$ and $J = [c; d]$ are two SVIs, $I \setminus J = I \setminus (I \cap J)$. $I \cap J = [a \cup c; b \cap d]$ is a SVI (potentially empty) contained in I. For this reason we focus in this subsection on the computation of the difference between a SVI $[l; u]$ and a SVI $[l'; u']$ such that $l \subseteq l' \subseteq u' \subseteq u$. This is not always possible to express this difference as a SVI, however it can be expressed as a union of pair-wise disjoint SVIs (indeed singletons are SVIs). We recall that we need SVIs to be pair-wise disjoint so as to ease the definition of the set abstract operators and to avoid the potential precision losses due to supports represented in several monomials. As an example consider the difference between the SVI $I = [\emptyset; \{x_0, x_1, x_2\}]$ and the SVI $J = [\{x_0\}; \{x_0, x_1\}]$. A first naive solution is to enumerate of every singleton in I, and to check whether this singleton is in J, in which case it is discarded; or not, in which case it is added to the result. In our case, this yields: $\{[\emptyset; \emptyset], [x_1; x_1], [x_2; x_2], [\{x_0, x_2\}; \{x_0, x_2\}], [\{x_1, x_2\}; \{x_1, x_2\}], [\{x_0, x_1, x_2\}; \{x_0, x_1, x_2\}]$. The difference is therefore expressed using 6 SVIs and is illustrated in figure 6.4. On the contrary, Figure 6.5 shows how the difference can be expressed using only two SVIs: $[\emptyset; \{x_1, x_2\}]$ and $[\{x_0, x_2\}; \{x_0, x_1, x_2\}]$. We recall that the number of numerical abstract elements (e.g. polyhedra) stored in our abstraction (and therefore the number of time atomic operations on polyhedra will be performed per abstract operation) is equal to the number of SVIs. Our goal is therefore to compute a SVI difference using as few SVIs as possible.

Proposition 6.10. *If* $I = [l; u]$ *and* $J = [l'; u']$ *with* $l \subseteq l' \subseteq u' \subseteq u$ *are such that* $I \setminus J = \biguplus_{k \in \mathcal{K}} I_k$ *and every* I_k *is a SVI, then* $|\mathcal{K}| \geqslant |u \setminus u'| + |l' \setminus l|$.

See proof on page 152.

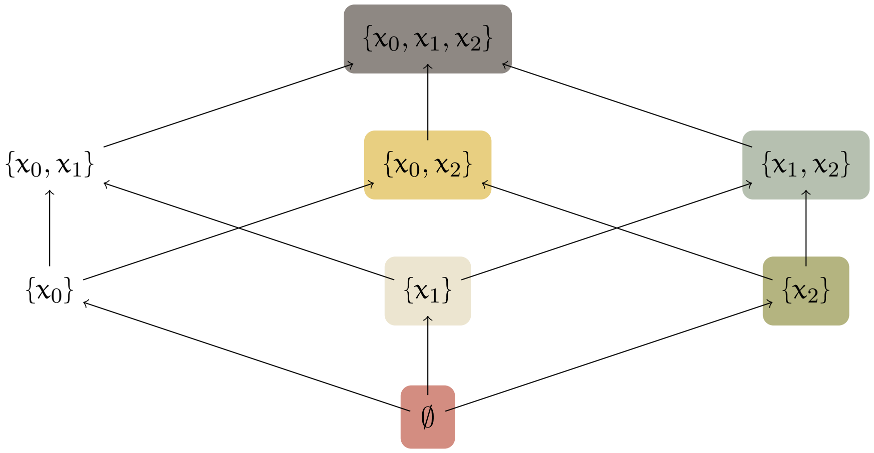


Figure 6.4: Non optimal SVI difference for $[\emptyset; \{x_0, x_1, x_2\}] \setminus [\{x_0\}; \{x_0, x_1\}]$

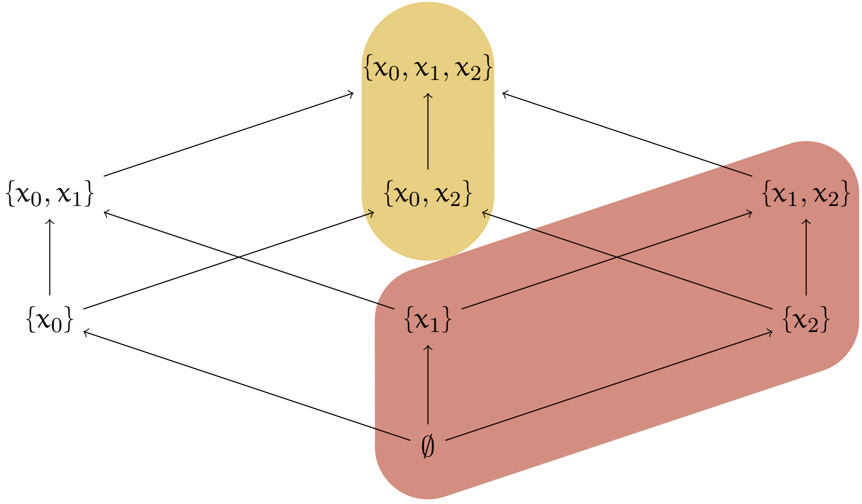


Figure 6.5: Optimal SVI difference for $[\emptyset; \{x_0, x_1, x_2\}] \setminus [\{x_0\}; \{x_0, x_1\}]$

Proposition 6.11. *If* $I = [l; u]$ *and* $J = [l'; u']$ *with* $l \subseteq l' \subseteq u' \subseteq u$, *such that* $u \setminus u' = \{y_1, \ldots, y_p\}$ *and* $l' \setminus l = \{x_1, \ldots, x_n\}$,

$$\textit{let} \quad (U_i)_{1 \leqslant i \leqslant n} = [l \cup \bigcup_{1 \leqslant k \leqslant i-1} \{x_k\}; u \setminus \{x_i\}]$$
$$\textit{and} \quad (L_j)_{1 \leqslant j \leqslant p} = [l' \cup \{y_j\}; u \setminus \bigcup_{1 \leqslant k \leqslant j-1} \{y_k\}]$$

we have $I \setminus J = \biguplus_{1 \leqslant i \leqslant n} U_i \uplus \biguplus_{1 \leqslant j \leqslant p} L_j$.

See proof on page 153.

Proposition 6.11 gives us a disjoint union of SVIs of $I \setminus J$. Moreover Proposition 6.10 ensures that the union found in Proposition 6.11 is optimal. The partitioning from Figure 6.6 was obtained using sets U_i and L_j from Program 6.11. Please not that this solution is not unique and that no choice heuristics were studied in this thesis.

Definition 6.19 (itv_diff). Following the aforementioned results, whenever $l \subseteq l' \subseteq u' \subseteq u$ we

Algorithm 6.2: itv_incl

```
Input  : $I$ an interval, $Is = \{I_1, \dots, I_n\}$ a set of intervals
Output: whether $I \subseteq \bigcup_{J \in Is} J$
rem ← $\{I\}$;
foreach $i \in \{1, n\}$ do
    rem ← $\bigcup_{L \in \mathtt{rem}}$ itv_diff$(L, I_i)$;
return $\bigcup_{L \in \mathtt{rem}} L = \emptyset$;
```

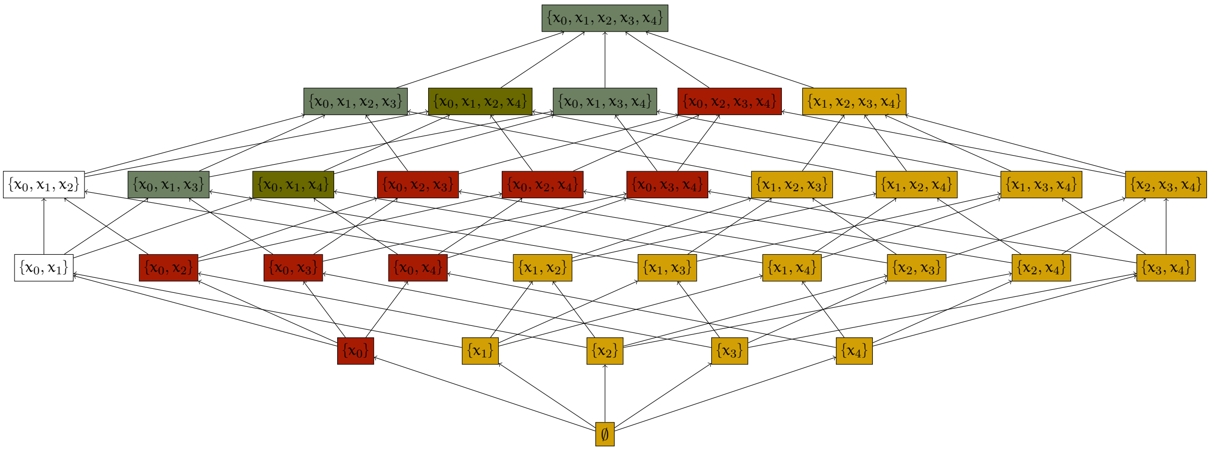


Figure 6.6: Disjoint union from Proposition 6.11 for $[\emptyset; \{x_0, x_1, x_2, x_3, x_4\}] \setminus [\{x_0, x_1\}; \{x_0, x_1, x_2\}]$

define:

$$\mathbf{itv\_diff}([l; u], [l'; u']) = \bigcup_{1 \leqslant i \leqslant n} \{U_i\} \cup \bigcup_{1 \leqslant j \leqslant p} \{L_j\}$$

where U_i and J_j are as defined in Proposition 6.11. This definition is generalized to any two SVIs by computing the difference with the intersection of the two SVIs.

Definition 6.20 (**itv_incl**)**.** Finally we define **itv_incl** operator which computes whether a SVI is contained in the union of several other SVIs. This algorithm removes one by one the SVIs in its second argument from its first argument (using Definition 6.19) and checks that the result is empty. The definition is provided in Algorithm 6.2.

See proof on page 153.

6.6.3 Operator definitions

We now provide operators on $\mathfrak{M}^\sharp_\wp$. We will define two unifications **unify_leq** and **unify**, **unify_leq** will be used to define the order relation on $\mathfrak{M}^\sharp_\wp$ and **unify** for the definition of the other operators. Indeed the general unification **unify** is such that: if $(\mathfrak{S}^{\sharp\,\prime}_1, \mathfrak{S}^{\sharp\,\prime}_2) = \mathbf{unify}(\mathfrak{S}^\sharp_1, \mathfrak{S}^\sharp_2)$, then $\gamma_\wp(\mathfrak{S}^\sharp_1) \subseteq \gamma_\wp(\mathfrak{S}^{\sharp\,\prime}_1)$ and $\gamma_\wp(\mathfrak{S}^\sharp_2) \subseteq \gamma_\wp(\mathfrak{S}^{\sharp\,\prime}_2)$. While this is enough for the abstract definition of monotone operators (as join, meet, widening) to be sound, it is not for the abstract inclusion test. Indeed $\gamma_\wp(\mathfrak{S}^{\sharp\,\prime}_1) \subseteq \gamma_\wp(\mathfrak{S}^{\sharp\,\prime}_2) \not\Rightarrow \gamma_\wp(\mathfrak{S}^\sharp_1) \subseteq \gamma_\wp(\mathfrak{S}^\sharp_2)$. For this reason we define an ad-hoc **unify_leq** operator that does not over-approximates its right argument.

For the following definitions we use the operator $\mathbf{itvs}(\mathfrak{S}^\sharp \in \mathfrak{M}^\sharp_\wp)$ which yields the set of SVIs on which monomials of $\mathfrak{S}^\sharp$ are defined: $\mathbf{itvs}(\{\langle\_, l_1, u_1\rangle, \dots, \langle\_, l_n, u_n\rangle\}) = \{[l_1; u_1], \dots, [l_n; u_n]\}$

Definition 6.21 (**unify_leq**)**.** Algorithm 6.3 provides the definition of a **unify_leq** operator

Algorithm 6.3: unify_leq

```
Input  : $\mathfrak{S}^\sharp_1, \mathfrak{S}^\sharp_2$ two abstract elements
Output: $\mathfrak{S}^{\sharp\,\prime}_1, \mathfrak{S}^{\sharp\,\prime}_2$ their unification
res ← $\emptyset$;
foreach $\langle N^\sharp, l, u\rangle \in \mathfrak{S}^\sharp_1$ do
    foreach $\langle \_, l', u'\rangle \in \mathfrak{S}^\sharp_2$ do
        if $[a; b] \triangleq [l; u] \cap [l'; u'] \neq \emptyset$ then
            res ← res $\cup \{\langle N^\sharp_{|b}, a, b\rangle\}$;
        end
    done
done
return (res, $\mathfrak{S}^\sharp_2$);
```

Example 6.13. For this example we consider the following, slightly modified, version of the running examples: $\mathfrak{S}^\sharp_1 = \{\langle\{x = 0\}, \emptyset, \{x\}\rangle, \langle\{y \leqslant 0, x = y\}, \{y\}, \{x, y\}\rangle\}$ and $\mathfrak{S}^\sharp_2 = \{(M_1 \triangleq)\langle\{x = 1\}, \{x\}, \{x\}\rangle, (M_2 \triangleq)\langle\{y \geqslant 0, y \leqslant z\}, \emptyset, \{y, z\}\rangle, (M_3 \triangleq)\langle\{y \geqslant x\}, \{x, y\}, \{x, y\}\rangle\}$. The only difference with our running examples is that the monomial $\langle\{y \geqslant x\}, \{x, y\}, \{x, y\}\rangle$ was added to $\mathfrak{S}^\sharp_2$. The **unify_leq** operator will leave $\mathfrak{S}^\sharp_2$ unchanged, however:

- The monomial $\langle\{x = 0\}, \emptyset, \{x\}\rangle$ (the SVI of which intersected the SVI from M_1 and the SVI from M_2) is split into the two monomials: $\langle\{\}, \emptyset, \emptyset\rangle$ (which SVI intersects only the one from M_2) and $\langle\{x = 0\}, \{x\}, \{x\}\rangle$ (which SVI intersects only the one from M_1).
- The monomial $\langle\{y \leqslant 0, x = y\}, \{y\}, \{x, y\}\rangle$ (which SVI intersected the SVI from M_2 and the SVI from M_3) is split into the two monomials: $\langle\{y \leqslant 0\}, \{y\}, \{y\}\rangle$ (which SVI intersects only the one from M_2) and $\langle\{x = y \leqslant 0\}, \{x, y\}, \{x, y\}\rangle$ (which SVI intersects only the one from M_3)

The result from the unification is therefore:

$$\{\langle\{\}, \emptyset, \emptyset\rangle, \langle\{x = 0\}, \{x\}, \{x\}\rangle, \langle\{y \leqslant 0\}, \{y\}, \{y\}\rangle, \langle\{x = y \leqslant 0\}, \{x, y\}, \{x, y\}\rangle\}$$

Please note that this unification tends to transform the ℘-CLIP abstraction into the partitioning abstraction.

Proposition 6.12 (Soundness of **unify_leq**)**.** *If* $(\mathfrak{S}^{\sharp\,\prime}_1, \mathfrak{S}^{\sharp\,\prime}_2) = \textit{\textbf{unify\_leq}}(\mathfrak{S}^\sharp_1, \mathfrak{S}^\sharp_2)$ *then:*

$$\begin{aligned}
&\Big(\bigcup_{I \in \textit{\textbf{itvs}}(\mathfrak{S}^\sharp_1)} I \subseteq \bigcup_{J \in \textit{\textbf{itvs}}(\mathfrak{S}^\sharp_2)} J\Big)\\
&\quad \Rightarrow \gamma_\wp(\mathfrak{S}^\sharp_1) \subseteq \gamma_\wp(\mathfrak{S}^{\sharp\,\prime}_1)\\
&\qquad \wedge \gamma_\wp(\mathfrak{S}^\sharp_2) = \gamma_\wp(\mathfrak{S}^{\sharp\,\prime}_2)\\
&\qquad \wedge \forall \langle \_, l, u\rangle \in \mathfrak{S}^{\sharp\,\prime}_1, \exists \langle \_, l', u'\rangle \in \mathfrak{S}^{\sharp\,\prime}_2, [l; u] \subseteq [l'; u']
\end{aligned}$$

See proof on page 153.

Remark 6.10. Whenever the projection operator of the underlying domain is exact, the **unify_leq** operator is exact in the sense that: $(\mathfrak{S}^{\sharp\,\prime}_1, \mathfrak{S}^{\sharp\,\prime}_2) = \textbf{unify\_leq}(\mathfrak{S}^\sharp_1, \mathfrak{S}^\sharp_2) \Rightarrow \gamma_\wp(\mathfrak{S}^\sharp_1) = \gamma_\wp(\mathfrak{S}^{\sharp\,\prime}_1) \wedge \gamma_\wp(\mathfrak{S}^\sharp_2) = \gamma_\wp(\mathfrak{S}^{\sharp\,\prime}_2)$

Using **unify_leq**, we can now define an order relation on $\mathfrak{M}^\sharp_\wp$.

Definition 6.22 (Order relation on $\mathfrak{M}^\sharp_\wp$: $\sqsubseteq_\wp$)**.** We define:

$$
\begin{aligned}
\sqsubseteq_{\wp} \stackrel{\Delta}{=} \{ & (\mathfrak{S}^\sharp_1, \mathfrak{S}^\sharp_2) \mid \forall I \in \mathbf{itvs}(\mathfrak{S}^\sharp_1), \mathbf{itv\_incl}(I, \mathbf{itvs}(\mathfrak{S}^\sharp_2)) \\
& \wedge ({\mathfrak{S}^\sharp_1}', {\mathfrak{S}^\sharp_2}') = \mathbf{unify\_leq}(\mathfrak{S}^\sharp_1, \mathfrak{S}^\sharp_2) \Rightarrow \forall U^\sharp_1 \in {\mathfrak{S}^\sharp_1}', \exists U^\sharp_2 \in {\mathfrak{S}^\sharp_2}', U^\sharp_1 \sqsubseteq U^\sharp_2 \\
\} &
\end{aligned}
$$

Proposition 6.13 (Soundness of $\sqsubseteq_{\wp}$). $\gamma_{\wp} \in (\mathfrak{M}^\sharp_{\wp}, \sqsubseteq_{\wp}) \to (\mathfrak{M}, \subseteq)$ *is monotonic.*

See proof on page 154.

Example 6.14. As for the previous example, we use the following slightly modified version of our running example: $\mathfrak{S}^\sharp_1 = \{\langle \{x = 0\}, \emptyset, \{x\}\rangle, \langle \{y \leqslant 0, x = y\}, \{y\}, \{x, y\}\rangle\}$ and $\mathfrak{S}^\sharp_2 = \{(M'_1 \stackrel{\Delta}{=})\langle \{x = 1\}, \{x\}, \{x\}\rangle, (M'_2 \stackrel{\Delta}{=})\langle \{y \geqslant 0, y \leqslant z\}, \emptyset, \{y, z\}\rangle, (M'_3 \stackrel{\Delta}{=})\langle \{y \geqslant x\}, \{x, y\}, \{x, y\}\rangle\}$. We have $\mathbf{itvs}(\mathfrak{S}^\sharp_1) = \{[\emptyset; \{x\}], [\{y\}, \{x, y\}]\}$, and $\mathbf{itvs}(\mathfrak{S}^\sharp_2) = \{[\{x\}; \{x\}], [\emptyset; \{y, z\}], [\{x, y\}; \{x, y\}]\}$. Hence the constraint that $\forall I \in \mathbf{itvs}(\mathfrak{S}^\sharp_1), \mathbf{itv\_incl}(I, \mathbf{itvs}(\mathfrak{S}^\sharp_2))$ is satisfied. Moreover we have shown in Example 6.13 that ${\mathfrak{S}^\sharp_1}' = \{(M_1 \stackrel{\Delta}{=})\langle \{\}, \emptyset, \emptyset\rangle, (M_2 \stackrel{\Delta}{=})\langle \{x = 0\}, \{x\}, \{x\}\rangle, (M_3 \stackrel{\Delta}{=})\langle \{y \leqslant 0\}, \{y\}, \{y\}\rangle, (M_4 \stackrel{\Delta}{=})\langle \{x = y \leqslant 0\}, \{x, y\}, \{x, y\}\rangle\}$ and ${\mathfrak{S}^\sharp_2}' = \mathfrak{S}^\sharp_2$. As we do not have $M_2 \sqsubseteq M'_i$ for $i \in \{1, 2, 3\}$, we do not have $\mathfrak{S}^\sharp_1 \sqsubseteq_{\wp} \mathfrak{S}^\sharp_2$. Note that this was expected by soundness of the operator, indeed $(x \mapsto 0) \in \gamma_{\wp}(\mathfrak{S}^\sharp_1)$ but $(x \mapsto 0) \notin \gamma_{\wp}(\mathfrak{S}^\sharp_2)$.

Let us now define abstract operators for the classical set operations: $\cup$ and $\cap$. We start by defining a **unify** operator which transforms two abstract elements in abstract elements defined on the same SVI sets.

Definition 6.23 (**unify**). Algorithm 6.4 provides a general purpose unification operator. Figure 6.7 illustrates the behavior of this operator.

Algorithm 6.4 is "symmetric" in $\mathfrak{S}^\sharp_1, \mathfrak{S}^\sharp_2$. The idea is the following: each monomial over some $[l; u]$ from $\mathfrak{S}^\sharp_1$ is partitioned into two sets: $\{[l_{1,c}; u_{1,c}], \dots, [l_{n,c}; u_{n,c}]\}$ and $\{[l_1; u_1] \dots [l_m; u_m]\}$ such that $[l; u] = [l_{1,c}; u_{1,c}] \cup \dots [l_{n,c}; u_{n,c}] \cup [l_1; u_1] \cup \dots [l_m; u_m]$. As an example in Figure 6.7, SVIs I is partitioned into $I_{1,c}$, $I_{2,c}$, I_1 and I_2. The $[l_{i,c}; u_{i,c}]$ are SVIs for which $\mathfrak{S}^\sharp_2$ also provides a monomial. The unification then ensures that if ${\mathfrak{S}^\sharp_1}', {\mathfrak{S}^\sharp_2}'$ both represent concrete environments over some shared variables set $\mathcal{W}$, then $\mathcal{W}$ is found in two monomials from ${\mathfrak{S}^\sharp_1}'$ and ${\mathfrak{S}^\sharp_2}'$ that are defined on the same SVI.

Proposition 6.14 (Soundness of **unify**). *If* $({\mathfrak{S}^\sharp_1}', {\mathfrak{S}^\sharp_2}') = \textbf{\textit{unify}}(\mathfrak{S}^\sharp_1, \mathfrak{S}^\sharp_2)$ *then:*

$$
\begin{aligned}
& \gamma_{\wp}(\mathfrak{S}^\sharp_1) \subseteq \gamma_{\wp}({\mathfrak{S}^\sharp_1}') \\
& \wedge \gamma_{\wp}(\mathfrak{S}^\sharp_2) \subseteq \gamma_{\wp}({\mathfrak{S}^\sharp_2}') \\
& \wedge \forall [l_1; u_1] \in \textbf{\textit{itvs}}({\mathfrak{S}^\sharp_1}'), \forall [l_2, u_2] \in \textbf{\textit{itvs}}({\mathfrak{S}^\sharp_2}'), [l_1; u_1] \cap [l_2; u_2] \neq \emptyset \Rightarrow l_1 = l_2 \wedge u_1 = u_2
\end{aligned}
$$

See proof on page 154.

Example 6.15 (Using the running example). Let us move back to our running example: $\mathfrak{S}^\sharp_1 = \{\langle \{x = 0\}, \emptyset, \{x\}\rangle, \langle \{y \leqslant 0, x = y\}, \{y\}, \{x, y\}\rangle\}$ and $\mathfrak{S}^\sharp_2 = \{\langle \{x = 1\}, \{x\}, \{x\}\rangle, \langle \{y \geqslant 0, y \leqslant z\}, \emptyset, \{y, z\}\rangle\}$. Let us show how this two abstract elements are unified. Figure 6.8 provides an illustration of the, pre and post unification, SVIs for $\mathfrak{S}^\sharp_1$ and $\mathfrak{S}^\sharp_2$. Let us denote ${\mathfrak{S}^\sharp_1}'$ and ${\mathfrak{S}^\sharp_2}'$ the unification of $\mathfrak{S}^\sharp_1$ and $\mathfrak{S}^\sharp_2$. Let us consider the choice of monomial $\langle \{y \geqslant 0, y \leqslant z\}, \emptyset, \{y, z\}\rangle$ at line 15. We initially have $\texttt{rem} = \{[\emptyset; \{y, z\}]\}$.

Algorithm 6.4: unify

```
Input  : $\mathfrak{S}^\sharp_1, \mathfrak{S}^\sharp_2$ two abstract elements
Output: $\mathfrak{S}^{\sharp\,\prime}_1, \mathfrak{S}^{\sharp\,\prime}_2$ their unification
1  $\mathtt{res}_1 \leftarrow \emptyset$;
2  $\mathtt{res}_2 \leftarrow \emptyset$;
3  $\mathtt{res}_{1,c} \leftarrow \emptyset$;
4  $\mathtt{res}_{2,c} \leftarrow \emptyset$;
5  foreach $\langle N^\sharp, l, u\rangle \in \mathfrak{S}^\sharp_1$ do
6      $\mathtt{rem} \leftarrow \{[l; u]\}$;
7      foreach $\langle N^{\sharp\prime}, l', u'\rangle \in \mathfrak{S}^\sharp_2$ do
8          if $[a; b] \triangleq [l; u] \cap [l'; u'] \neq \emptyset$ then
9              $\mathtt{res}_{1,c} \leftarrow \mathtt{res}_{1,c} \cup \{\langle N^\sharp_{|b}, a, b\rangle\}$;
10             $\mathtt{rem} \leftarrow \bigcup_{J \in \mathtt{rem}} \textbf{diff\_itv}(J, [a; b])$;
11         end
12         $\mathtt{res}_1 \leftarrow \mathtt{res}_1 \bigcup_{[a;b] \in \mathtt{rem}} \{\langle N^\sharp_{|b}, a, b\rangle\}$;
13     done
14 done
15 foreach $\langle N^{\sharp\prime}, l', u'\rangle \in \mathfrak{S}^\sharp_2$ do
16     $\mathtt{rem} \leftarrow \{[l'; u']\}$;
17     foreach $\langle N^\sharp, l, u\rangle \in \mathfrak{S}^\sharp_1$ do
18         if $[a; b] \triangleq [l; u] \cap [l'; u'] \neq \emptyset$ then
19             $\mathtt{res}_{2,c} \leftarrow \mathtt{res}_{2,c} \cup \{\langle N^{\sharp\prime}_{|b}, a, b\rangle\}$;
20             $\mathtt{rem} \leftarrow \bigcup_{J \in \mathtt{rem}} \textbf{diff\_itv}(J, [a; b])$;
21         end
22         $\mathtt{res}_2 \leftarrow \mathtt{res}_2 \bigcup_{[a;b] \in \mathtt{rem}} \{\langle N^{\sharp\prime}_{|b}, a, b\rangle\}$;
23     done
24 done
25 return $(\mathtt{res}_{1,c} \cup \mathtt{res}_1, \mathtt{res}_{2,c} \cup \mathtt{res}_2)$;
```

- When $\langle\{x = 0\}, \emptyset, \{x\}\rangle$ is chosen at line 17: we have $[a; b] \triangleq [\emptyset; \{y, z\}] \cap [\emptyset; \{x\}] = [\emptyset; \emptyset] \neq \emptyset$. We then have: $\mathtt{res}_{2,c} \leftarrow \emptyset \cup \{\langle\{y \geqslant 0, y \leqslant z\}_{|\emptyset}, \emptyset, \emptyset\rangle\} = \{\langle\{\}, \emptyset, \emptyset\rangle\}$. Moreover $\mathtt{rem} \leftarrow$ **diff_itv**$([\emptyset, \{y, z\}], [\emptyset; \emptyset]) = \{[\{y\}; \{y\}], [\{z\}, \{y, z\}]\}$
- When $\langle\{y \leqslant 0, x = y\}, \{y\}, \{x, y\}\rangle$ is chosen at line 17: we have $[a; b] \triangleq [\emptyset; \{y, z\}] \cap [\{y\}; \{x, y\}] = [\{y\}; \{y\}] \neq \emptyset$. We then have: $\mathtt{res}_{2,c} \leftarrow \mathtt{res}_{2,c} \cup \{\langle\{y \geqslant 0, y \leqslant z\}_{|\{y\}}, \{y\}, \{y\}\rangle\} = \{\langle\{\}, \emptyset, \emptyset\rangle, \langle\{y \geqslant 0\}, \{y\}, \{y\}\rangle\}$. Moreover $\mathtt{rem} \leftarrow$ **diff_itv**$([\{y\}; \{y\}], [\{y\}; \{y\}]) \cup$ **diff_itv**$([\{z\}; \{y, z\}], [\{y\}; \{y\}]) = \emptyset \cup \{[\{z\}; \{y, z\}]\} = \{[\{z\}; \{y, z\}]\}$.

Once the two monomials from $\mathfrak{S}^\sharp_1$ have been considered, we have $\mathtt{res}_2 = \{\langle\{y \geqslant 0, y \leqslant z\}_{|\{y,z\}}, \{z\}, \{y, z\}\rangle\} = \{\langle\{y \geqslant 0, y \leqslant z\}, \{z\}, \{y, z\}\rangle\}$ is added to $\mathtt{res}_2$. We skip the details for the other monomials, the result of the unification is then $\mathfrak{S}^{\sharp\,\prime}_1 = \{\langle\emptyset, \emptyset, \emptyset\rangle, \langle\{x = 0\}, \{x\}, \{x\}\rangle, \langle\{y \leqslant 0\}, \{y\}, \{y\}\rangle, \langle\{y \leqslant 0, x = y\}, \{x, y\}, \{x, y\}\rangle\}$ and $\mathfrak{S}^{\sharp\,\prime}_2 = \{\langle\emptyset, \emptyset, \emptyset\rangle, \langle\{y \geqslant 0\}, \{y\}, \{y\}\rangle, \langle\{y \geqslant 0, y \leqslant z\}, \{z\}, \{y, z\}\rangle, \langle\{x = 1\}, \{x\}, \{x\}\rangle\}$, illustrated in the following table:

SVIs	$\mathfrak{S}^{\sharp\,\prime}_1$	$\mathfrak{S}^{\sharp\,\prime}_2$
$[\emptyset; \emptyset]$	$\emptyset$	$\emptyset$
$[\{x\}, \{x\}]$	$\{x = 0\}$	$\{x = 1\}$
$[\{y\}, \{y\}]$	$\{y \leqslant 0\}$	$\{y \geqslant 0\}$
$[\{x, y\}, \{x, y\}]$	$\{y \leqslant 0, x = y\}$	
$[\{z\}, \{y, z\}]$		$\{y \geqslant 0, y \leqslant z\}$

Given **unify** we can now define abstract operators for set operations $\cap$ and $\cup$. Note that

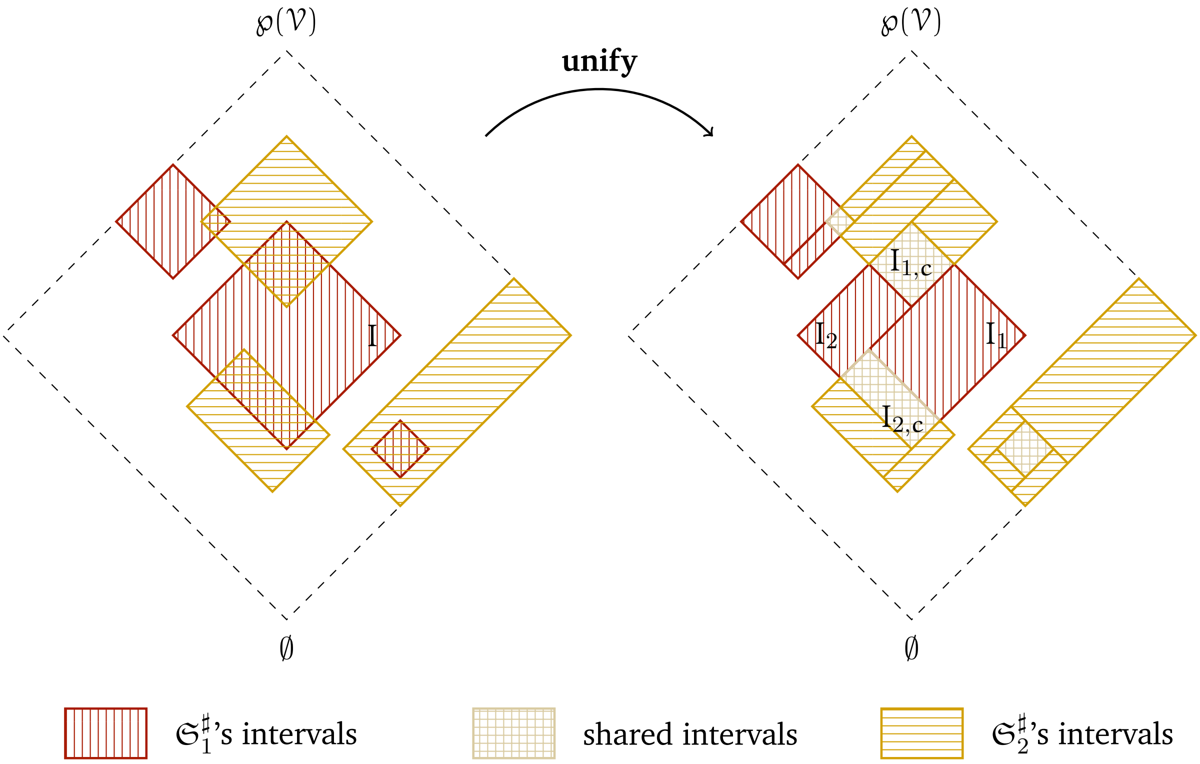


Figure 6.7: Illustration of the behavior of **unify**

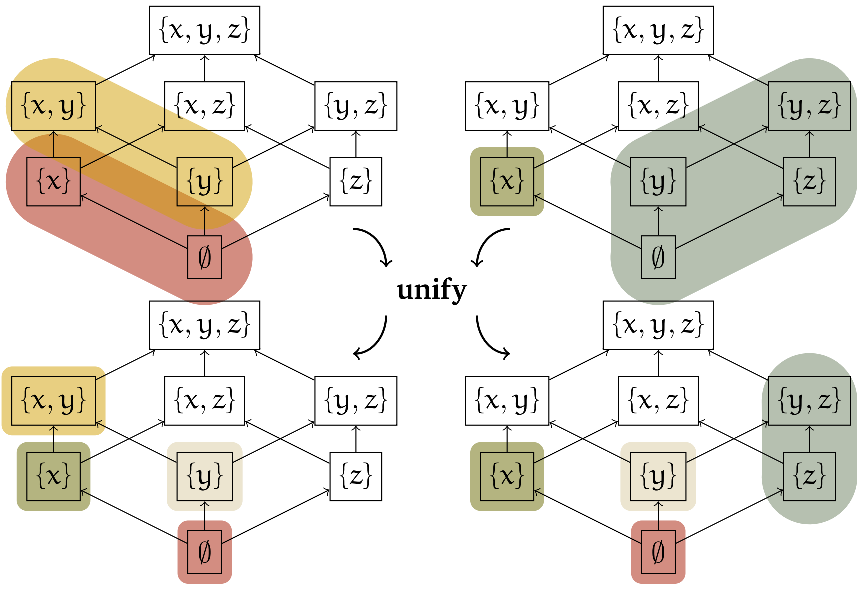


Figure 6.8: Illustration of example 6.15

unify_leq can also be used to unify abstract element, however it is less precise as it will enforce the use of the SVI partitioning of its right argument rather than subdividing both partitionings. These operators are computed by: (1) unifying the two abstract elements and (2) applying point-wise the according abstract operators on the underlying Clip abstraction.

Definition 6.24 (Set abstract operators)**.** We define $\sqcup_\wp$ and $\sqcap_\wp$ by:

$$
\begin{array}{l}
\mathfrak{S}_1^\sharp \sqcup_\wp \mathfrak{S}_2^\sharp \triangleq \\
\quad \textbf{let } (\mathfrak{S}_1^{\sharp\,\prime}, \mathfrak{S}_2^{\sharp\,\prime}) = \textbf{unify}(\mathfrak{S}_1^\sharp, \mathfrak{S}_2^\sharp) \textbf{ in} \\
\qquad \displaystyle\bigcup_{[l;u] \in \textbf{itvs}(\mathfrak{S}_1^{\sharp\,\prime}) \cup \textbf{itvs}(\mathfrak{S}_2^{\sharp\,\prime})} \begin{cases} \{\langle N_1^\sharp, l, u\rangle \sqcup \langle N_2^\sharp, l, u\rangle\} & \text{if } \langle N_1^\sharp, l, u\rangle \in \mathfrak{S}_1^{\sharp\,\prime} \text{ and } \langle N_2^\sharp, l, u\rangle \in \mathfrak{S}_2^{\sharp\,\prime} \\ \{\langle N^\sharp, l, u\rangle\} & \text{else if } \langle N^\sharp, l, u\rangle \in \mathfrak{S}_1^{\sharp\,\prime} \text{ or } \langle N^\sharp, l, u\rangle \in \mathfrak{S}_2^{\sharp\,\prime} \end{cases} \\
\mathfrak{S}_1^\sharp \sqcap_\wp \mathfrak{S}_2^\sharp \triangleq \\
\quad \textbf{let } (\mathfrak{S}_1^{\sharp\,\prime}, \mathfrak{S}_2^{\sharp\,\prime}) = \textbf{unify}(\mathfrak{S}_1^\sharp, \mathfrak{S}_2^\sharp) \textbf{ in} \\
\qquad \displaystyle\bigcup_{[l;u] \in \textbf{itvs}(\mathfrak{S}_1^{\sharp\,\prime}) \cap \textbf{itvs}(\mathfrak{S}_2^{\sharp\,\prime})} \{\langle N_1^\sharp, l, u\rangle \sqcap \langle N_2^\sharp, l, u\rangle\} \quad \text{where } \langle N_1^\sharp, l, u\rangle \in \mathfrak{S}_1^{\sharp\,\prime} \text{ and } \langle N_2^\sharp, l, u\rangle \in \mathfrak{S}_2^{\sharp\,\prime}
\end{array}
$$

We can see that these definitions are very close to the ones presented in the description of the partitioning abstraction from Section 6.3.1. Contrary to the CLIP abstraction, we see that the ℘-CLIP abstraction does not over-approximate the set of possible supports of the represented maps when computing an abstract union.

Example 6.16 (Using the running example)**.** We provided in the previous example (Example 6.15) the unification of the two abstract element: $\mathfrak{S}_1^\sharp = \{\langle\{x = 0\}, \emptyset, \{x\}\rangle, \langle\{y \leqslant 0, x = y\}, \{y\}, \{x, y\}\rangle\}$ and $\mathfrak{S}_2^\sharp = \{\langle\{x = 1\}, \{x\}, \{x\}\rangle, \langle\{y \geqslant 0, y \leqslant z\}, \emptyset, \{y, z\}\rangle\}$. As explained before $\sqcup_\wp$ and $\sqcap_\wp$ are both defined point-wise on the unification of $\mathfrak{S}_1^\sharp$ and $\mathfrak{S}_2^\sharp$. The following table provides the values of $\mathfrak{S}_1^\sharp$ and $\mathfrak{S}_2^\sharp$:

SVIs	$\mathfrak{S}_1^{\sharp\,\prime}$	$\mathfrak{S}_2^{\sharp\,\prime}$	$\mathfrak{S}_1^{\sharp\,\prime} \sqcap_\wp \mathfrak{S}_2^{\sharp\,\prime}$	$\mathfrak{S}_1^{\sharp\,\prime} \sqcup_\wp \mathfrak{S}_2^{\sharp\,\prime}$
$[\emptyset; \emptyset]$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$[\{x\}, \{x\}]$	$\{x = 0\}$	$\{x = 1\}$	$\bot$	$\{0 \leqslant x \leqslant 1\}$
$[\{y\}, \{y\}]$	$\{y \leqslant 0\}$	$\{y \geqslant 0\}$	$\{y = 0\}$	$\top$
$[\{x, y\}, \{x, y\}]$	$\{y \leqslant 0, x = y\}$	$\bot$		$\{y \leqslant 0, x = y\}$
$[\{z\}, \{y, z\}]$	$\bot$	$\{y \geqslant 0, y \leqslant z\}$	$\bot$	$\{y \geqslant 0, y \leqslant z\}$

Proposition 6.15 (Soundness of $\sqcup_\wp$ and $\sqcap_\wp$)**.** *$\sqcup_\wp$ and $\sqcap_\wp$ are sound* w.r.t. *to $\cup$ and $\cap$:*

$$
\begin{array}{c}
\forall \mathfrak{S}_1^\sharp, \mathfrak{S}_2^\sharp, \gamma_\wp(\mathfrak{S}_1^\sharp) \cup \gamma_\wp(\mathfrak{S}_2^\sharp) \subseteq \gamma_\wp(\mathfrak{S}_1^\sharp \sqcup_\wp \mathfrak{S}_2^\sharp) \\
\forall \mathfrak{S}_1^\sharp, \mathfrak{S}_2^\sharp, \gamma_\wp(\mathfrak{S}_1^\sharp) \cap \gamma_\wp(\mathfrak{S}_2^\sharp) \subseteq \gamma_\wp(\mathfrak{S}_1^\sharp \sqcap_\wp \mathfrak{S}_2^\sharp)
\end{array}
$$

See proof on page 155.

As emphasized by the previous definitions and examples, unification of abstract elements will make the abstraction converge towards a complete partitioning. This is due to the fact that unification splits SVIs rather that merge them. We have seen that the merging operation might induce an environment over-approximation. Merging two monomials M_1 and M_2 found in an abstract element $\mathfrak{S}^\sharp$ is done by computing $(\mathfrak{S}^\sharp \setminus \{M_1, M_2\}) \cup (M_1 \sqcup M_2)$. Notice however that the result is not well-defined as the SVI resulting from $(M_1 \sqcup M_2)$ might overlap other SVIs already present in $\mathfrak{S}^\sharp$. Therefore we define a **wf** function in Algorithm 6.5 which will ensure the well-formedness of a set of monomials, in the sense that no two SVIs overlap. As we try to reduce the number of monomials so as not to converge towards partitioning **wf** will not divide already present SVIs, but rather merge them.

Definition 6.25 (**merge** operator)**.** Using the previous remarks and the **wf** operator we can define the following **merge** operator. This operator takes an abstract element from $\mathfrak{M}_\wp^\sharp$ and two of its monomials and yields an abstract element where the two monomials have been merged.

$$\textbf{merge}(\mathfrak{S}^\sharp, U_1^\sharp, U_2^\sharp) = \textbf{wf}(\mathfrak{S}^\sharp \setminus \{U_1^\sharp, U_2^\sharp\} \cup \{U_1^\sharp \sqcup U_2^\sharp\})$$

Algorithm 6.5: wf operator

```
Input  : $S$ a set of monomials
Output: $\mathfrak{S}^\sharp$ a well-formed abstract element
res ← $S$;
while $\exists\langle N^\sharp, l, u\rangle \in \mathsf{res}, \langle N^{\sharp\prime}, l', u'\rangle \in \mathsf{res}, \langle N^\sharp, l, u\rangle \neq \langle N^{\sharp\prime}, l', u'\rangle \wedge [l; u] \cap [l'; u'] \neq \emptyset$ do
    res ← $(\mathsf{res} \setminus \{\langle N^\sharp, l, u\rangle, \langle N^{\sharp\prime}, l', u'\rangle\}) \cup \{\langle N^\sharp, l, u\rangle \sqcup \langle N^{\sharp\prime}, l', u'\rangle\}$;
end
return res;
```

Algorithm 6.6: reduce_monomials operator

```
Input  : $\mathfrak{S}^\sharp$ an abstract element, $n$ an integer
Output: an abstract element containing less than $n$ monomials, over-approximating $\mathfrak{S}^\sharp$
res ← $\mathfrak{S}^\sharp$;
while $|\mathfrak{S}^\sharp| > n$ do
    $(U_1^\sharp, U_2^\sharp)$ ← mergeable($\mathfrak{S}^\sharp$);
    res ← merge(res, $U_1^\sharp, U_2^\sharp$);
end
return res;
```

Proposition 6.16 (Soundness of **merge**). *For all $\mathfrak{S}^\sharp$, $U_1^\sharp$, $U_2^\sharp$, such that $\{U_1^\sharp, U_2^\sharp\} \subseteq \mathfrak{S}^\sharp$ we have $\gamma_\wp(\mathfrak{S}^\sharp) \subseteq \gamma_\wp(\textbf{merge}(\mathfrak{S}^\sharp, U_1^\sharp, U_2^\sharp))$.*

See proof on page 155.

We assume given a **mergeable** function that, given an abstract element $\mathfrak{S}^\sharp$ produces the next pair of monomials that should be merged. The **mergeable** function can be heuristically defined, the evaluation of the "mergeability" of two monomials $\langle N_1^\sharp, l_1, u_1\rangle$ and $\langle N_2^\sharp, l_2, u_2\rangle$ should take into account the two following: (1) how big is the difference between $[l_1; u_1] \cup [l_2, u_2]$ and $[l_1 \cap l_2; u_1 \cup u_2]$? (2) how much will $N_1^\sharp \boxed{\star} N_2^\sharp$ over-approximate the disjunction. Please note that the evaluation of the "mergeability" of two monomials can not be made independently from the rest of the monomials. Indeed after the merge operation is performed, the **wf** operator might collapse other monomials into the newly formed one. We will not provide here heuristics for the **mergeable** function. Indeed point (2) alone in the simpler case: how imprecise is the computation of $N_1^\sharp \sqcup^u N_2^\sharp$, has already been extensively studied (see e.g. [BHZ09, RHC18]).

Given the **mergeable** function, we can define a **reduce_monomials** operator (defined by Algorithm 6.6). **reduce_monomials**$(\mathfrak{S}^\sharp, n)$ computes an over-approximation of $\mathfrak{S}^\sharp$ that contains at most n monomials. This is done by iteratively merging pairs of monomials, these pairs are chosen by the **mergeable** function. Note that we need to ensure that abstract elements are well-formed, therefore we apply the **wf** operator after each merging of two monomials, this might induce more mergers. Therefore when calling **reduce_monomials**$(\mathfrak{S}^\sharp, n)$ with $|\mathfrak{S}^\sharp| = m$, the number of call to **mergeable** might be lower that $m - n - 1$.

Example 6.17. Let us consider the following abstract element: $\mathfrak{S}^\sharp = \{\langle \top^{\{x\}}, \{x\}, \{x\}\rangle, \langle\{x = y, x \geqslant 0\}, \{x, y\}, \{x, y\}\rangle, \langle\{x = z, x \geqslant 0\}, \{x, z\}, \{x, z\}\rangle\}$. $\mathfrak{S}^\sharp$ contains three monomials, defined respectively on the singleton SVIs $[\{x\}, \{x\}]$, $[\{x, y\}, \{x, y\}]$ and finally $[\{x, z\}, \{x, z\}]$. Thanks to the Clip abstraction, we have seen that the monomial defined on $[\{x, z\}, \{x, z\}]$ and the monomial defined on $[\{x, y\}, \{x, y\}]$ can be merged without losing numerical precision (note however that this will induce an environment over-approximation). Merging this two monomials yields: $\langle\{x = y, x = z\}, \{x\}, \{x, y, z\}\rangle$. The result from the **merge** operation can not be $\{\langle\top, \{x\}, \{x\}\rangle, \langle\{x = y, x = z, x \geqslant 0\}, \{x\}, \{x, y, z\}\rangle\}$ as this is not a well-formed abstract element, indeed $[\{x\}, \{x\}] \cap [\{x\}, \{x, y, z\}] = \{x\} \neq \emptyset$. Therefore these two monomials are merged, which requires the computation of $\top^{\{x\}} \boxed{\star} \{x = y, x = z, x \geqslant 0\} = \{x = y, x = z\}$. Finally the resulting abstract element is $\mathfrak{S}^{\sharp\,\prime} = \langle\{x = y, x = z\}, \{x\}, \{x, y, z\}\rangle$, note that $\gamma_{\wp}(\mathfrak{S}^{\sharp\,\prime})$ numerically over-approximates environments defined on $\{x, y\}$ and $\{x, z\}$.

6.6.4 Widening operator

As the lattice $(\mathfrak{M}^{\sharp}_{\wp}, \sqsubseteq_{\wp})$ has infinite height, we need to define a widening operator in order to ensure the stabilization of infinitely increasing chains. As for union and intersection we can rely on the **unify** operator to ensure that both abstract elements are defined on the same set of SVIs, and then pair-wise apply the widening operator to the underlying domain. However consider the following sequence:

$$\begin{aligned}
X^{\sharp}_0 &= \{\langle \_, \emptyset, \{x_0\}\rangle\},\\
&\dots\\
X^{\sharp}_n &= \{\langle \_, \emptyset, \{x_0, \dots, x_n\}\rangle\},\\
&\dots
\end{aligned}$$

We then have:

$$\begin{array}{lll}
Y^{\sharp}_1 = & X^{\sharp}_0 \triangledown X^{\sharp}_1 = & \{\langle \_, \emptyset, \{x_0\}\rangle,\\
 & & \langle \_, \{x_1\}, \{x_0, x_1\}\rangle\},\\
\dots & & \\
Y^{\sharp}_n = & Y^{\sharp}_{n-1} \triangledown X^{\sharp}_n = & \{\langle \_, \emptyset, \{x_0\}\rangle,\\
 & & \dots\\
 & & \langle \_, \{x_n\}, \{x_0, \dots, x_n\}\rangle\}
\end{array}$$

We see that the number of monomials is diverging. Therefore in addition to the stabilization of the numerical constraints, we also need to ensure stabilization of the set of SVIs.

We now propose a widening operator which “freezes” the number of monomials when it is first called. The idea is the following: instead of splitting monomials to unify abstract elements, we merge them. This is the role of the loop at line 3 in Algorithm 6.7. Figure 6.9 illustrates this merging phase. In both abstract elements we merge monomials the SVIs of which intersect a common SVI in the other abstract element. Indeed when computing $\mathfrak{S}^{\sharp}_1 \triangledown_{\wp} \mathfrak{S}^{\sharp}_2$, we need to discover a one to (at most) one relation from the monomials of $\mathfrak{S}^{\sharp}_1$ to the monomials of $\mathfrak{S}^{\sharp}_2$. This one to (at most) one relation is computed by iteratively merging elements until each momonial from $\mathfrak{S}^{\sharp}_2$ intersects at most one element from $\mathfrak{S}^{\sharp}_1$ and reciprocally. There are then two cases to consider:

- Every element of $\mathfrak{S}^{\sharp}_2$ is contained in at least one element from $\mathfrak{S}^{\sharp}_1$ (actually there is at most one such element thanks to the merging phase). In this case we apply a pair-wise widening between monomials from $\mathfrak{S}^{\sharp}_1$ and monomials from $\mathfrak{S}^{\sharp}_2$.
- At least one monomial from $\mathfrak{S}^{\sharp}_2$ is not contained in a monomial from $\mathfrak{S}^{\sharp}_1$. In this case we over-approximate to $\top_{\wp}$.

Definition 6.26 ($\triangledown_{\wp}$ operator)**.** Algorithm 6.7 provides the definition of a $\triangledown_{\wp}$ operator.

Proposition 6.17 (Soundness and stabilizing property of $\triangledown_{\wp}$)**.** *For all $\mathfrak{S}^{\sharp}_1$ and $\mathfrak{S}^{\sharp}_2$ in $\mathfrak{M}^{\sharp}_{\wp}$,*

$$\gamma_{\wp}(\mathfrak{S}^{\sharp}_1) \subseteq \gamma_{\wp}(\mathfrak{S}^{\sharp}_1 \triangledown_{\wp} \mathfrak{S}^{\sharp}_2) \textit{ and } \gamma_{\wp}(\mathfrak{S}^{\sharp}_1) \subseteq \gamma_{\wp}(\mathfrak{S}^{\sharp}_1 \triangledown_{\wp} \mathfrak{S}^{\sharp}_2)$$

Moreover $\triangledown_{\wp}$ stabilizes sequences.

See proof on page 155.

Algorithm 6.7: $\nabla_\wp$ operator

```
Input  : $\mathfrak{S}^\sharp_1, \mathfrak{S}^\sharp_2$ two abstract elements
Output: A stabilizing over-approximation of $\gamma_\wp(\mathfrak{S}^\sharp_1) \cup \gamma_\wp(\mathfrak{S}^\sharp_2)$
merg$_1$ $\leftarrow \mathfrak{S}^\sharp_1$;
merg$_2$ $\leftarrow \mathfrak{S}^\sharp_2$;
while (1)     $\exists \mathsf{U}^\sharp_1, \mathsf{U}^{\sharp\,\prime}_1 \in \mathtt{merg}_1, \exists \mathsf{U}^\sharp_2 \in \mathtt{merg}_2,\ \boldsymbol{itv}(\mathsf{U}^\sharp_1) \cap \boldsymbol{itv}(\mathsf{U}^\sharp_2) \neq \emptyset \wedge \boldsymbol{itv}(\mathsf{U}^{\sharp\,\prime}_1) \cap \boldsymbol{itv}(\mathsf{U}^\sharp_2) \neq \emptyset$
      or (2)  $\exists \mathsf{U}^\sharp_2, \mathsf{U}^{\sharp\,\prime}_2 \in \mathtt{merg}_2, \exists \mathsf{U}^\sharp_1 \in \mathtt{merg}_1,\ \boldsymbol{itv}(\mathsf{U}^\sharp_2) \cap \boldsymbol{itv}(\mathsf{U}^\sharp_1) \neq \emptyset \wedge \boldsymbol{itv}(\mathsf{U}^{\sharp\,\prime}_2) \cap \boldsymbol{itv}(\mathsf{U}^\sharp_1) \neq \emptyset$
  do
    if (1) then
        merg$_1$ $\leftarrow$ merge(merg$_1$, $\mathsf{U}^\sharp_1, \mathsf{U}^{\sharp\,\prime}_1$)
    else
        merg$_2$ $\leftarrow$ merge(merg$_2$, $\mathsf{U}^\sharp_2, \mathsf{U}^{\sharp\,\prime}_2$)
end
res $\leftarrow \emptyset$;
if $\forall \mathrm{I}_2 \in \boldsymbol{itvs}(\mathtt{merg}_2), \exists \mathrm{I}_1 \in \boldsymbol{itvs}(\mathtt{merg}_1), \mathrm{I}_2 \subseteq \mathrm{I}_1$ then
    res $\leftarrow \emptyset$;
    foreach $\mathsf{U}^\sharp_1 \in \mathtt{merg}_1$ do
        res $\leftarrow$ res $\cup \left\{ \begin{array}{ll} \mathsf{U}^\sharp_1 \nabla \mathsf{U}^\sharp_2 & \text{if } \exists \mathsf{U}^\sharp_2 \in \mathfrak{S}^\sharp_2,\ \mathbf{itv}(\mathsf{U}^\sharp_2) \subseteq \mathbf{itv}(\mathsf{U}^\sharp_1) \\ \mathsf{U}^\sharp_1 & \text{otherwise} \end{array} \right\}$
    done
    return res
else
    return $\top_\wp$
```

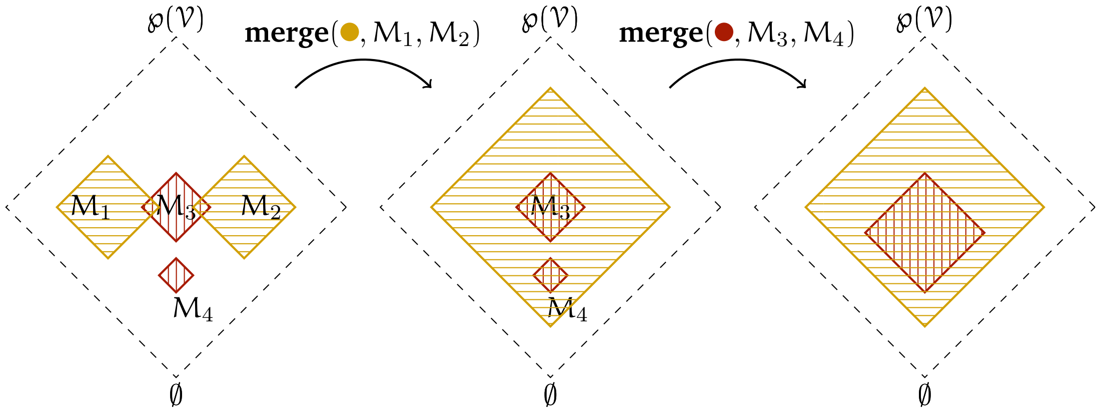


Figure 6.9: Merging phase of $\nabla_\wp$

6.6.5 Abstract transformers

Now that we have provided abstract operators on $\mathfrak{M}^\sharp_\wp$ for classical set operations as well as a widening operator, we focus on the definition of abstract transformers for the two classical transformation $\mathbb{S}[\![\mathtt{Assume}(c)]\!]$ and $\mathbb{S}[\![\mathtt{x} \leftarrow e]\!]$. As the Clip abstraction already provides sound abstract transformers defined on $\mathfrak{M}^\sharp$, our lifting to the $\wp$-Clip abstraction will amount to a point-wise application of those operators (this point-wise lifting is the same as the one introduced in Section 6.3.1 for the definition of abstract transformers on the partitioning domain). Note however that the underlying operator modifies the SVI on which a monomial is defined. Therefore the resulting set of monomials might not be well-formed, in the sense that it may contain monomials with overlapping SVIs. In the previous section, we provided a **wf** operator which soundly transformed a set of monomials into a well-formed abstract element. However this operator was designed with the intent of reducing the number of monomials. We now provide in Algo-

Algorithm 6.8: $\mathbf{wf}_\mathrm{p}$ operator

```
Input  : $S$ a set of monomials
Output: $\mathfrak{S}^\sharp$ a well-formed abstract element
res $\leftarrow S$;
while $\exists \langle N^\sharp, l, u\rangle \in$ res, $\langle N^{\sharp\prime}, l', u'\rangle \in$ res, $\langle N^\sharp, l, u\rangle \neq \langle N^{\sharp\prime}, l', u'\rangle \wedge [l;u] \cap [l';u'] \neq \emptyset$ do
    $[a;b] \leftarrow [l;u] \cap [l';u']$;
    diff$_1$ $\leftarrow$ itv_diff$([l;u],[a;b])$;
    diff$_2$ $\leftarrow$ itv_diff$([l';u'],[a;b])$;
    res $\leftarrow$ res $\setminus \{\langle N^\sharp, l, u\rangle, \langle N^{\sharp\prime}, l', u'\rangle\}$;
    res $\leftarrow$ res $\cup \bigcup_{[c;d]\in \mathtt{diff}_1} \langle N^\sharp_{|d}, c, d\rangle \cup \bigcup_{[c;d]\in \mathtt{diff}_2} \langle N^{\sharp\prime}_{|d}, c, d\rangle \cup \{\langle N^\sharp_{|b} \sqcup^b N^{\sharp\prime}_{|b}, a, b\rangle\}$
end
return res;
```

rithm 6.8 a $\mathbf{wf}_\mathrm{p}$ operator which aims at precision rather than conciseness. This operator looks for two monomials, the SVI of which ($[l;u]$ and $[l';u']$) have a non empty intersection. It then replaces these two monomials by: a monomial defined on the intersection $[l;u] \cap [l';u']$ and monomials for each of the SVIs from the SVI difference $[l;u] \setminus [l';u']$, similarly for $[l';u'] \setminus [l;u]$.

Proposition 6.18 (Soundness of Algorithm 6.8)**.** *Let S be a set of monomials, we have:*

$$\bigcup_{\mathfrak{U}^\sharp \in S} \gamma(\mathfrak{U}^\sharp) \subseteq \gamma_\wp(\boldsymbol{wf}_\mathrm{p}(S))$$

See proof on page 156.

Definition 6.27 (Abstract transformers on $\mathfrak{M}^\sharp_\wp$)**.** We define the following abstract transformers:

$$\mathbb{S}^\sharp[\![\mathtt{Assume(c)}]\!]_\wp(\mathfrak{S}^\sharp) = \mathbf{wf}_\mathrm{p}(\{\mathbb{S}^\sharp[\![\mathtt{Assume(c)}]\!](\mathfrak{U}^\sharp) \mid \mathfrak{U}^\sharp \in \mathfrak{S}^\sharp\})$$
$$\mathbb{S}^\sharp[\![\mathtt{x} \leftarrow e]\!]_\wp(\mathfrak{S}^\sharp) = \mathbf{wf}_\mathrm{p}(\{\mathbb{S}^\sharp[\![\mathtt{x} \leftarrow e]\!](\mathfrak{U}^\sharp) \mid \mathfrak{U}^\sharp \in \mathfrak{S}^\sharp\})$$

Proposition 6.19 (Soundness of the abstract transformers)**.** $\mathbb{S}^\sharp[\![\mathtt{x} \leftarrow e]\!]_\wp$ *and* $\mathbb{S}^\sharp[\![\mathtt{Assume(c)}]\!]_\wp$ *are sound transformers.*

See proof on page 156.

6.6.6 Conclusion

As partitioning is often used to lift classical numerical domains, we felt that the Clip abstraction should provide a way to be made more precise as well. For this reason, we provided in this section the $\wp$-Clip abstraction. The precision of this abstraction, contrary to the partitioning abstraction, can be tuned dynamically during the analysis, providing, at one end of the spectrum the partitioning abstraction and at the other end the Clip abstraction.

6.7 Unbounded maps

In the introduction to this chapter (Section 6.1) we mentioned not only the need for heterogeneous numerical environments but also for unbounded numerical environments. In this section we revisit the classical lifting of bounded numerical domains to unbounded numerical domains by means of summarizing variables (see [GDD$^+$04]), however we will here show how this lifting can be defined when starting from heterogeneous environments. Summarizing is based on the folding of several concrete objects (a potentially infinite number) to an abstract element which summarizes all concrete objects. The folding is encoded in a function f mapping summarized variables to the set of concrete variables they abstract. The concretization function γ from the underlying domain (yielding back partial environments $\mathcal{V}' \nrightarrow \mathbb{I}$) can then be parameterized by this *summarizing function* to produce environments in $\mathcal{V} \nrightarrow \mathbb{I}$. Given a mapping from summary variables: $\mathcal{V}'$ to sets of concrete variables $f \in \mathcal{V}' \to \wp(\mathcal{V})$ where $f(v_1) \cap f(v_2) \neq \emptyset \Rightarrow v_1 = v_2$, we start by defining the *collapsing* of a partial map $\rho \in \mathcal{V} \nrightarrow \mathbb{I}$ under a summarizing function f. This collapsing will be used in the definition of the parameterized concretization function.

Definition 6.28 (Environment collapsing $\downarrow_\square (\square)$)**.** We define the collapse of an environment defined on concrete variables (a potentially infinite number of them) under a summarizing function, this produces a set of environments defined on summary variables that meet the constraints from the original environment.

$$\downarrow_f (\rho) = \{\rho' \in \mathcal{V}' \nrightarrow \mathbb{I} \mid \forall v' \in \mathcal{V}',\ \rho'(v') = \begin{cases} \textbf{undefined} & \text{if } f(v') \cap \textbf{def}(\rho) = \emptyset \\ \rho(v) & \text{otherwise if } \exists v \in f(v') \cap \textbf{def}(\rho) \end{cases} \}$$

Under a summary function f the collapse of an environment ρ therefore produces environments ρ' that:

- are undefined on a variable v' as soon as ρ is undefined on every concrete variable associated to v' by the summarizing
- is defined on a variable v as soon as ρ is defined on some concrete variable associated to v, moreover in such a case v' has the same image by ρ' as the image of v by ρ.

Example 6.18. Consider $\mathcal{V}' = \{x, y, z, t\}$ and $\mathcal{V} = \{a, b, c, d, g, h\}$, the environment $\rho = (a \mapsto 0, b \mapsto 1, c \mapsto 2, d \mapsto 3, h \mapsto 4)$ and finally the summarizing function $f = (x \mapsto \{a\}, y \mapsto \{b, c\}, z \mapsto \{d\}, t \mapsto \{g\})$. Collapsing environment ρ under f yields the set of environments: $(x \mapsto 0, y \mapsto 1, z \mapsto 3)$ and $(x \mapsto 0, y \mapsto 2, z \mapsto 3)$.

Given a summarizing function $f \in \mathcal{V}' \nrightarrow \wp(\mathcal{V})$, and a concretization function γ yield concrete elements in $\mathcal{V}' \nrightarrow \mathbb{I}$ we can now lift γ in a concretization function producing concrete values in $\mathcal{V} \nrightarrow \mathbb{I}$.

Definition 6.29 (Parameterized concretization $\gamma[f]$)**.** Given a summarizing function $f \in \mathcal{V}' \nrightarrow \wp(\mathcal{V})$ and a function $\gamma : \mathcal{N}^\sharp \to \wp(\mathcal{V}' \nrightarrow \mathbb{I})$ concretizing abstract elements from $\mathcal{N}^\sharp$ we define:

$$\gamma[f] = \lambda N^\sharp.\{\rho \in \mathcal{V} \nrightarrow \mathbb{I} \mid \downarrow_f (\rho) \subseteq \gamma(N^\sharp)\}$$

We have $\gamma[f] \in \mathcal{N}^\sharp \to \wp(\mathcal{V} \nrightarrow \mathbb{I})$.

Given an abstract element $N^\sharp$ its concretization under $\gamma[f]$ is the set of environments the collapsing of which is contained in $\gamma(N^\sharp)$.

Example 6.19. As a first example, we consider:

- that $\gamma = \gamma_{P_Q}^{\{x,y,z\}}$, the classical concretization function of polyhedra over $\mathbb{Z}$ defined over the environment $\{x, y, z\}$;
- that $\mathcal{V} = \{x_1, y_1, y_2, z_1, z_2, t_1\}$ and $\mathcal{W} = \{x, y, z\}$;
- that we are given a summarizing function: $f = (x \mapsto \{x_1\}, y \mapsto \{y_1, y_2\}, z \mapsto \{z_1, z_2\})$;
- finally that $N^\sharp = \{y \geqslant x, z \leqslant 3\}$.

Let us describe $\gamma[f](N^\sharp)$. An element $\rho \in \mathcal{V} \nrightarrow \mathbb{I}$ is in $\gamma[f](N^\sharp)$ if and only if $\downarrow_f (\rho) \subseteq \gamma(N^\sharp)$. We have $\rho' \in \downarrow_f (\rho) \Rightarrow \textbf{def}(\rho') = \{v' \mid f(v') \cap \textbf{def}(\rho) \neq \emptyset\}$. Moreover we have necessarily $\textbf{def}(\rho') = \{x, y, z\}$, this ensures that $\textbf{def}(\rho) \cap f(x) \neq \emptyset$, $\textbf{def}(\rho) \cap f(y) \neq \emptyset$ and $\textbf{def}(\rho) \cap f(z) \neq \emptyset$. Therefore if $\rho \in \gamma[f](N^\sharp)$, $\textbf{def}(\rho)$ contains $\{x_1\}$, at least one element among $\{y_1, y_2\}$, and at least one element among $\{z_1, z_2\}$. Therefore let ρ be one such environment, let $X = \{\rho(x_1)\}$, $Y = \{\rho(y_1), \rho(y_2)\}$ and

$Z = \{\rho(z_1), \rho(z_2)\}$[3], we then have $\forall v_x \in X, v_y \in Y, v_z \in Z, (x \mapsto v_x, y \mapsto v_y, z \mapsto v_z) \in \gamma(N^\sharp)$. This ensures that $\forall v_x \in X, \forall v_y \in Y, \forall v_z \in Z, v_y \geqslant v_x \wedge v_z \leqslant 3$. Finally we get that (and these are sufficient conditions):

$$
\begin{aligned}
A \triangleq\, & \{x_1 \mapsto \alpha, y_{\{1,2\}} \mapsto \beta, z_{\{1,2\}} \mapsto \delta \mid \alpha \geqslant \beta \wedge \delta \leqslant 3\} \\
& \cup \{x_1 \mapsto \alpha, y_1 \mapsto \beta, y_2 \mapsto \beta', z_1 \mapsto \delta \mid \alpha \geqslant \beta \wedge \alpha \geqslant \beta' \wedge \delta \leqslant 3\} \\
& \cup \{x_1 \mapsto \alpha, y_1 \mapsto \beta, z_1 \mapsto \delta, z_2 \mapsto \delta' \mid \alpha \geqslant \beta \wedge \delta' \leqslant 3 \wedge \delta \leqslant 3\} \\
& \cup \{x_1 \mapsto \alpha, y_1 \mapsto \beta, y_2 \mapsto \beta', z_1 \mapsto \delta, z_2 \mapsto \delta' \mid \alpha \geqslant \beta \wedge \alpha \geqslant \beta' \wedge \delta' \leqslant 3 \wedge \delta \leqslant 3\} \\
\gamma[f](N^\sharp) =\, & A \cup \{\rho \uplus (t_1 \mapsto \epsilon) \mid \rho \in A \wedge \epsilon \in \mathbb{Z}\}
\end{aligned}
$$

Remark 6.11. The previous example showed that

- a variable in $\mathcal{V}$ that is not in the image of the summarizing function will be contained or not (without constraints) in the environment of the resulting concrete maps.
- if a variable v summarizes a set S of variables, the presence of v in the original concretization does not imply the presence of all variables from S, only of at least one.

Both are design choices motivated by reasons which will be made clear in the use case of the following chapter. Indeed the tree abstraction will track an over-approximation of the set of variables, thus removing the need for the numerical domain to do it; moreover summarization variables will be used for sets of variables which might not (and in some cases will not) be present at the same time in the concretization, thus justifying the second design choice.

Example 6.20. As a second example, we consider:

- that the underlying abstraction is the octagon abstraction lifted with the CLIP abstraction.
- that $\mathcal{V}$, $\mathcal{V}'$ and f are as defined in Ex 6.18
- finally that $N^\sharp = \langle\{x \leqslant y\}, \{x\}, \{x, y\}\rangle$.

we have: $\gamma(N^\sharp) = \{(x \mapsto \alpha) \mid \alpha \in \mathbb{Z}\} \cup \{(x \mapsto \alpha, y \mapsto \beta) \mid \alpha \leqslant \beta\}$. We have: $\rho \in \gamma[f](N^\sharp) \Leftrightarrow \downarrow_f (\rho) \subseteq \gamma(N^\sharp) \Rightarrow \forall \rho' \in \downarrow_f (\rho), \{x\} \subseteq \mathbf{def}(\rho') \subseteq \{x, y\}$. Therefore if we assume m defined on d then $f(z) \cap \mathbf{def}(\rho) \neq \emptyset$ hence there would be an element in $\downarrow_f (\rho)$ defined on z. Hence m is not defined on d, similarly for g. Moreover $\forall \rho' \in \downarrow_f (\rho), \{x\} \subseteq \mathbf{def}(\rho')$ implies that ρ is defined on a. Finally:

$$
\begin{aligned}
A \triangleq\, & \{(a \mapsto \alpha) \mid \alpha \in \mathbb{Z}\} \\
& \cup \{(a \mapsto \alpha, b \mapsto \beta) \mid \alpha \leqslant \beta\} \\
& \cup \{(a \mapsto \alpha, c \mapsto \beta) \mid \alpha \leqslant \beta\} \\
& \cup \{(a \mapsto \alpha, b \mapsto \beta, c \mapsto \delta) \mid \alpha \leqslant \beta \wedge \alpha \leqslant \delta\} \\
\gamma[f](N^\sharp) =\, & A \cup \{\rho \uplus (h \mapsto \epsilon) \mid \rho \in A \wedge \epsilon \in \mathbb{Z}\}
\end{aligned}
$$

The abstract domains we will define in the following sections will employ this summarizing framework. The manipulation of summarized variables requires the definition of a $\mathbf{fold}(E, x, \mathcal{S})$ (resp. $\mathbf{expand}(E, x, \mathcal{S})$) operator yielding a new environment where x is used as a summary variable for $\mathcal{S}$ (resp. where a summary variable x is desummarized into a set of variables $\mathcal{S}$). Let $\mathcal{S}$ and $\mathcal{S}'$ be two finite sets of elements such that $\mathcal{S}' \cap \mathcal{S} \subseteq \{x\}$, the classical definition of the fold and expand function on numerical abstract domains is ($N^\sharp \in \mathcal{N}^S$): $\mathbf{expand}(N^\sharp, x, \mathcal{S}') = \bigsqcap_{v \in \mathcal{S}'} N^\sharp[x \mapsto v]_{|(\mathcal{S} \setminus \{x\}) \cup \mathcal{S}'}$ and $\mathbf{fold}(N^\sharp, x, \mathcal{S}') = \bigsqcup_{v \in \mathcal{S}'} N^\sharp[v \mapsto x]_{|(\mathcal{S} \setminus \mathcal{S}') \cup \{x\}}$ (which generalize the one introduced in [GDD+04]). Following these definitions, these operators are lifted on elements of $\mathfrak{M}^\sharp$.

[3] These definitions abuse notations, indeed ρ is defined on at least one element from $\{y_1, y_2\}$, not necessarily on both, the set Y is the set containing all possible images of y_1 and y_2, similarly for z.

6.8 Conclusion

We provided an abstraction able to represent heterogeneous sets of environments. This abstraction generalizes to relational domains an approach already used in the particular case of intervals. The precision of this abstraction, in particular for the join operator, comes from iterated attempts to assert constraints taken from already known invariants.

In addition to the definition of the CLIP abstraction we have shown how its definition can be generalized in the $\wp$-CLIP abstraction, so that state of the art results (partitioning at one end of the spectrum, and the folk technique for non relational domains at the other end) can be seen as a special case of our work.

The definition of an abstract domain representing trees with numerically labeled leaves (which we want to use as a pre-analysis guiding framing, in a C modular iterator), requires the ability to represent numerical constraints on the leaves of a set of trees. As we want to represent sets of trees that do not necessarily share the same numerical leaves positions, we need to represent heterogeneous environments. Thanks to the results from this chapter, we can move on to the definition of this tree abstraction in the next chapter.

Chapter 7

Tree abstraction

7.1 Introduction

7.1.1 Trees and terms

In Chapter 5, we emphasized that the quality of a modular analyzer (in terms of reusability and precision of the inferred summaries) relies heavily on the quality of the framing performed by the analyzer. Program 7.1 describes an `append` function in C, this function adds an integer at the end of a linked list. The infinite set of unbounded terms of the form `*(*( …*(head + 4) …+ 4) + 4)` (shown in Figure 7.1) represents memory zones that are used by the `append` function. Having a pre-analysis capable of inferring and proving that this set of terms represents the only memory zones reachable in Program 7.1 would allow us to soundly forget all the `data` fields of the linked list. This would improve: (1) precision: as we know that they are not modified by the function call, (2) body analysis efficiency: as the input state is reduced (the cost of atomic operations in the underlying numerical domains is guided by the number of variables), (3) reusability: as constraints on the usage of the first analysis are relaxed by the removal of constraints.

Inferring this set of terms requires the discovery of invariants having the form of maps from program variables to sets of terms. In this chapter we will provide an abstract domain able to infer such invariants.

The remainder of this introduction provides two other case studies showing how the ability to infer environments mapping variables to tree-shaped structures can be used in program analysis.

Symbolic relations. Program 7.2 is a C function computing an approximation of the golden ratio (as it is the limit of the sequence $r_0 = 1$, $r_{n+1} = 1 + \frac{1}{r_n}$). As classical numerical domains can not represent such numerical relations, methods were proposed to track symbolic equality between expressions (see [Min06b]). However such methods can not handle the unbounded iteration of Program 7.2. The set of reachable states at the end of Program 7.2 can be expressed by $\mathsf{r} = 1 + 1/(1 + 1/\ldots 1\ldots)$ with depth n. The abstraction presented in this chapter will (1) provide a way to join two symbolic environments where variables are bound to different expressions; (2) define a widening operator stabilizing sequences of sets of terms. An illustration of the behavior of the domain defined in this section on the value of `r` in Program 7.2 can be found in Figure 7.2.

Numerical environment. Consider now the OCaml Program 7.3, we want to prove that the `assert false` expression is never reached. This program builds a list of size $2 * \mathsf{n}$ with alternating values $\mathsf{x} + 1$ and $\mathsf{x} - 1$. The assertion states that the head of the list is $\mathsf{x} + 1$. After the definition of `t` there are two types of reachable states. (1) Those that have not gone through recursive calls $(\mathsf{t} \mapsto \texttt{[]}, \mathsf{x} \mapsto \mathbb{Z}, \mathsf{n} \mapsto 0)$, and (2) those that have gone through at least one iteration of the loop: $(\mathsf{t} \mapsto \texttt{[}a_1\texttt{;}\ a_2\texttt{;}\ a_3\texttt{;}\ \ldots\texttt{]}, x \mapsto \alpha, \mathsf{n} > 0, a_1 \mapsto \alpha + 1, a_2 \mapsto \alpha - 1, a_3 \mapsto \alpha + 1)$, where

```c
typedef struct node
{
  int data;
  struct node* next;
} node;

node* append(node* head, int data)
{
  if (head==NULL) {
    return (create(data, NULL));
  } else {
    node *cursor=head;
    while(cursor->next != NULL)
      cursor=cursor->next;
    node* new_node=create(data,NULL);
    cursor->next=new_node;
    return head;
  }
}
```

Program 7.1: Append to list in C

```c
float golden_ratio(int n) {
  int i = 0;
  float r = 1;
  while (i < n) {
    r = 1 + 1 / r;
    i += 1;
  }
  return r;
}
```

Program 7.2: Golden ratio in C

```ocaml
let rec f x n =
  match n with
  | 0 -> []
  | _ -> (x+1)::(x-1)::(f x (n-1))

let () =
  (*Assume x:int and n:int>=0*)
  let t = f x n in
  match t with
  | [] -> ()
  | p :: q when p > x -> ()
  | _ -> assert false
```

Program 7.3: List type in Ocaml

$\alpha \in \mathbb{Z}$. Therefore we need to be able to keep numerical relations between the parametric and unbounded number of numeric values appearing in t and numeric variables from the program. We can see that results from Chapter 6 will be useful to represent these numerical relations.

7.1.2 Outline of the chapter

The remainder of the chapter is organized as follow: We start, in Section 7.2, by presenting the concrete semantics we want to abstract. As the abstraction defined in this chapter and the state of the art method from [GGLM12] for the representation of sets of trees with numerically labeled leaves both use tree regular languages, we will recall, in Section 7.3, their definition and the properties used in subsequent sections. We can then provide a description of state of the art in Section 7.4. Section 7.3 will also recall some results on regular languages as these will be used in the definition of our abstraction. In Section 7.5 we build a first abstraction which forgets numerical values and focuses on abstracting tree shapes, this will be done using tree automata. In Section 7.6, we define a second abstraction which aims at precisely representing numerical constraints between trees and program variables. In Section 7.8 we provide remarks on the implementation and results of the analyzer.

Due to the high number of abstract operators provided in this chapter, not all of them come with a soundness proof. We consider as future works the completion of these proofs.

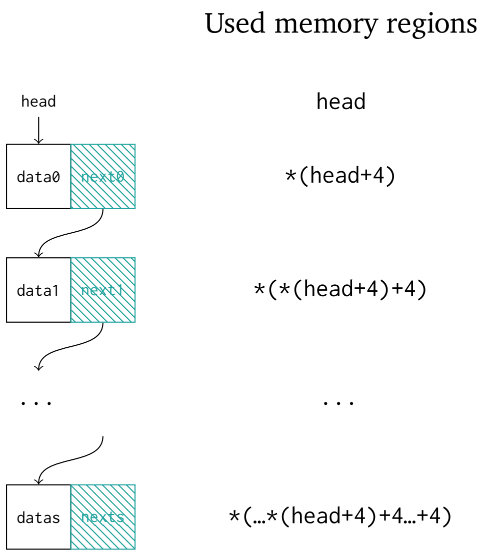


Figure 7.1: Reachable memory zones in Program 7.1

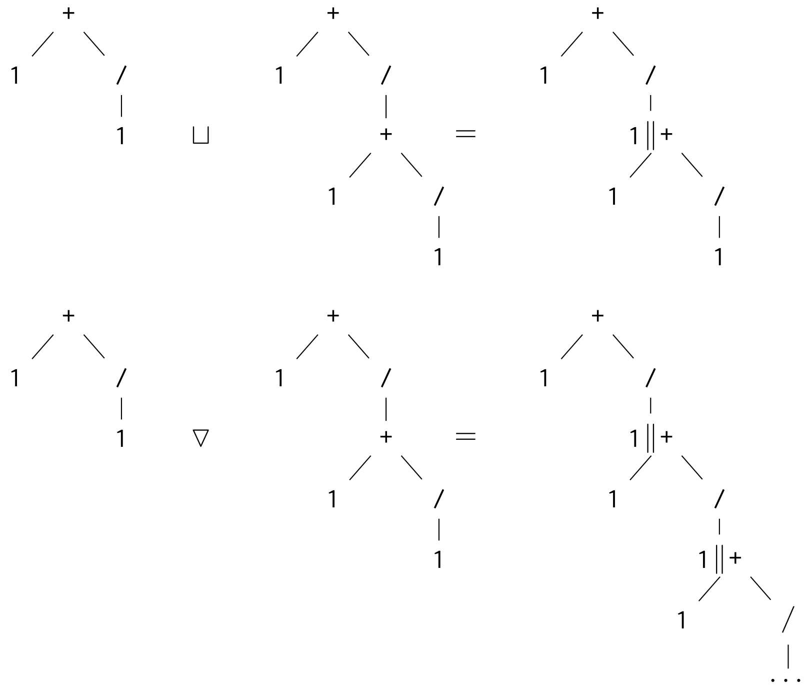


Figure 7.2: Joining and widening of Program 7.2

7.2 Syntax and concrete semantics

7.2.1 Terms and numerical terms

In this subsection we provide definitions for terms, numerical terms, as well as some of the notations used thereafter. In the following $\mathbb{I} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$.

Definition 7.1 (Alphabet, ranked alphabet, arity)**.** An *alphabet* $\mathcal{F}$ is a finite set, a *ranked alphabet* is a pair $\mathcal{R} = (\mathcal{F}, a)$ where $\mathcal{F}$ is an alphabet and $a \in \mathcal{F} \to \mathbb{N}$. For $f \in \mathcal{F}$, we call *arity* of f the value $a(f)$. We assume that $\mathbb{I}$ and $\mathcal{F}$ are disjoint. In the following, $\mathcal{F}_n$ denotes the subset of $\mathcal{F}$ of arity n.

Definition 7.2 (Terms)**.** When $\mathcal{R}$ contains at least one symbol of arity 0, we define *terms* over $\mathcal{R}$ (denoted $T(\mathcal{R})$) to be the smallest set such that:

- $\forall p \geqslant 0,\ f \in \mathcal{F}, t_1,\ \ldots, t_p \in T(\mathcal{R}),\ a(f) = p \Rightarrow f(t_1, \ldots, t_p) \in T(\mathcal{R})$

Definition 7.3 (Numerical terms)**.** We define the set of *numerical terms* over $\mathcal{R}$ (denoted $T_{\mathbb{I}}(\mathcal{R})$) to be the smallest set such that:

- $\mathbb{I} \subseteq T_{\mathbb{I}}(\mathcal{R})$
- $\forall p \geqslant 0,\ f \in \mathcal{F}, t_1,\ \ldots, t_p \in T_{\mathbb{I}}(\mathcal{R}),\ a(f) = p \Rightarrow f(t_1, \ldots, t_p) \in T_{\mathbb{I}}(\mathcal{R})$

Remark 7.1. Moreover given a term $t \in T(\mathcal{R})$ we denote $f = \textbf{head}(t) \in \mathcal{F}$ and $\textbf{sons}(t)$ the possibly empty tuple $(t_1, \ldots, t_n)$ of elements of $T(\mathcal{R})$ such that $t = f(t_1, \ldots, t_n)$.

Example 7.1. Consider the ranked alphabet $\mathcal{R} = \{\texttt{*}(1), \texttt{\&}(1), \texttt{+}(2), \texttt{x}(0)\}$, $u(n)$ means that symbol u has arity n. Then $\texttt{\&x} \in T(\mathcal{R})$, but $\texttt{*(\&x+4)} \in T_{\mathbb{I}}(\mathcal{R})$, and $\texttt{*(\&x+4)} \notin T(\mathcal{R})$. Using this alphabet we can model C pointer arithmetic.

Example 7.2. $U = \{+(x, y) \mid x \leqslant y\}$ and $V = \{+(x, +(z, y)) \mid x \leqslant y \wedge z \leqslant y\}$ are two sets of numerical terms over $\mathcal{R} = \{+(2)\}$ which we will use as running examples in the following sections of this chapter.

7.2.2 Syntax of the tree manipulating language

We consider here a small imperative language, manipulating numerical variables $\mathcal{V}$, extending with tree manipulating primitives. We have two sets of variables: a set of numerical variables ($\mathcal{V}$) and a set of tree variables ($\mathcal{T}$). In addition to the usual constructions provided a numerical imperative languages we add:

- An assignment statement for tree expressions (*stmt*).
- Tree expressions (*tree-expr*) which are expressions that are evaluated to trees. This adds the ability to:
  - build a tree which contains only a numerical leaf: `make_integer`(e);
  - read the i-th son of a tree t: `get_son`(t, i).
  - build a tree from a root symbol and its subtrees: `make_symbolic`$(\mathcal{F}$, *tree-expr*$, \ldots,$ *tree-expr*$)$
- finally the set of expressions of the language is also extended to allow the retrieval of information from trees (*expr*). :
  - `get_num_head`(*tree-expr*) retrieves the numerical value of the head of the tree (note that, in the concrete semantics this implies that the tree is reduced to a single leaf, as numerical values can only be found in leaves)
  - `is_symbol`(*tree-expr*) enables to test whether the head of the tree is a symbol or a numeric value.
  - simple test on symbols in trees

Definition 7.4 (Language syntax)**.** The language is defined to be:

$$
\begin{aligned}
\mathit{tree\text{-}expr} \triangleq\ & \mid \texttt{make\_symbolic}(\mathcal{F}, \mathit{tree\text{-}expr}, \ldots, \mathit{tree\text{-}expr})\\
& \mid \texttt{make\_integer}(\mathit{expr})\\
& \mid \texttt{get\_son}(\mathit{tree\text{-}expr}, \mathit{expr})\\
& \mid \mathcal{T}\\
\mathit{stmt} \triangleq\ & \mid \texttt{while (}\mathit{expr}\texttt{) \{}\mathit{stmt}\texttt{\}}\\
& \mid \texttt{if (}\mathit{expr}\texttt{) \{}\mathit{stmt}\texttt{\} else \{}\mathit{stmt}\texttt{\}}\\
& \mid \mathcal{T}\ \texttt{=}\ \mathit{tree\text{-}expr}\\
& \mid \mathcal{V}\ \texttt{=}\ \mathit{expr}\\
\mathit{expr} \triangleq\ & \mid \mathcal{W} \mid \mathbb{I} \mid \mathit{expr} \bowtie \mathit{expr} \mid \neg\mathit{expr} \qquad\qquad\qquad \bowtie \in \{<, \leqslant, =, \neq, \geqslant, >, +, -, \times, /\}\\
& \mid \texttt{get\_num\_head}(\mathit{tree\text{-}expr})\\
& \mid \texttt{is\_symbol}(\mathit{tree\text{-}expr})\\
& \mid \texttt{get\_sym\_head}(\mathit{tree\text{-}expr})\texttt{==}\mathcal{F}
\end{aligned}
$$

Remark 7.2. For presentation purposes, the mathematical definitions of the syntax and the semantics of the considered language are built upon two new set of variables $\mathcal{T}$ and $\mathcal{V}$. However in the actual implementation we extended the "universal" language shipped with the Mopsa framework, this language provides C-like types. Therefore rather than actually partitioning the set of variables, we relied on the type language. This is visible in the two example: Program 7.4 and Program 7.5. Moreover in the implementation in the universal language, symbols handling is assimilated to string handling. Note however that the set of manipulated symbols is supposed to be finite, for this reason we only handle litteral strings. Thereafter we denote as $\mathcal{R}$ the ranked alphabet obtained by gathering all litteral strings in the program, their arity is gathered statically from their use in the program.

7.2.3 Concrete semantics of the language

Having given the syntax of the considered language we now define its concrete semantics. We assume already defined an evaluation function $\mathbb{E}\llbracket . \rrbracket : (\mathcal{V} \to \mathbb{I}) \to \wp(\mathbb{I})$. This function will be extended to the newly added language features. Similarly we assume already defined an execution function $\mathbb{S}\llbracket . \rrbracket : \wp(\mathcal{V} \to \mathbb{I}) \to \wp(\mathcal{V} \to \mathbb{I})$. As mentioned in Remark 7.2, the language extension adds a new set of variables: numerical term variables. Therefore our semantic extension requires $\mathbb{E}\llbracket . \rrbracket$ and $\mathbb{S}\llbracket . \rrbracket$ to not be defined on $(\mathcal{V} \to \mathbb{I})$ but rather on $(\mathcal{V} \to \mathbb{I}) \times (\mathcal{T} \to T_{\mathbb{I}}(\mathcal{R}))$. We abuse notations and assume that $\mathbb{E}\llbracket . \rrbracket$ and $\mathbb{S}\llbracket . \rrbracket$ have been lifted to this new signature on the already existing language features.

Definition 7.5 (Concrete semantics)**.** In the following E denotes a numerical environment: $E \in \mathcal{V} \to \mathbb{I}$ and F denotes a numerical tree environment: $F \in \mathcal{T} \to T_{\mathbb{I}}(\mathcal{R})$. $T, T_1, \ldots, T_m$ denote tree

```
int i;
int n;
tree y;
assume(n >= 0);
i = 0;
y = make_symbolic("p",{});
while (i < n) {
  y = make_symbolic("*",
           {make_symbolic("+",
                 {y,
                  make_integer(4)
                 })
           });
  i = i+1;
 }
```

Program 7.4: `*(p+4)` iterated

expressions (*tree-expr*), e denotes an expression (*expr*), t denotes a tree variable ($\mathcal{T}$).

$$
\begin{aligned}
&\mathbb{E}[\![\texttt{make\_symbolic}(s \in \mathcal{F}_m, T_1, \dots, T_m)]\!](E,F) = \{s(t_1, \dots, t_m) \mid \forall i,\ t_i \in \mathbb{E}[\![T_i]\!](E,F)\}\\
&\mathbb{E}[\![\texttt{make\_integer}(e \in \textit{expr})]\!](E,F) = \mathbb{E}[\![e]\!](E,F)\\
&\mathbb{E}[\![\texttt{is\_symbol}(T)]\!](E,F) = \{\texttt{true} \mid \exists t \in \mathbb{E}[\![T]\!](E,F), \exists f \in \mathcal{R},\ t = f(\dots)\}\\
&\qquad\qquad\qquad\qquad\qquad \cup \{\texttt{false} \mid \exists t \in \mathbb{E}[\![T]\!](E,F), t \in \mathbb{I}\}\\
&\mathbb{E}[\![t]\!](E,F) = \{F(t)\}\\
&\mathbb{E}[\![\texttt{get\_son}(T,e)]\!](E,F) = \{t \mid \exists i \in \mathbb{E}[\![e]\!](E,F),\ t' \in \mathbb{E}[\![T]\!](E,F), f \in \mathcal{F}_{m>i},\\
&\qquad\qquad\qquad\qquad\qquad t' = f(t_0, \dots, t_{m-1}) \wedge t_i = t\}\\
&\mathbb{E}[\![\texttt{get\_num\_head}(T)]\!](E,F) = \{i \in \mathbb{I} \mid \exists t \in \mathbb{E}[\![T]\!](E,F),\ t = i\}\\
&\mathbb{E}[\![\texttt{get\_sym\_head}(T)]\!](E,F) = \{s \in \mathcal{R} \mid \exists t \in \mathbb{E}[\![T]\!](E,F),\ t = s(\dots)\}\\
&\mathbb{S}[\![t = T]\!](R) = \bigcup_{(E,F) \in R} \{(E, F[t \mapsto t_v]) \mid t_v \in \mathbb{E}[\![T]\!](E,F)\}
\end{aligned}
$$

Remark 7.3. Note that the definition of $\mathbb{E}[\![\texttt{is\_symbol}(T)]\!]$ yields special values `true` and `false`, which we assume representable in the underlying numerical domain $\mathbb{I}$. For example we could use the classical `true` $= 1$ and `false` $= 0$. The universal language from the Mopsa framework comes with a boolean type, which we used.

7.2.4 Example

Now that we have given the syntax and semantics of our tree language, let us show some examples of the expressiveness of this language. To do so we translated two of our introductory problems into this language. Program 7.4 computes the memory zones used by `append` from Prog 7.1, and Program 7.5 simulates the behavior of Program 7.3.

7.3 Background – (Tree) regular languages

In this section we briefly recall the definitions of regular and tree regular languages, these definitions are followed by some properties of such sets of languages. We focus on the definitions

```
int n; int i; int x; int rep;
tree t;
assume(n>=0);
i = 0;
t = make_symbolic("Nil”,{});
while (i < n) {
  t = make_symbolic("Cons”, {make_integer(x-1), t});
  t = make_symbolic("Cons”, {make_integer(x+1), t});
  i = i + 1;
 };
if (get_sym_head(t) != "Nil”) {
    rep = get_num_head(get_son(t,0));
    assert(rep > x);
 }
```

Program 7.5: List manipulation

and properties necessary to the understanding of the abstractions defined in subsequent sections. For an in-depth description of regular languages and a proof of propositions please refer to [HMU06], for tree regular languages please refer to [CDG+07].

7.3.1 Regular languages

Generalities

We provide here a definition of regular languages. Among all possible equivalent definitions we choose here to define them as the set of languages that can be recognized by a finite automaton.

Definition 7.6 (Finite Automaton FA)**.** We call *finite automaton* (FA) a quintuplet $\mathcal{A} = (\Sigma, Q, I, Q_f, \delta)$ where:

- Σ is a finite set: *the alphabet*;
- Q is a finite set: the set of all possible *states*;
- $I \subseteq Q$ is the set of all *initial states*;
- $Q_f \subseteq Q$ is the set of all *final states*;
- $\delta \subseteq Q \times \Sigma \times Q$ is the set of all *transitions*.

Definition 7.7 (Accepted word and recognized language)**.** Given $\mathcal{A} = (\Sigma, Q, I, Q_f, \delta) \in$ FA, a word $w = w_1 \dots w_n \in \Sigma^n$, a state $q \in Q$, we denote as $\delta^\star(q, w)$ the set of states $\{q_n \in Q \mid \exists (q_0, \dots, q_{n-1}) \in Q^{n-1}, q_0 = q \wedge \forall i \in \{0, \dots, n-1\}, (q_i, w_{i+1}, q_{i+1}) \in \delta\}$. A word w is said to be *accepted* by a finite automaton $\mathcal{A} = (\Sigma, Q, I, Q_f, \delta)$ if there exists $q_0 \in I$ such that $\delta^\star(w, q_0) \cap Q_f \neq \emptyset$. The set of words accepted by a FA $\mathcal{A}$ will be called the language *recognized* by $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$.

Definition 7.8 (Regular languages)**.** A language L is said to be *regular* if there exists a FA $\mathcal{A}$, such that $L = \mathcal{L}(\mathcal{A})$. The set of all regular languages over the alphabet Σ is denoted $\mathrm{Reg}(\Sigma)$.

Example 7.3.

- The language of all words over Σ is regular;
- the empty language is regular;
- every finite language is regular.

Definition 7.9 (deterministic finite automaton DFA)**.** We call *deterministic finite automaton* a finite automaton $\mathcal{A} = (\Sigma, Q, I, Q_f, \delta)$ such that:

- $|I| = 1$
- $\forall q \in Q, \forall a \in \Sigma, |\{q' \in Q \mid (q, a, q') \in \delta\}| = 1$

In this case we will abuse notations and denote $q' = \delta(q, a)$ the unique element such that $(q, a, q') \in \delta$.

Proposition 7.1 (Equivalence between FA and DFA). *For every regular language* L *there exists a* DFA $\mathcal{A}$ *such that* $L = \mathcal{L}(\mathcal{A})$. *Moreover given* $\mathcal{A}' \in$ FA *one can compute* $\mathcal{A} \in$ DFA *such that* $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

Proposition 7.2 (regular language stability). *The set of regular languages is stable under:*

- *union*
- *intersection*
- *concatenation*
- *complementation*
- *Kleene star*

Moreover all these operations are computable on the automaton representation of regular languages.

Proposition 7.3. $(Reg(\Sigma), \subseteq, \cap, \cup, \Box^c, \emptyset, \Sigma^\star)$ *is a non-complete complemented lattice with infinite height.*

As we want to use regular languages to represent reachable sets of states in a program, and as the lattice has infinite height, we need to define a widening operator ensuring sequence stabilization. We follow the idea of regular widening introduced in [Fer01], also used in e.g. [Gal08] and [Bot18].

Widening definition

Definition 7.10 (Quotient). Let $\mathcal{A} = (\Sigma, Q, I, Q_f, \delta) \in$ DFA and $\sim$ be an equivalence relation on Q. We call *quotient automaton* of $\mathcal{A}$ by $\sim$, denoted $\mathcal{A}/\sim$ the automaton: $(\Sigma, Q/\sim, \{[q]^\sim \mid q \in I\}, \{[q]^\sim \mid q \in Q_f\}, \delta')$, where: $\delta' = \{([q]^\sim, a, [\delta(q, a)]^\sim) \mid q \in Q, a \in \Sigma\}$. The number of states of $\mathcal{A}/\sim$ is the index of $\sim$.

Proposition 7.4 (Quotienting over-approximates). *For all* $\mathcal{A} \in$ DFA *and all equivalence relations* $\sim$, *we have* $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}/\sim)$.

Definition 7.11 (Congruence). Let $\mathcal{A} = (\Sigma, Q, I, Q_f, \delta) \in$ DFA and $\sim$ be an equivalence relation on Q. We call $\sim$ a *congruence* whenever:

- $\forall q_1, q_2, q_1 \sim q_2 \Rightarrow (\forall a \in \Sigma, \delta(q_1, a) \sim \delta(q_2, a))$
- $\forall q_1, q_2, q_1 \sim q_2 \Rightarrow (q_1 \in Q_f \Leftrightarrow q_2 \in Q_f)$

Proposition 7.5 (Quotienting by congruence is exact). *For all* $\mathcal{A} \in$ DFA *and all congruence* $\sim$, *we have* $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}/\sim)$.

Definition 7.12 (Nerode congruence). Given $\mathcal{A} = (\Sigma, Q, I, Q_f, \delta) \in$ DFA, we denote as: $L_q = \{w \mid \delta^\star(q, w) \cap Q_f \neq \emptyset\}$. the *Nerode congruence* is then defined as: $q_1 \sim^N q_2 \triangleq L_{q_1} = L_{q_2}$. Note that this is indeed a congruence.

Proposition 7.6 (Existence and uniqueness of a minimal equivalent DFA). *For every regular language* L *there exists a unique (up to renaming) minimal* DFA $\mathcal{A}$, *such that* $\mathcal{L}(\mathcal{A}) = L$. *Given a* DFA $\mathcal{A}'$ *recognizing* L, *one can choose* $\mathcal{A} = \mathcal{A}'/\sim^N$ *where* $\sim^N$ *is the Nerode congruence associated with* $\mathcal{A}'$.

Definition 7.13 (Myhill-Nerode sequence). Given $\mathcal{A} = (\Sigma, Q, I, Q_f, \delta) \in$ DFA, the following sequence of equivalence relation is called the *Myhill-Nerode sequence* :

- $q_1 \sim_0^N q_2 \overset{\Delta}{\Leftrightarrow} (q_1 \in Q_f \Leftrightarrow q_2 \in Q_f)$
- $q_1 \sim_{i+1}^N q_2 \overset{\Delta}{\Leftrightarrow} (q_1 \sim_i^N q_2) \wedge \forall a \in \Sigma, \delta(q_1, a) \sim_i^N \delta(q_2, a)$

Proposition 7.7 (Myhill-Nerode sequence stabilizes)**.** *Given $\mathcal{A} = (\Sigma, Q, I, Q_f, \delta) \in$ DFA, there exists an integer* k *such that the Myhill-Nerode sequence of automaton* $\mathcal{A}$ *satisfies* $\forall j \geqslant k, \sim^N_j = \sim^N_k$*. Moreover* $\sim^N = \sim^N_k$

Definition 7.14 ($[\square]_\square$)**.** Given $\mathcal{A} \in$ DFA, and an integer n, let $(\sim^N_i)_{i \geqslant 0}$ be the Myhill-Nerode sequence associated to $\mathcal{A}$, we denote as $\phi(n) = \max\{k \mid k \leqslant |\mathcal{A}| \wedge | \sim^N_k | \leqslant n\}$ (this is well-defined for $n \geqslant 2$), we then define $[\mathcal{A}]_n \triangleq \mathcal{A} / \sim^N_{\phi(n)}$.

The Myhill-Nerode sequence is a sequence of equivalence relation such that every relation refines the previous one, $\phi(n)$ enables to chose the index of the first equivalence relation in this sequence that has an index of at most n.

Remark 7.4. Thanks to Proposition 7.1 we can assimilate regular languages as the DFA that recognizes them. A regular languages is a potentially infinite set of words, whereas a DFA provides a finite representation of the language. The widening operator from the following proposition is therefore defined on DFA, rather than on regular languages.

Definition 7.15 (Widening operator)**.** We define a widening operator, parameterized by an integer w. Given $\mathcal{A} \in$ DFA and $\mathcal{A}' \in$ DFA: $\mathcal{A} \triangledown_w \mathcal{A}' \triangleq [\mathcal{A} \cup \mathcal{A}']_w$.

Remark 7.5. We abused notations by using $\cup$ to denote the operator taking two DFAs as argument and returning a DFA recognizing the union of the two languages recognized by its inputs.

Proposition 7.8 ($\triangledown_w$)**.** *For every integer* $w \geqslant 2$*,* $\triangledown_w$ *over-approximates the union and stabilizes infinite sequences.*

Regular expressions/operations

In the following, regular expressions will be also be used to represent regular languages. We recall their definition here to provide the notations used in the following. Given an alphabet Σ, the set $\mathrm{Regex}(\Sigma)$ of regular expressions over Σ is:

$$\begin{aligned} \mathrm{Regex}(\Sigma) \triangleq &\mid \epsilon \mid a && a \in \Sigma \\ &\mid \mathrm{Regex}(\Sigma) + \mathrm{Regex}(\Sigma) \\ &\mid \mathrm{Regex}(\Sigma).\mathrm{Regex}(\Sigma) \\ &\mid \mathrm{Regex}(\Sigma)^\star \end{aligned}$$

These notations will also be used for the operators on regular languages: $+$ denotes the union, . the concatenation and * the Kleene star.

Resolution of Gaussian systems (see [Sak09])

For the definition of our tree abstraction, we will need to solve systems of the form:

$$\forall i \in \{1, \ldots, n\}, X_i = \begin{cases} \bigcup_{j=1}^{n} a_{i,j}.X_j \cup \{\epsilon\} & \text{if } i \in I \\ \bigcup_{j=1}^{n} a_{i,j}.X_j & \text{otherwise} \end{cases} \tag{7.1}$$

with unknown $(X_i) \in \wp(\Sigma^\star)$, where $I \subseteq \{1, \ldots, n\}$, and where $\forall i, j, a_{i,j} \in \Sigma$. Moreover in addition to the computation of the unique solution, we use the fact that this solution maps every X_i to a regular language. This result is achieved by performing several Gaussian eliminations, this requires expressing X_i in terms of the X_j for $j \neq i$. This can be achieved by Arden's rule.

Proposition 7.9 (Arden's rule (case $\epsilon \notin K$))**.** *Let* K *and* L *be two languages over* Σ*, such that* $\epsilon \notin K$*, then the equation* $X = KX \cup L$ *admits a unique solution that is* $X = K^\star L$.

We can rewrite an equation $X_i = \bigcup_{j=1}^{n} a_{i,j}X_j$ to the form $X_i = \bigcup_{j\neq i} a_{i,j}X_j \cup a_{i,i}X_i$. Using Proposition 7.9 we conclude that $X_i = a_{i,i}^{\star}(\bigcup_{j\neq i} a_{i,j}X_j)$. Having shown that we are able to eliminate X_i from the right hand side of the equation defining X_i we can iterate this process in a Gaussian elimination fashion. This yields several results:

- Eq. 7.1 has a unique solution;
- The unique solution maps the X_i to regular languages;
- The unique solution can be computed.

7.3.2 Tree regular languages

We now recall the definition of tree regular languages as well as some of the properties used for the abstraction of subsequent sections. Note that the properties and definitions of this subsection will be very close to those of the Section 7.3.1. Regular languages are actually a special case of tree regular languages when every symbol from the ranked alphabet has arity 1.

Generalities

Definition 7.16 (Finite Tree Automaton FTA)**.** A *finite tree automaton* (FTA) is a tuple $(\mathcal{R}, Q, Q_f, \delta)$, where:

- $\mathcal{R}$ is a ranked alphabet
- Q is a finite set of states;
- $Q_f \subseteq Q$ is the set of final states;
- $\delta \in \wp_f(\bigcup_{n\in\mathbb{N}} \mathcal{F}_n \times Q^n \times Q)$ is the set of transitions.

We define $\overline{\delta} : (\bigcup_{n\in\mathbb{N}} \mathcal{F}_n \times Q^n) \to \wp(Q)$ by: $\overline{\delta}(f, \overrightarrow{q}) = \{q' \mid (f, \overrightarrow{q}, q') \in \delta\}$.

Definition 7.17 (Reachability function)**.** Given a FTA $\mathcal{A} = (\mathcal{R}, Q, Q_f, \delta)$ we define a reachability function $\delta^\star : T(\mathcal{R}) \to \wp(Q)$:

$$\delta^\star(t) = \textbf{let } t_1, \ldots, t_n = \textbf{sons}(t) \textbf{ in} \bigcup_{(q_1,\ldots,q_n)\in(\delta^\star(t_1),\ldots,\delta^\star(t_n))} \{q \mid (\textbf{head}(t), (q_1, \ldots, q_n), q) \in \delta\}$$

If **sons**(t) is the empty tuple (which is the case when t is a constant a), the union is made over a unique element (which is the empty tuple), which then boils down to: $\{q \mid (a, (), q) \in \delta\}$. If **sons**$(t)$ is not the empty tuple and for some i, $\delta^\star(t_i)$ is empty, then $\delta^\star(t)$ is also empty

Example 7.4. Consider the ranked alphabet $\mathcal{R} = \{f(2), a(0)\}$, and the automaton $\mathcal{A} = (\mathcal{R}, \{u, v\}, \{v\}, \{a() \to u, f(v, v) \to v, f(u, u) \to u, f(u, u) \to v\})$. Then $\delta^\star(a) = \{u\}$, $\delta^\star(f(a, a)) = \{u, v\}$, $\delta^\star(f(f(a, a), a)) = \{u, v\}$.

Definition 7.18 (Accepted tree, recognized tree language)**.** Given a FTA $\mathcal{A} = (\mathcal{R}, Q, Q_f, \delta)$, a term t, we say that t is *accepted* by the automaton if $\delta^\star(t) \cap Q_f \neq \emptyset$. The set of trees accepted by a FTA $\mathcal{A}$ will be called the *recognized tree language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$.

Example 7.5. With the definition of Example 7.4, $\mathcal{L}(\mathcal{A})$ is the set of terms over $\mathcal{R}$ that contain at least one f.

Definition 7.19 (Tree regular languages)**.** A set of terms $\mathcal{T}$ over a ranked alphabet $\mathcal{R}$ is called *tree regular* if there exists a FTA $\mathcal{A}$ over $\mathcal{R}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{T}$. The set of tree regular languages over a ranked alphabet $\mathcal{R}$ is denoted $\text{TReg}(\mathcal{R})$.

Example 7.6. • The language of all trees over $\mathcal{R}$ is tree regular;

- the empty language is tree regular;
- every finite tree language is tree regular.

Definition 7.20 (Deterministic Finite Tree Automaton Dfta). We call *deterministic finite tree automaton* a finite tree automaton $\mathcal{A} = (\mathcal{R}, Q, Q_f, \delta)$: such that: $\forall n \in \mathbb{N}$, $f \in \mathcal{F}_n$, $\overrightarrow{q} \in Q^n$, $|\{q' \mid (f, \overrightarrow{q}, q') \in \delta\}| = 1$. We then abuse notations and denote as $\delta(f, \overrightarrow{q})$ the unique element in the set $\{q' \mid (f, \overrightarrow{q}, q') \in \delta\}$.

Proposition 7.10 (Equivalence between Fta and Dfta). *As for regular languages, for every tree regular language* L *there exists* $\mathcal{A} \in$ *Dfta such that* $\mathcal{L}(\mathcal{A}) = L$. *Moreover given* $\mathcal{A}' \in$ *Fta one can compute* $\mathcal{A} \in$ *Dfta such that* $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

Proposition 7.11. *The set of tree regular languages is stable under:*

- *union*
- *intersection*
- *complementation*

Moreover all these operations are computable on the automaton representation of tree regular languages.

Proposition 7.12. $(\mathit{TReg}(\mathcal{R}), \subseteq, \cap, \cup, \square^c, \emptyset, T(\mathcal{R}))$ *is a non-complete, complemented lattice with infinite height.*

Widening operator

As for regular languages, the fact that the lattice has infinite height creates the need for a widening operator. The closeness between regular languages and tree regular languages enable us to follow the classical widening definition (presented in Section 7.3.1) and adapt it to tree regular languages.

Definition 7.21 (Quotient). Let $\mathcal{A} = (\mathcal{R}, Q, Q_f, \delta) \in$ Dfta and $\sim$ be an equivalence relation on Q. We call *quotient automaton* of $\mathcal{A}$ by $\sim$, denoted $\mathcal{A}/\sim$ the automaton: $(\mathcal{R}, Q/\sim, \{[q] \mid q \in Q_f\}, \delta')$, where: $\delta' = \{(f, ([q_1], \ldots, [q_n]), [\delta(f, (q_1, \ldots, q_n))]) \mid n \in \mathbb{N}, q \in Q, f \in \mathcal{F}_n\}$. The number of state of $\mathcal{A}/\sim$ is the index of $\sim$.

Proposition 7.13 (Quotienting over-approximates). *For all* $\mathcal{A} \in$ *Dfta and all equivalence relation* $\sim$, *we have* $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}/\sim)$.

Definition 7.22 (Congruence). Let $\mathcal{A} = (\Sigma, Q, I, Q_f, \delta) \in$ Dfa and $\sim$ be an equivalence relation on Q. We call $\sim$ a *congruence* whenever:

- $\forall p_1, \ldots, p_n \in Q,\ \forall q_1, \ldots, q_n \in Q, (\bigwedge_{i=1}^{n} p_i \sim q_i) \Rightarrow \forall f \in \mathcal{F}_n,\ \delta(f, p_1, \ldots, p_n) \sim \delta(f, q_1, \ldots, q_n)$
- $\forall p, q, p \sim q \Rightarrow (p \in Q_f \Leftrightarrow q \in Q_f)$

Proposition 7.14 (Quotienting by congruence is exact). *For all* $\mathcal{A} \in$ *Dfta and all congruence* $\sim$, *we have* $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}/\sim)$.

As for regular languages, we have the existence of a minimal equivalent Dfta.

Proposition 7.15 (Existence and uniqueness of a minimal equivalent Dfta). *For every tree regular language* L, *there exists a unique (up to renaming) minimal Dfta* $\mathcal{A}$, *such that* $\mathcal{L}(\mathcal{A}) = L$.

Definition 7.23 (Myhill-Nerode sequence). Given $\mathcal{A} = (\mathcal{R}, Q, Q_f, \delta) \in$ Dfta, the following sequence of equivalence relation is called the *Myhill-Nerode sequence*: :

- $p \sim_0^N p' \triangleq (p \in Q_f \Leftrightarrow p' \in Q_f)$
- $p \sim_{i+1}^N p' \triangleq (p \sim_i^N p') \wedge \forall n \in \mathbb{N}, \forall i \in \{1, \ldots, n\}, \forall f \in \mathcal{F}_n, \forall (q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_n) \in Q^{n-1}, \delta(f, (q_1, \ldots, q_{i-1}, p, q_{i+1}, \ldots, q_n)) \sim_i^N \delta(f, (q_1, \ldots, q_{i-1}, p', q_{i+1}, \ldots, q_n))$

Proposition 7.16 (Myhill-Nerode sequence stabilizes). *Given* $\mathcal{A} = (\mathcal{R}, Q, Q_f, \delta) \in$ *Dfta, there exists an integer* k *such that the Myhill-Nerode sequence of automaton* $\mathcal{A}$ *satisfies* $\forall j \geqslant k, \sim_j^N = \sim_k^N$.

Definition 7.24 ($[\square]_\square$)**.** Given $\mathcal{A} \in$ Dfta, and an integer n, let $(\sim^N_i)_{i \geqslant 0}$ be the Myhill-Nerode sequence associated to $\mathcal{A}$, we denote as $\phi(n) = \max\{k \mid k \leqslant |\mathcal{A}| \wedge |\sim^N_k| \leqslant n\}$ (this is well-defined for $n \geqslant 2$), we then define $[\mathcal{A}]_n \triangleq \mathcal{A}/\sim^N_{\phi(n)}$.

Remark 7.6. Thanks to Proposition 7.10 we can assimilate tree regular languages as the Dfta that recognizes them. The widening operator from the following proposition is therefore defined on Dfta, rather than tree regular languages.

Definition 7.25 (Widening operator)**.** We define a widening operator, parameterized by an integer w. Given $\mathcal{A} \in$ Dfa and $\mathcal{A}' \in$ Dfa: $\mathcal{A} \triangledown_w \mathcal{A}' \triangleq [\mathcal{A} \cup \mathcal{A}']_w$.

Proposition 7.17 ($\triangledown_w$)**.** *For every integer $w \geqslant 2$, $\triangledown_w$ over-approximates the union and stabilizes infinite sequences.*

See proof on page 157.

Example 7.7. Consider the ranked algebra $\mathcal{F} = \{f(2), a(0)\}$, the following complete and deterministic automaton: $Q = \{0, 1, 2, 3, 4\}$, $Q_f = \{1, 2, 3, 4\}$, $\delta = \{a() \to 1, f(1, 1) \to 2, f(1, 2) \to 3, f(1, 3) \to 4\}$, where every unmentioned transition mentioned goes towards 0. The recognized language is a, $f(a, a)$, $f(a, f(a, a))$ and $f(a, f(a, f(a, a)))$. In order to relax constraints on this automaton, we compute the following equivalence of index 4: $\{3, 4\}, \{0\}, \{1\}, \{2\}$ and merge equivalent states (this introduces a loop). We therefore obtain the automaton: $Q = \{0, 1, 2, 3\}$, $Q_f = \{1, 2, 3\}$, $\delta = \{a() \to 1, f(1, 1) \to 2, f(1, 2) \to 3, f(1, 3) \to 3\}$, which recognizes the language: $f(a, f(\dots, f(a, a) \dots))$. Table 7.1 defines four tree automata $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$. On these elements, $\mathcal{B}_0, \mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$ are defined by: $\mathcal{B}_0 = \mathcal{A}_0$ and $\mathcal{B}_i = \mathcal{B}_{i-1} \triangledown \mathcal{A}_i$ for the constant $w = 4$. ● labels on nodes of Tab 7.1 mean that the sub-tree rooted in that node is also accepted. We can see that $\mathcal{B}_1$ (resp. $\mathcal{B}_2$) represents exactly $\mathcal{A}_0 \cup \mathcal{A}_1$ (resp. $\mathcal{A}_0 \cup \mathcal{A}_1 \cup \mathcal{A}_2$) whereas the language accepted by $\mathcal{B}_3$ is an over approximation of $\mathcal{A}_0 \cup \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3 (\triangleq \mathcal{C})$. Indeed representing effectively $\mathcal{C}$ would require 5 states. Therefore an equivalence of index 4 is computed on the states of the automaton representing the exact union, once this equivalence is computed, equivalent states are merged, thus loosing precision.

7.3.3 Implementation remarks

As both the regular language lattice and the tree regular language lattice were needed for the development of the numerical term abstraction (presented in subsequent sections), all above-mentioned operations have been implemented. We chose to implement our own regular and tree regular lattice operations instead of using existing libraries as we needed to:

- Test several design choices;
- Relax some operations. Indeed some of the above mentioned operators can be relaxed to compute an over-approximation (for the language inclusion ordering) of their expected result. This comes from the fact that these are used in an abstract interpreter. Note however that this induces a precision loss;
- Define a widening operator (operator not provided in classical (tree) regular language implementations). It is not defined as a composition of low level operators, but rather as a modified minimization algorithm.

In this subsection we describe some features of our implementation.

Generalities. The regular (resp. tree regular) lattice and its operations have been implemented as a generic abstract domain in the Mopsa framework (see Chapter 3). This entails the classical abstract lattice operations: join, meet, inclusion test, emptyness checking, widening operator. Both implementations provide additional operators, which are domain dependant (e.g. concatenation, gaussian solver, ...).

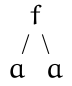
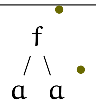
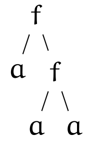
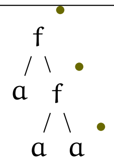
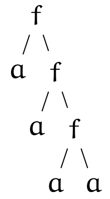
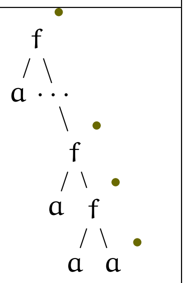
name	Q	Q_f	δ	$\mathcal{L}$
$\mathcal{A}_0$	$\{0,1\}$	$\{1\}$	$a() \to 1$	a
$\mathcal{A}_1$	$\{0,1,2\}$	$\{2\}$	$a() \to 1$ $f(1,1) \to 2$	
$\mathcal{B}_1$	$\{0,1,2\}$	$\{1,2\}$	$a() \to 1$ $f(1,1) \to 2$	
$\mathcal{A}_2$	$\{0,1,2,3\}$	$\{3\}$	$a() \to 1$ $f(1,1) \to 2$ $f(1,2) \to 3$	
$\mathcal{B}_2$	$\{0,1,2,3\}$	$\{1,2,3\}$	$a() \to 1$ $f(1,1) \to 2$ $f(1,2) \to 3$	
$\mathcal{A}_3$	$\{0,1,2,3,4\}$	$\{4\}$	$a() \to 1$ $f(1,1) \to 2$ $f(1,2) \to 3$ $f(1,3) \to 4$	
$\mathcal{B}_3$	$\{0,1,2,3\}$	$\{1,2,3\}$	$a() \to 1$ $f(1,1) \to 2$ $f(1,2) \to 3$ $f(1,3) \to 3$	

Table 7.1: Illustration of widening operator on tree regular languages

Automaton representation. Following the definitions, we represent (tree) regular languages as their automaton. Our implementation contains: a deterministic automaton signature as well as a non deterministic one. We try to keep them deterministic as long as possible. The union and intersection are therefore computed via product automaton (which produces a deterministic output as soon as its inputs are deterministic).

7.4 State of the art

7.4.1 Using (Tree) Regular languages

In [BHRV06]and [BHV04] tree automata and regular automata are used for the model checking of programs manipulating C pointers and structures. Other uses have been made of tree automata in verification: shape analysis of C programs as in [DPV11], computation of an over-approximation of terms computable by attackers of cryptographic protocols as in [Mon99]. As mentioned in Section 7.3, widening regular languages by the computation of an equivalence relation of bounded index is also done in [Fer01] and in [GJJ06].

7.4.2 Tree languages

Previous works on sets of trees abstractions [Mau99] were able to recognize larger classes of tree languages than tree automata, however they do not consider sets of trees with numerically labeled leaves. Termination analyses have been proposed for the analysis of programs manipulating tree structures (AVL, red-black trees) see [HIRV07].

7.4.3 Numerical terms

We focus in this chapter on the abstraction of trees labeled with numerical values, therefore the work closest to ours would be [GGLM12]. For this reason we give here a summarized account of their results, and point out the differences between their work and ours. Their goal is to represent trees, the leaves of which can be labeled with elements from a set $\mathcal{D}$. They assume given an abstract lattice $\mathcal{D}^\sharp$, forming a representation $(\wp(\mathcal{D}), \subseteq) \xleftarrow{\gamma} (\mathcal{D}^\sharp, \sqsubseteq)$. Their starting point is the representation of sets of numerical terms by tree automata. They extend tree automata in the following manner: each state of the tree automaton is mapped (via a function we denote ϕ) to an abstract element from an abstract lattice. Running a tree labeled with elements from $\mathcal{D}$ in such a tree automaton is a straightforward extension of the classical definition (provided in Section 7.3):

- if the head of the tree is labeled with a symbol we follow the usual definition
- otherwise if the head is an element $x \in \mathcal{D}$ then running the tree yields the set of states: $\{q \mid x \in \gamma(\phi(q))\}$. These are the states, the adjoined abstract element of which represent a set containing x.

These extended tree automata are concretized towards the set of trees they accept. We see that their abstraction can represent more general sets of trees than the ones considered in this chapter. Indeed as states are labeled with an element from an abstract lattice, we could choose, for example, the interval lattice presented in Chapter 2, thus yielding numerically labeled trees. However, choosing the constant string lattice, we obtain a representation for trees let leaves of which are labeled with strings. However this generality comes at the cost of precision: no relations can be expressed between the numerical values stored in the leaves of the represented trees, nor could we express relations between the leaves of the represented trees and the other program variables. Another drawback is the fact that the precise inclusion test would require the abstract lattice to be equipped with a complementation operator, which is not the case for the interval lattice, therefore the authors had to result to the classical technique of extending the lattice with a partitioning (for example the interval lattice is extended with the partitioning $]-\infty;0[, [0;0],]0;\infty[$), this technique loses precision (on top of the precision loss induced by the use of an abstract lattice).

Example 7.8.
- Consider the following set of numerically labeled trees: $S = \{\beta, f(\alpha, \beta), f(\alpha, f(\alpha, \beta)), \cdots \mid \beta \geqslant 0, \alpha \leqslant -3\}$. Consider moreover the tree automaton $\mathcal{A} = (\{f(2)\}, \{a, b\}, \{b\}, \{f(a, b) \to b\})$ augmented with the following ϕ function: $\phi(a) =]-\infty; -3]$ and

$\phi(b) = [0; \infty]$. Using results from [GGLM12], $(\mathcal{A}, \phi)$ represents exactly the tree set S. The abstraction defined in this chapter is also able to precisely represent S.

- Consider now the following set of numerically labeled trees: $S = \{\beta, f(\alpha, \beta), f(\alpha, f(\alpha, \beta)), \cdots \mid \beta = \alpha + 1\}$. The abstraction from [GGLM12] can not precisely represent S due to the numerical relationality between the leaves of the trees. The abstraction presented here is able to precisely represent S.

Chapter intermission. In previous subsections, we have shown that we needed abstractions representing environments mapping variables to numerical trees. In the two following sections (Section 7.5 and Section 7.6) we define two abstractions representing such numerical tree environments. Both will be defined as a lifting of a value abstraction for numerical tree sets to an environment. These liftings can be compared to the lifting of the interval value domain (representing $\wp(\mathbb{R})$) to the interval domain (representing $\wp(\mathcal{V} \to \mathbb{R})$). Note however that this lifting induces a loss of relationality for the resulting environments. For this reason Section 7.7 will explain how the lifting is modified to regain relational environments.

7.5 Natural term abstraction by tree automata

In this subsection we start by defining a value abstraction for tree sets (in Section 7.5.1), which is then lifted to an environment abstraction (in Section 7.5.3). This abstraction will tackle the problem of representing sets of tree shapes. This will be done using tree automata (introduced in Section 7.3.2). As we only focus on tree shapes, numerical values contained in leaves will be abstracted away and replaced by the $\square$ symbol.

7.5.1 Value abstraction

Definition 7.26 ($\mathcal{R}^\square$)**.** Given $\mathcal{R}$, We denote as $\mathcal{R}^\square$ the ranked alphabet $\mathcal{R}$ after adding the symbol $\square$ of arity 0 (we assume that $\square \notin \mathcal{R}$).

Definition 7.27 (Holify function)**.** Given a numerical term $t \in T_{\mathbb{I}}(\mathcal{R})$ we define inductively:

$$t^\square = \begin{cases} f(t_1^\square, \dots, t_n^\square) & \text{when } t = f(t_1, \dots, t_n) \\ \square & \text{when } t = n \in \mathbb{I} \end{cases}$$

$t^\square$ is the term obtained by replacing every number with the $\square$ symbol in t.

Proposition 7.18 (Tree automata abstraction)**.** $(\wp(T_{\mathbb{I}}(\mathcal{R})), \subseteq) \xleftarrow{\gamma}$ *(TReg*$(\mathcal{R}^\square), \subseteq)$*, where:*

$$\gamma(\mathcal{A}) = \{t \mid t^\square \in \mathcal{L}(\mathcal{A})\}$$

is a representation. Moreover with such a γ *definition,* $\cup$*,* $\cap$ *soundly abstracts the union and the intersection.*

Remark 7.7. Due to the use of the tree regular abstraction, we only have a representation and not a Galois connection. Indeed, consider now ranked alphabet $\{a(1), b(1), \epsilon(0)\}$ and the set of terms $\mathcal{T} = \{\epsilon, a(b(\epsilon)), a(a(b(b(\epsilon)))), \dots\}$. We can prove (in a similar way as for $a^n b^n$ in regular languages) that $\mathcal{T}$ is not tree regular, moreover $\mathcal{T}$ does not have a best tree regular over approximation.

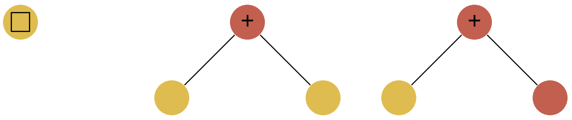


Figure 7.3: Automaton from Example 7.9

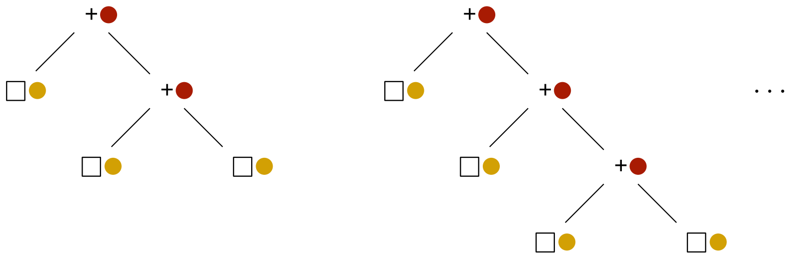


Figure 7.4: Examples of accepted trees from Example 7.9

Example 7.9. Let $\mathcal{R} = \{+(2)\}$ and $\mathcal{A} = (\{🔴, 🟡\}, \mathcal{R}^{\Box}, \{🔴, 🟡\}, \{(\Box() \to 🟡, +(🟡, 🟡) \to 🔴, +(🟡, 🔴) \to 🔴)\})$. This automaton is represented in Figure 7.3. Examples of terms recognized by $\mathcal{A}$ are shown on Figure 7.4. Natural terms U and V, from our running example (defined in Example 7.2), are also contained in $\gamma(\mathcal{A})$. Moreover as we do not provide numerical constraints: $1 + (3 + 4)$, 23, $1 + (2 + (3 + 4))$ are also elements in $\gamma(\mathcal{A})$.

Remark 7.8. Consider the two following complete and deterministic tree automata: $\mathcal{A} = (\{a, b, h\}, \{+(2)\}, \{a\}, \{\Box() \to b, +(b, b) \to a\})$ and $\mathcal{B} = (\{a, b, c, h\}, \{+(2)\}, \{a\}, \{\Box() \to b, +(b, b) \to c, +(b, c) \to a\})$ (unmentioned transitions go to h). These are similar tree automata as the one showed in Table 7.1. $\mathcal{A}$ recognizes the tree $+(\Box, \Box)$ and $\mathcal{B}$ recognizes the tree $+(\Box, +(\Box, \Box))$, they over-approximate respectively U and V from our running example. $\mathcal{A} \cup \mathcal{B}$ is recognized by the following complete and deterministic tree automaton: $\mathcal{C} = (\{a, b, c, h\}, \{+(2)\}, \{a, c\}, \{\Box() \to b, +(b, b) \to c, +(b, c) \to a\})$. If we want to widen $\mathcal{A}$ and $\mathcal{B}$ with parameter 3, the following equivalence relation is computed: $\{\{h\}, \{b\}, \{a, c\}\}$. Merging equivalent states in $\mathcal{C}$ produces $(\{a, b, h\}, \{+(2)\}, \{a\}, \{\Box() \to b, +(b, b) \to a, +(b, a) \to a\})$, which contains a loop and over-approximates the union. For illustrations please refer to Table 7.1.

7.5.2 Abstract transformers

We provide here the definition of Algorithms 7.1 to 7.6, that respectively provide operators **make_symbolic**, **make_integer**, **is_symbol**, **get_sym_head**, **get_num_head**, **get_son**. These algorithms operate on tree automata and will be used in the definition of the semantics of our abstract environment. Note moreover that these operators rely on basic tree automaton manipulations.

- Algorithm 7.1 takes several tree automata as inputs, concatenate all their transitions and add new transitions from every tuple of final states to some new final state.
- Algorithm 7.2 builds a tree automaton recognizing the language $\{\Box\}$. **fresh_state** yields a fresh state name.
- Algorithm 7.3 checks whether there is some transition labeled with a non $\Box$ symbol going to one of the final state of the automaton (all states are assumed reachable in the input representation).
- Algorithm 7.4 collects all transitions labels for transitions labeled with a non $\Box$ symbol and going to a final state.
- Algorithm 7.5 yields $\top$ if the $\Box$ symbol yields a final state and $\bot$ otherwise. Please note that this prevents us from recovering any numerical value stored in the numerical trees represented.
- Algorithm 7.6 takes as inputs: one tree automaton and an integer i, collects all possible i-th sons of final states and sets these as the new final states.

Algorithm 7.1: make_symbolic

```
Input  : $s \in \mathcal{F}_n, t_1, \dots, t_n \in$ DFTA, where the $t_i$ share the same ranked algebra $\mathcal{R}$
Output: a FTA
$f \leftarrow$ a fresh state;
 $(Q^i, \mathcal{R}^i, Q^i_f, \delta^i)_{i \leqslant n} \leftarrow$ a renaming of $t_1, \dots, t_n$ so that they share no state;
$Q \leftarrow \bigcup_{i \leqslant n} Q^i$;
$\delta \leftarrow \bigcup_{(f_1,\dots,f_n) \in Q^1_f \times \dots \times Q^n_f} \{s(f_1, \dots, f_n) \to f\} \cup \bigcup_{i \leqslant n} \delta^i$;
return $(Q, \mathcal{R}, \{f\}, \delta)$;
```

Algorithm 7.2: make_integer

```
Input  : None
Output: a FTA
$a \leftarrow$ fresh_state();
return $\langle \{a\}, \mathcal{R}, \{a\}, \{\Box() \to a\} \rangle$;
```

Algorithm 7.3: is_symbol

```
Input  : $(Q, \mathcal{R}, Q_f, \delta) \in$ DFTA, where all states are reachable
Output: true, false or $\top$
$r \leftarrow \emptyset$;
foreach $f \in Q_f$ do
    if $\exists s(\dots) \to f \in \delta \wedge s \neq \Box$ then
        if $\Box \to f \in \delta$ then
            return $\top$;
        else
            return true;
    else
        return false;
```

Algorithm 7.4: get_sym_head

```
Input  : $(Q, \mathcal{R}, Q_f, \delta) \in$ DFTA, where all states are reachable
Output: a set of symbols
return $\bigcup_{f \in Q_f} \bigcup_{s(\dots) \to f \in \delta \wedge s \neq \Box} \{s\}$;
```

Algorithm 7.5: get_num_head

```
Input  : $(Q, \mathcal{R}, Q_f, \delta) \in$ DFTA
Output: a numerical abstract value
if $\exists f \in Q_f, \Box \to f \in \delta$ then
    return $\top$
else
    return $\bot$
```

Algorithm 7.6: get_son

```
Input  : $(Q, \mathcal{R}, Q_f, \delta) \in$ DFTA, $i \in \mathbb{N}$
Output: a FTA
$r \leftarrow \emptyset$;
foreach $f \in Q_f$ do
    foreach $s(p_0, \dots, p_{m-1}) \to f \in \delta$ do
        $r \leftarrow \{p_i\} \cup r$;
return $(Q, \mathcal{R}, r, \delta)$;
```

7.5.3 Environment abstraction

Now that we are given an abstraction for numerical term sets, let us show how this is lifted to a notion of abstract numerical term environments mapping variables to numerical terms. Given a set of numerical term variables $\mathcal{T}$, consider $\mathfrak{F}^\sharp \triangleq (\mathcal{T} \to \mathrm{TReg}(\mathcal{R}^\square)) \cup \{\bot\}$ and the set operators defined by the point-wise lifting of operators on $\mathrm{TReg}(\mathcal{R}^\square)$. We also lift the concretization function $\wp(T_{\mathbb{I}}(\mathcal{R})) \leftarrow \mathrm{TReg}(\mathcal{R}^\square)$. We assume given an abstract numerical environment $E^\sharp$ and an abstract evaluator $\mathbb{E}^\sharp[\![e]\!]$. As for the concrete semantics, we lift the existing abstract semantics to account for tree environments: $\mathbb{E}^\sharp[\![\square]\!](E^\sharp, F^\sharp)$ and extend it to tree expressions and tree statements in the following definition.

Definition 7.28 (Abstract semantics)**.** In the following, $s \in \mathcal{R}$, $T, T_1, \dots, T_m \in$ *tree-expr*, $e \in$ *expr*, $t \in \mathcal{T}$:

$$\mathbb{E}^\sharp[\![\texttt{make\_symbolic}(s, T_1, \dots, T_m)]\!](E^\sharp, F^\sharp) = \textbf{make\_symbolic}(s, \mathbb{E}^\sharp[\![T_1]\!](E^\sharp, F^\sharp), \dots, \mathbb{E}^\sharp[\![T_m]\!](E^\sharp, F^\sharp))$$

$$\mathbb{E}^\sharp[\![\texttt{make\_integer}(e)]\!](E^\sharp, F^\sharp) = \begin{array}{l} \textbf{let } \_ = \mathbb{E}^\sharp[\![e]\!](E^\sharp, F^\sharp) \textbf{ in} \\ \textbf{make\_integer}() \end{array}$$

$$\mathbb{E}^\sharp[\![\texttt{is\_symbol}(T)]\!](E^\sharp, F^\sharp) = \textbf{is\_symbol}(\mathbb{E}^\sharp[\![T]\!](E^\sharp, F^\sharp))$$

$$\mathbb{E}^\sharp[\![t]\!](E^\sharp, F^\sharp) = \{F^\sharp(t)\}$$

$$\mathbb{E}^\sharp[\![\texttt{get\_son}(T, e)]\!](E^\sharp, F^\sharp) = \bigcup_{i \in \mathbb{E}^\sharp[\![e]\!](E^\sharp, F^\sharp) \cap \{0, \dots, m-1\}} \textbf{get\_son}(\mathbb{E}^\sharp[\![T]\!](E^\sharp, F^\sharp), i)$$

$$\mathbb{E}^\sharp[\![\texttt{get\_num\_head}(T)]\!](E^\sharp, F^\sharp) = \textbf{get\_num\_head}(\mathbb{E}^\sharp[\![T]\!](E^\sharp, F^\sharp))$$

$$\mathbb{E}^\sharp[\![\texttt{get\_sym\_head}(T)]\!](E^\sharp, F^\sharp) = \textbf{get\_sym\_head}(\mathbb{E}^\sharp[\![T]\!](E^\sharp, F^\sharp))$$

$$\mathbb{S}^\sharp[\![\texttt{t=T}]\!](E^\sharp, F^\sharp) = \{(E^\sharp, F^\sharp[t \mapsto \mathbb{E}^\sharp[\![T]\!](E^\sharp, F^\sharp)])\}$$

Remark 7.9. Note that most of these definitions are just a trivial lifting of the underlying transformers, operating on tree automata.

Remark 7.10. The definition of $\mathbb{E}^\sharp[\![\texttt{get\_son}(T, e)]\!](E^\sharp, F^\sharp)$ iterates over $i \in \mathbb{E}^\sharp[\![e]\!](E^\sharp, F^\sharp) \cap \{0, \dots, m-1\}$. This can be handled by the case-by-case evaluation provided by the MOPSA framework, presented in Chapter 3, and already used in the definition of the string domain from Chapter 4.

7.6 Natural term abstraction by numerical constraints

We have seen in Chapter 6 how to represent sets of maps with heterogeneous supports and how to lift their concretization (modulo a summarization function) to sets of maps with infinite and heterogeneous supports. Given a tree shape (in the sense of Section 7.5), we can associate a numeric variable to each numeric leaf, and use a numeric abstract element to represent the possible values of these leaves. We will name the variables of each leaf as the path from the root to the

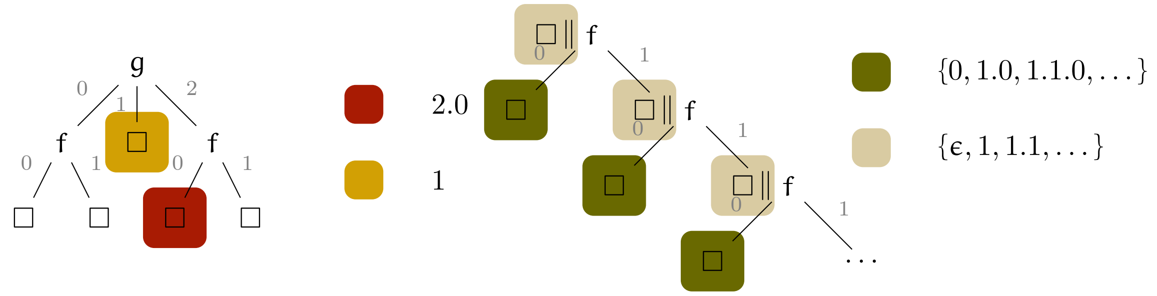


Figure 7.5: Leaves labelling by words and summarization as word sets

leaf, i.e., $\mathcal{V}$ is a set of words over $\{0, ..., n-1\}$ where n is the maximum arity of the considered ranked alphabet (see Figure 7.5). A summarized variable then represents a set of such paths. We will abstract such sets as regular expressions. Using the summarization extended to heterogeneous supports presented in Chapter 6, it will be possible to represent, using a single numeric abstract element, a set of contraints over the numerical leaves of an infinite set of unbounded trees of arbitrary shape.

7.6.1 Value abstraction

The presentation of our computable abstraction able to represent numerical values in trees is broken down (for presentation purposes only) into two consecutive abstractions. The first one is not computable (as there is a potentially infinite number of numerical variables), as numerical terms are abstracted as partial environments over tree paths to numerical values. This abstraction looses most of the tree shapes but focuses on their numerical environment. A second abstraction will show how partial environments over paths are abstracted into numerical abstract elements defined over a regular expression environment. There will be a finite number of regular expression (yielding a computable abstraction), each regular expression will be used to convey constraints over a potentially infinite number of concrete tree leaves.

We start by providing in the following definitions some of the vocabulary used in the reminder of this chapter.

Definition 7.29 (Words)**.** In the following, when $\mathcal{R}$ is a ranked alphabet of maximum arity n, we call *words* sequences of integers, $w = (w_0, \dots, w_{p-1}) \in \{0, \dots, (n-1)\}^p$ will be called a word of *length* p (denoted $|w|$), w_i denotes the i-th integer of the sequence, $\overline{w} = (w_1, \dots, w_{p-1})$ is the tail of word w, $\mathcal{W}(\mathcal{R}) = \{0, \dots, (n-1)\}^\star$ is the set of all words over $\{0, \dots, n-1\}$ of arbitrary size. Words will be used to denote paths in trees, in order to avoid confusion such paths will be denoted $\wr 0, 1, 1 \wr$ for the word $(0, 1, 1)$.

We now define the subterm of a term t at position w where w is a word. This is used in the definition of the abstraction.

Definition 7.30 (Position in a term)**.** Given a numerical term t and a word w we inductively define the subterm of t at position w (denoted $t_{|w}$) to be:

$$t_{|w} = \begin{cases} (t_{w_0})_{|\overline{w}} & \text{when } |w| > 0 \wedge t = f(t_0, \dots, t_{p-1}) \text{ with } w_0 < p \\ t & \text{when } |w| = 0 \\ \textbf{undefined} & \text{otherwise} \end{cases}$$

Moreover we denote as $\textbf{numeric}(t) = \{w \in \mathcal{W}(\mathcal{R}) \mid t_{|w} \in \mathbb{Z}\}$.

Definition 7.31 (Positioning with exact numerical constraints)**.** We define $\mathcal{C}(\mathcal{R}) \triangleq \wp(\mathcal{W}(\mathcal{R}) \nrightarrow \mathbb{I})$, an element of $\mathcal{C}(\mathcal{R})$ is therefore a set of partial maps that are acceptable bindings of positions to numeric values.

Definition 7.32 (Numerical map associated to a term)**.** When t is a numerical term, we denote as $t_{\mathbb{I}} \in \textbf{numeric}(t) \to \mathbb{I}$ the function such that $\forall w \in \textbf{numeric}(t) g, t_{\mathbb{I}}(w) = t_{|w}$

$t_{\mathbb{I}}$ is the numerical map associated with the numerical term t, it maps its positions containing numerical values to the according numerical value. Our first abstraction, provided in the following proposition, abstracts terms by their numerical map.

Proposition 7.19 (Galois connection with numerical terms)**.** *We have the following Galois connection:* $(\wp(T_{\mathbb{I}}(\mathcal{R})), \subseteq) \underset{\alpha_{\mathcal{C}(\mathcal{R})}}{\overset{\gamma_{\mathcal{C}(\mathcal{R})}}{\leftrightarrows}} (\mathcal{C}(\mathcal{R}), \subseteq)$*, with:*

$$\begin{aligned}\gamma_{\mathcal{C}(\mathcal{R})}(\Gamma) &= \{t \in T_{\mathbb{I}}(\mathcal{R}) \mid t_{\mathbb{I}} \in \Gamma\}\\ \alpha_{\mathcal{C}(\mathcal{R})}(\mathcal{T}) &= \{t_{\mathbb{I}} \mid t \in \mathcal{T}\}\end{aligned}$$

See proof on page 157.

Example 7.10. Consider our running example (introduced in Example 7.2), $V = \{+(x, +(z, y)) \mid x \leqslant y \wedge z \leqslant y\}$, we have $\alpha_{\mathcal{C}(\mathcal{R})}(V) = \{\wr 0 \int \mapsto \alpha, \wr 1, 0 \int \mapsto \gamma, \wr 1, 1 \int \mapsto \beta \mid \alpha \leqslant \beta \wedge \gamma \leqslant \beta\}$. The concretization of which is exactly V.

Example 7.11. Consider however the ranked alphabet $\{f(2), g(2), a(0)\}$, and the tree a. Its abstraction contains only the empty map, the concretization of which is the set of all terms that do not contain any numerical value. For example: $f(g(a, a), a), g(a, a), \ldots$. This emphasizes that we loose information on:

- the labels in the numerical terms: we only have the path from the root of the term to leaves with numerical labels, not the actual symbols along the path.
- the shape of the numerical terms: we do not keep any information on subterms that do not contain numerical values.

Now that we have abstracted away the shape of the automaton, we are left with numerical environments with potentially infinite dimensions (that are words over the alphabet $\{0, \ldots, n-1\}$) and different definition sets. Therefore following the idea of Section 6.7 we want to define a summarization for sets of words over the alphabet $\{0, \ldots, n-1\}$. A summarization of such a language can be expressed as a partition into sub-languages, moreover in order to define the numerical abstract set operators we will need to split and merge partitions in order to unify numerical abstract elements. For this reason we need to use a subset of the set of languages that is: machine representable, and closed under usual set operations. The set of regular languages over the alphabet $\{0, \ldots, n-1\}$ is a subset of the set of languages over this alphabet, that satisfies both constraints. Hence given a set $\{r_1, \ldots, r_m\}$ of regular expressions (with respective recognized languages $\{L_1, \ldots, L_m\}$), we summarize all words in L_i inside a common variable r_i

Definition 7.33 ($\uparrow \{r_1, \ldots, r_m\}$)**.** When $\{r_1, \ldots, r_m\}$ is a set of pair-wise non intersecting regular expressions, we define its associated summarization function to be:

$$\uparrow \{r_1, \ldots, r_m\} = \left\{ \begin{array}{ll} r_i & \mapsto \mathcal{L}(r_i) \\ \{r_1, \ldots, r_n\} & \to \wp(\mathcal{W}(\mathcal{R})) \end{array} \right.$$

In the following, Reg_n denotes the set of regular expressions over the alphabet $A_n \triangleq \{0, \ldots, n-1\}$. In order to disambiguate regular expressions over integers from integers we will typeset them within $\lfloor . \rfloor$ in a bold font as in: $\lfloor \mathbf{0} + \mathbf{0}.\mathbf{1}^\star \rfloor$.

Example 7.12. Using notations from Section 6.7, $\mathcal{V}' = \mathrm{Reg}_n$ and $\mathcal{V} = \mathcal{W}(\mathcal{R})$. Consider our running example (introduced in Example 7.2), numerical terms from $V = \{+(x, +(z, y)) \mid x \leqslant y \wedge z \leqslant y\}$ contain three paths to numerical values: $\wr 0 \int$, $\wr 1, 0 \int$ and $\wr 1, 1 \int$. Numerical constraints on $\wr 0 \int$ and $\wr 1, 0 \int$ are similar, therefore the two paths are summarized into one regular expression: $\lfloor \mathbf{0} + \mathbf{1}.\mathbf{0} \rfloor$, $\wr 1, 1 \int$ is left alone in its regular expression: $\lfloor \mathbf{1}.\mathbf{1} \rfloor$. The two constraints $x \leqslant y \wedge z \leqslant y$ can now be expressed as one: $\lfloor \mathbf{0} + \mathbf{1}.\mathbf{0} \rfloor \leqslant \lfloor \mathbf{1}.\mathbf{1} \rfloor$.

In Example 7.12, we saw that tree paths with similar numerical constraints can be summarized in one regular expression. For precision purposes, we do not want to summarize all tree paths into one regular expression. Hence, we will keep several disjoint regular expressions, which we call a subpartitioning. Contrary to a partitioning of s, a subpartitioning does not require that the set of partitions covers s. Indeed when a set of tree paths is unconstrained we can just remove it from the partitioning, therefore no dimension in the numerical abstract environment will be allocated for this path.

Definition 7.34 (Subpartitioning)**.** Given a regular expression s, a *subpartitioning* of s is a set $\{s_1, \dots, s_n\}$ of regular expressions such that $\forall i \neq j,\ s_i \cap s_j = \emptyset$ and $\bigcup_{i=1}^{n} s_i \subseteq s$. We note $P(s)$ the set of all subpartitioning of s. Moreover if $S = \{s_1, \dots, s_n\}$ is a set of regular expressions, $[S]_\emptyset = S \setminus \{\emptyset\}$.

The following definition uses an underlying numerical abstraction denoted $\mathfrak{M}^\sharp$ (we reuse the notation from the Clip domain from Section 6.4). The abstraction defined in this section is parametric in this underlying numeric domain. We assume that it provides sound abstract set operators as well as sound abstract transformers for the concrete semantics presented in Section 6.2. Obviously the Clip, $\wp$-Clip and partitioning abstractions defined in the previous chapter are examples of such abstractions. Moreover we denote as $\mathfrak{M}^\sharp_\mathfrak{p}$ the subset of $\mathfrak{M}^\sharp$ that represents set of numerical maps which supports are bounded by $\mathfrak{p}$. Note that (1) for the reminder of this chapter, the examples will be based on the Clip abstraction, (2) we abuse notations, when c is a set of constraints, $\{c\}$ refers to $\langle c, \mathbf{fv}(c), \mathbf{fv}(c)\rangle \in \mathfrak{M}^\sharp$ from the Clip abstraction.

Definition 7.35 (Positioning with numerical abstraction)**.** Given a ranked alphabet $\mathcal{R}$, where the maximum arity of symbols is n, we define $\mathcal{C}^\sharp(\mathcal{R}) = \{\langle s, \mathfrak{p}, R^\sharp\rangle \mid s \in \mathrm{Reg}_n, \mathfrak{p} \in P(s), R^\sharp \in \mathfrak{M}^\sharp_\mathfrak{p}\}$. Therefore $\mathcal{C}^\sharp(\mathcal{R})$ are triples containing:

- s: (called support) a regular expression coding for positions at which numerical values can be located.
- $\mathfrak{p}$: a subpartitioning of s. Elements of the same partition are subject to the same numerical constraints. Note that these partitions are regular.
- $R^\sharp$: an abstract numeric element, a dimension is associated to each partition, this dimension plays the role of a summary dimension.

Example 7.13 (Continuing Example 7.12)**.** V is represented by the abstract element: $V^\sharp = \langle \lfloor \mathbf{0} + \mathbf{1}.(\mathbf{0}+\mathbf{1})\rfloor, \{\lfloor \mathbf{0}+\mathbf{1}.\mathbf{0}\rfloor, \lfloor \mathbf{1}.\mathbf{1}\rfloor\}, \{\lfloor \mathbf{0}+\mathbf{1}.\mathbf{0}\rfloor \leqslant \lfloor \mathbf{1}.\mathbf{1}\rfloor\}\rangle$.

Unification

The previous definition shows that two elements $U^\sharp = \langle s, \mathfrak{p}, R^\sharp\rangle$ and $V^\sharp = \langle s', \mathfrak{p}', R^{\sharp\prime}\rangle$ can have different subpartitionings ($\mathfrak{p}$ and $\mathfrak{p}'$). However the partitions in $\mathfrak{p}$ and in $\mathfrak{p}'$ might overlap, thus giving constraints to similar tree paths. Therefore in order to define the classical operators: $\sqsubseteq, \sqcup$ and $\triangledown$, we need to unify the two abstract elements ($U^\sharp$ and $V^\sharp$) so that: given a tree path and the partition in which it is contained in $U^\sharp$, it is contained in the same partition in $V^\sharp$. This will enable us to rely on abstract operators of the numerical domain. In order to perform unification, we rely on the **expand** and **fold** operators. We recall that $\mathbf{fold}(R^\sharp, x, S)$ is used to summarize variables from S in variable x: this amounts to joining environments where every variable from S was renamed to x. Dually $\mathbf{expand}(R^\sharp, x, S)$ is used to de-summarize variable x to the set S: each new variable from S will behave as x with respect to other variables.

Indeed consider our running example:

$$\begin{array}{llll} U^\sharp & = \langle \lfloor \mathbf{0}+\mathbf{1}\rfloor, & \{\lfloor \mathbf{0}\rfloor, \lfloor \mathbf{1}\rfloor\}, & \{\lfloor \mathbf{0}\rfloor \leqslant \lfloor \mathbf{1}\rfloor\} \quad \rangle \\ V^\sharp & = \langle \lfloor \mathbf{0}+\mathbf{1}.(\mathbf{0}+\mathbf{1})\rfloor, & \{\lfloor \mathbf{0}+\mathbf{1}.\mathbf{0}\rfloor, \lfloor \mathbf{1}.\mathbf{1}\rfloor\}, & \{\lfloor \mathbf{0}+\mathbf{1}.\mathbf{0}\rfloor \leqslant \lfloor \mathbf{1}.\mathbf{1}\rfloor\} \quad \rangle \end{array}$$

We see that constraints on tree path $\wr 0 \int$ is given: in $U^\sharp$ by partition $\lfloor \mathbf{0} \rfloor$ and in $V^\sharp$ by partition $\lfloor \mathbf{0} + \mathbf{1.0} \rfloor$. However we can split the partition $\lfloor \mathbf{0} + \mathbf{1.0} \rfloor$ into two partitions: $\lfloor \mathbf{0} \rfloor$ and $\lfloor \mathbf{1.0} \rfloor$, and expand variable $\lfloor \mathbf{0} + \mathbf{1.0} \rfloor$ into the two variables $\lfloor \mathbf{0} \rfloor$ and $\lfloor \mathbf{1.0} \rfloor$ in the numeric component:

$$\mathbf{expand}(\{\lfloor \mathbf{0} + \mathbf{1.0} \rfloor \leqslant \lfloor \mathbf{1.1} \rfloor\}, \lfloor \mathbf{0} + \mathbf{1.0} \rfloor, \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1.0} \rfloor\}) = \{\lfloor \mathbf{0} \rfloor \leqslant \lfloor \mathbf{1.1} \rfloor, \lfloor \mathbf{1.0} \rfloor \leqslant \lfloor \mathbf{1.1} \rfloor\}$$

Once $U^\sharp$ and $V^\sharp$ are unified we can rely on the numerical join to soundly abstract the union. Note that we could also unify the partitions in $U^\sharp$ and $V^\sharp$ by merging them instead of splitting, but merging results in a loss of precision. Indeed, consider the example where: in $U^\sharp$ we have $\lfloor \mathbf{0} \rfloor \geqslant 0$ and $\lfloor \mathbf{1} \rfloor \leqslant 0$ and in $V^\sharp$ we have $\lfloor \mathbf{0} + \mathbf{1} \rfloor = 0$. Splitting partitions in $V^\sharp$ yields: $\lfloor \mathbf{0} \rfloor = 0, \lfloor \mathbf{1} \rfloor = 0$, after joining we get $\lfloor \mathbf{0} \rfloor \geqslant 0, \lfloor \mathbf{1} \rfloor \leqslant 0$. Whereas merging partitions in $U^\sharp$ yields $\lfloor \mathbf{0} + \mathbf{1} \rfloor$ unconstrained, after joining we also get that $\lfloor \mathbf{0} + \mathbf{1} \rfloor$ is unconstrained. However unifying by splitting or merging partitions in both abstract elements might result in an over-approximation of the initial elements. This does not pose a threat to the soundness of the join operator, but it does for the inclusion test. Unifying by splitting partitions induces an increase in the number of partitions which we want to avoid when trying to stabilize abstract elements in the widening. Hence, we define three unification operators:

- An operator **unify_join** that splits partitions from $U^\sharp$ and $V^\sharp$, this operator might induce an over-approximation for both $U^\sharp$ and $V^\sharp$ and is used in the join operation. This operator is presented in Algorithm 7.7, and illustrated in Figure 7.6. Once elements are unified we can distinguish three kinds of partitions:
  - partitions found in both abstract elements (▩ in Figure 7.6).
  - partitions found in only one of the two, which do not overlap over the support of the other abstract element (denoted u^o), these are outer-partitions. Information on such partitions can be soundly kept when joining two abstract elements (partition a in Figure 7.6).
  - partitions found in only one of the two, which overlap over the support of the other abstract element, these are inner-partitions. Information on such partitions can not be soundly kept when joining two abstract elements. (partition b in Figure 7.6)
- An operator **unify_subset** that does not modify $V^\sharp$ (in order to avoid over-approximating it), we only split and merge (using the **fold** operator) partitions from $U^\sharp$ as, if the over-approximated $U^\sharp$ is smaller than $V^\sharp$, then so is the original $U^\sharp$. This operator is presented in Algorithm 7.8.
- An operator **unify_widen** that unifies $U^\sharp$ and $V^\sharp$ by only merging partitions so that the number of partitions does not increase. This operator is used in the widening definition, it is defined in Algorithm 7.9. Algorithm 7.9 uses a **connected_component** function that given two sets of vertices V_1 and V_2 and a set of edges in $V_1 \times V_2$ computes a set of pairs $C_1 \subseteq V_1, C_2 \subseteq V_2$ of maximal (for the inclusion) connected component such that: $V_1 \subseteq \cup_{(C_1, C_2) \in \mathbf{connected\_component}(V_1, V_2, E)} C_1$. This can be obtained by iteratively merging partitions that overlap in both arguments until the abstract elements have the exact same partitions, this is similar to the merging done in Algorithm 6.7. As an example, on the two subpartitionings $\{\lfloor \mathbf{0} + \mathbf{1} \rfloor, \lfloor \mathbf{1.1} \rfloor, \lfloor \mathbf{1.0} \rfloor\}$ and $\{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1} + \mathbf{1.1} \rfloor, \lfloor \mathbf{1.0} \rfloor\}$. This will compute the pairs: $(\{\lfloor \mathbf{0} + \mathbf{1} \rfloor, \lfloor \mathbf{1.1} \rfloor\}, \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1} + \mathbf{1.1} \rfloor\})$ and $(\lfloor \mathbf{1.0} \rfloor, \lfloor \mathbf{1.0} \rfloor)$

Example 7.14. As an illustration of these three algorithms consider the following example:

$$\begin{array}{rlllll} U^\sharp = & \langle & \lfloor \mathbf{0} + \mathbf{1} + \mathbf{2} \rfloor, & \{\lfloor \mathbf{0} + \mathbf{1} \rfloor, \lfloor \mathbf{2} \rfloor\}, & \{\lfloor \mathbf{0} + \mathbf{1} \rfloor = 0, \lfloor \mathbf{2} \rfloor = 2\} & \rangle \\ V^\sharp = & \langle & \lfloor \mathbf{0} + \mathbf{1} + \mathbf{2} \rfloor, & \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1} + \mathbf{2} \rfloor\}, & \{\lfloor \mathbf{0} \rfloor = 0, \lfloor \mathbf{1} + \mathbf{2} \rfloor = 2\} & \rangle \end{array}$$

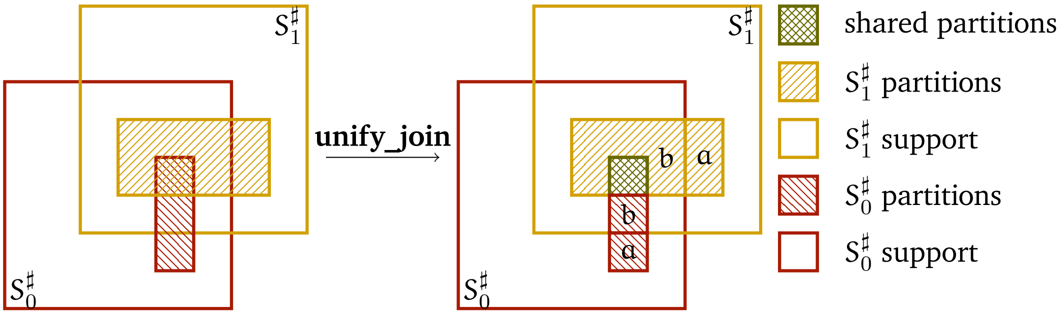


Figure 7.6: Unification operator

Algorithm 7.7: unify_join operator

```
Input  : $\langle s, \{p_1, \dots, p_n\}, R^\sharp\rangle, \langle s', \{p'_1, \dots, p'_m\}, R^{\sharp\prime}\rangle$ two abstract elements
Output: two unified abstract elements
$(\underline{c_{i,j}})_{i \leqslant n, j \leqslant m} \leftarrow p_i \cap p'_j$;
$(\underline{p_i})_{i \leqslant n} \leftarrow p_i \setminus s'$;
$(\underline{p'_j})_{j \leqslant m} \leftarrow p'_j \setminus s$;
$(\underline{q_i})_{i \leqslant n} \leftarrow p_i \cap s' \cap (\cup_{j \leqslant m} \underline{c_{i,j}})^c$;
$(\underline{q'_j})_{j \leqslant m} \leftarrow p'_j \cap s \cap (\cup_{i \leqslant n} \underline{c_{i,j}})^c$;
$\underline{R^\sharp} \leftarrow R^\sharp$ ;
$\underline{R^{\sharp\prime}} \leftarrow R^{\sharp\prime}$ ;
for $i = 1$ to $n$ do
    $R^\sharp \leftarrow$ expand$(R^\sharp, p_i, [\{c_{i,j}\}_{j \leqslant m} \cup \{\underline{p_i}\} \cup \{\underline{q_i}\}]_\emptyset)$;
for $j = 1$ to $m$ do
    $R^{\sharp\prime} \leftarrow$ expand$(R^{\sharp\prime}, p'_j, [\{c_{i,j}\}_{i \leqslant n} \cup \{\underline{p'_j}\} \cup \{\underline{q'_j}\}]_\emptyset)$;
return $\langle s, \bigcup_{i \leqslant n, j \leqslant m} [\{\underline{q_i}, \underline{p_i}, \underline{c_{i,j}}\}]_\emptyset, \underline{R^\sharp}\rangle, \langle s', \bigcup_{i \leqslant n, j \leqslant m} [\{\underline{q'_i}, \underline{p'_j}, \underline{c_{i,j}}\}]_\emptyset, \underline{R^{\sharp\prime}}\rangle$;
```

Algorithm 7.8: unify_subset operator

```
Input  : $\langle s_1, \{p_1, \dots, p_n\}, R_1^\sharp\rangle, \langle s_2, \{p'_1, \dots, p'_m\}, R_2^\sharp\rangle$ two abstract elements
Output: $U^\sharp$, $V^\sharp$ two unified abstract elements, $V^\sharp$ is not modified
$\{\underline{c_{i,j}}\}_{i \leqslant n, j \leqslant m} \leftarrow p_i \cap p'_j$;
$R^{\sharp\star} \leftarrow R_1^\sharp$;
$\mathfrak{p}^\star \leftarrow \emptyset$;
for $i = 1$ to $n$ do
    if $[\{\underline{c_{i,j}}\}_{j \leqslant m}]_\emptyset = \emptyset$ then
        $R^{\sharp\star} \leftarrow \mathbb{S}^\sharp[\![\mathtt{remove}(p_i)]\!](R^{\sharp\star})$;
    else
        $R^{\sharp\star} \leftarrow$ expand$(R^{\sharp\star}, p_i, [\{c_{i,j}\}_{j \leqslant m}]_\emptyset)$;
for $j = 1$ to $m$ do
    $b \leftarrow \bigcup_{i=1}^{n} \underline{c_{i,j}}$;
    if $b \neq \emptyset$ then
        $R^{\sharp\star} \leftarrow$ fold$(R^{\sharp\star}, b, [\{c_{i,j}\}_{i \leqslant n}]_\emptyset)$;
        $\mathfrak{p}^\star \leftarrow \{b\} \cup \mathfrak{p}^\star$
return $\langle s_1, \mathfrak{p}^\star, R^{\sharp\star}\rangle, \langle s_2, \{p'_1, \dots, p'_m\}, R_2^\sharp\rangle$
```

Algorithm 7.9: unify_widen operator

```
Input  : $\langle s_1, \mathfrak{p}_1, R_1^\sharp\rangle, \langle s_2, \mathfrak{p}_2, R_2^\sharp\rangle$ two abstract elements
Output: $U^\sharp$, $V^\sharp$ two unified abstract elements
$\widetilde{\mathfrak{p}_1} \leftarrow \{p \in \mathfrak{p}_1 \mid p \subseteq s_2^c \cup \bigcup_{p' \in \mathfrak{p}_2} p'\}$;
$V_1 \leftarrow \widetilde{\mathfrak{p}_1}$ ;
$V_2 \leftarrow \mathfrak{p}_2$;
$E \leftarrow \{(p_1, p_2) \in \widetilde{\mathfrak{p}_1} \times \mathfrak{p}_2 \mid p_1 \cap p_2 \neq \emptyset\}$;
$S \leftarrow$ connected_component($(V_1, V_2, E)$);
$\mathfrak{p}_1^\star \leftarrow \emptyset$; $\mathfrak{p}_2^\star \leftarrow \emptyset$; $R_1^{\sharp\star} \leftarrow R_1^\sharp$; $R_2^{\sharp\star} \leftarrow R_2^\sharp$;
foreach $(C_1, C_2) \in S$ do
    $a \leftarrow \cup_{e \in C_1} e$;
    $b \leftarrow \cup_{e' \in C_2} e'$;
    if $b \neq \emptyset$ then
        $\mathfrak{p}_2^\star \leftarrow \{b\} \cup \mathfrak{p}_2^\star$;
        $R_2^{\sharp\star} \leftarrow$ fold($R_2^{\sharp\star}, b, C_2$);
        $\mathfrak{p}_1^\star \leftarrow \{a\} \cup \mathfrak{p}_1^\star$;
        $R_1^{\sharp\star} \leftarrow$ fold($R_1^{\sharp\star}, a, C_1$);
    $f \leftarrow f \uplus [a \mapsto b]$;
endfch
return $\langle s_1, \mathfrak{p}_1^\star, (R_1^{\sharp\star})_{|\mathfrak{p}_1^\star}\rangle$, $\langle s_2, \mathfrak{p}_2^\star, (R_2^{\sharp\star})_{|\mathfrak{p}_2^\star}\rangle$, $f$
```

the different unification operators will produce the following results:

unify_join	$U^\sharp =$	$\langle$	$\lfloor \mathbf{0}+\mathbf{1}+\mathbf{2}\rfloor$	$\{\lfloor\mathbf{0}\rfloor, \lfloor\mathbf{1}\rfloor, \lfloor\mathbf{2}\rfloor\}$	$\{\lfloor\mathbf{0}\rfloor = 0, \lfloor\mathbf{1}\rfloor = 0, \lfloor\mathbf{2}\rfloor = 2\}$	$\rangle$
	$V^\sharp =$	$\langle$	$\lfloor \mathbf{0}+\mathbf{1}+\mathbf{2}\rfloor$	$\{\lfloor\mathbf{0}\rfloor, \lfloor\mathbf{1}\rfloor, \lfloor\mathbf{2}\rfloor\}$	$\{\lfloor\mathbf{0}\rfloor = 0, \lfloor\mathbf{1}\rfloor = 2, \lfloor\mathbf{2}\rfloor = 2\}$	$\rangle$
unify_leq	$U^\sharp =$	$\langle$	$\lfloor \mathbf{0}+\mathbf{1}+\mathbf{2}\rfloor$	$\{\lfloor\mathbf{0}\rfloor, \lfloor\mathbf{1}+\mathbf{2}\rfloor\}$	$\{\lfloor\mathbf{0}\rfloor = 0, 0 \leqslant \lfloor\mathbf{1}+\mathbf{2}\rfloor \leqslant 2\}$	$\rangle$
	$V^\sharp =$	$\langle$	$\lfloor \mathbf{0}+\mathbf{1}+\mathbf{2}\rfloor$	$\{\lfloor\mathbf{0}\rfloor, \lfloor\mathbf{1}+\mathbf{2}\rfloor\}$	$\{\lfloor\mathbf{0}\rfloor = 0, \lfloor\mathbf{1}+\mathbf{2}\rfloor = 2\}$	$\rangle$
unify_widen	$U^\sharp =$	$\langle$	$\lfloor \mathbf{0}+\mathbf{1}+\mathbf{2}\rfloor$	$\{\lfloor\mathbf{0}+\mathbf{1}+\mathbf{2}\rfloor\}$	$\{0 \leqslant \lfloor\mathbf{0}+\mathbf{1}+\mathbf{2}\rfloor \leqslant 2\}$	$\rangle$
	$V^\sharp =$	$\langle$	$\lfloor \mathbf{0}+\mathbf{1}+\mathbf{2}\rfloor$	$\{\lfloor\mathbf{0}+\mathbf{1}+\mathbf{2}\rfloor\}$	$\{0 \leqslant \lfloor\mathbf{0}+\mathbf{1}+\mathbf{2}\rfloor \leqslant 2\}$	$\rangle$

Example 7.15. Consider the following abstract elements on the ranked alphabet $\{f(2)\}$: $U^\sharp = \langle\lfloor\mathbf{0}+\mathbf{1}\rfloor, \{\lfloor\mathbf{0}\rfloor, \lfloor\mathbf{1}\rfloor\}, \{\lfloor\mathbf{0}\rfloor = \lfloor\mathbf{1}\rfloor\}\rangle$, $V^\sharp = \langle\lfloor\mathbf{0}+\mathbf{1}+\mathbf{0.1}\rfloor, \{\lfloor\mathbf{0}+\mathbf{0.1}\rfloor, \lfloor\mathbf{1}\rfloor\}, \{\lfloor\mathbf{0}+\mathbf{0.1}\rfloor \geqslant \lfloor\mathbf{1}\rfloor\}\rangle$. In the reminder of the example, notations are taken from the definition of the application of **unify_join** to $U^\sharp$, $V^\sharp$ (Algorithm 7.7). $p_1 = \lfloor\mathbf{0}\rfloor$, $p_2 = \lfloor\mathbf{1}\rfloor$, $p'_1 = \lfloor\mathbf{0}+\mathbf{0.1}\rfloor$, $p'_2 = \lfloor\mathbf{1}\rfloor$, then $\underline{p'_1} = \lfloor\mathbf{0.1}\rfloor$, and $\underline{c_{1,1}} = \lfloor\mathbf{0}\rfloor$, $\underline{c_{2,2}} = \lfloor\mathbf{1}\rfloor$ (unmentioned values are $\emptyset$). Therefore: **unify_join**$(U^\sharp, V^\sharp) = \langle\lfloor\mathbf{0}+\mathbf{1}\rfloor, \{\lfloor\mathbf{0}\rfloor, \lfloor\mathbf{1}\rfloor\}, \{\lfloor\mathbf{0}\rfloor = \lfloor\mathbf{1}\rfloor\}(\triangleq R^\sharp)\rangle, \langle\lfloor\mathbf{0}+\mathbf{1}+\mathbf{0.1}\rfloor, \{\lfloor\mathbf{0}\rfloor, \lfloor\mathbf{1}\rfloor, \lfloor\mathbf{0.1}\rfloor\}, \{\lfloor\mathbf{0}\rfloor \geqslant \lfloor\mathbf{1}\rfloor, \lfloor\mathbf{0.1}\rfloor \geqslant \lfloor\mathbf{1}\rfloor\}(\triangleq R^{\sharp\prime})$.

Set operators

Definition 7.36 (Comparison $\sqsubseteq_{\mathcal{C}^\sharp(\mathcal{R})}$)**.** Using **unify_subset** we define a relation on $\mathcal{C}^\sharp(\mathcal{R})$:

$$\begin{aligned}\sqsubseteq_{\mathcal{C}^\sharp(\mathcal{R})} = \{(U^\sharp, V^\sharp) \mid (\langle s, \mathfrak{p}, N^\sharp\rangle, \langle s', \mathfrak{p}', N^{\sharp\prime}\rangle) &= \textbf{unify\_subset}(U^\sharp, V^\sharp) \\ \Rightarrow s \subseteq s' \wedge \forall b \in \mathfrak{p}', (b \subseteq s^c \vee \exists! a \in \mathfrak{p},\ b \cap s = a) \wedge N^\sharp &\sqsubseteq N^{\sharp\prime}[\phi]\}\end{aligned}$$

where ϕ is the renaming from $\mathfrak{p}'$ into $\mathfrak{p}$ that renames b to a when such an a exists.

Example 7.16. Going back to our running example: $U^\sharp = \langle\lfloor\mathbf{0}+\mathbf{1}\rfloor, \{\lfloor\mathbf{0}\rfloor, \lfloor\mathbf{1}\rfloor\}, \{\lfloor\mathbf{0}\rfloor \leqslant \lfloor\mathbf{1}\rfloor\}(= A^\sharp)\rangle$ and $V^\sharp = \langle\lfloor\mathbf{0}+\mathbf{1}.(\mathbf{0}+\mathbf{1})\rfloor, \{\lfloor\mathbf{0}+\mathbf{1.0}\rfloor, \lfloor\mathbf{1.1}\rfloor\}, \{\lfloor\mathbf{0}+\mathbf{1.0}\rfloor \leqslant \lfloor\mathbf{1.1}\rfloor\}\rangle$. We have $s \not\subseteq s'$ hence $U^\sharp \not\sqsubseteq V^\sharp$. However if we now consider $W^\sharp$: $\langle\lfloor(\epsilon+\mathbf{1}).(\mathbf{0}+\mathbf{1})\rfloor, \{\lfloor(\epsilon+\mathbf{1}).\mathbf{0}\rfloor, \lfloor(\epsilon+\mathbf{1}).\mathbf{1}\rfloor\}, \{\lfloor(\epsilon+\mathbf{1}).\mathbf{0}\rfloor \leqslant \lfloor(\epsilon+\mathbf{1}).\mathbf{1}\rfloor\}(= B^\sharp)\rangle$. $W^\sharp$ is already unified with $U^\sharp$, we have $s \subseteq s'$ and $\phi : (\lfloor(\epsilon+\mathbf{1}).\mathbf{0}\rfloor \mapsto \mathbf{0}, \lfloor(\epsilon+\mathbf{1}).\mathbf{1}\rfloor \mapsto \lfloor\mathbf{1}\rfloor)$. Moreover $A^\sharp \sqsubseteq B^\sharp[\phi] = \{\lfloor\mathbf{0}\rfloor \leqslant \lfloor\mathbf{1}\rfloor\}$. Hence $U^\sharp \sqsubseteq W^\sharp$.

Proposition 7.20. *We have the following representation:*

$$(\mathcal{C}(\mathcal{R}), \sqsubseteq_{\mathcal{C}(\mathcal{R})}) \xleftarrow{\gamma_1} (\mathcal{C}^\sharp(\mathcal{R}), \sqsubseteq_{\mathcal{C}^\sharp(\mathcal{R})})$$

where:

$$\gamma_1(\langle s, \mathfrak{p}, R^\sharp \rangle) = \{\rho \mid \boldsymbol{def}(\rho) \subseteq \gamma_{Reg_n}(s) \land \rho \in \gamma[\uparrow \mathfrak{p}](R^\sharp)\}$$

By composition we get: $(\wp(T_{\mathbb{I}}(\mathcal{R})), \subseteq) \xleftarrow{\gamma_2} (\mathcal{C}^\sharp(\mathcal{R}), \sqsubseteq_{\mathcal{C}^\sharp\mathcal{R}})$, *with* $\gamma_2 = \gamma_{\mathcal{C}(\mathcal{R})} \circ \gamma_1$.

See proof on page 157.

Example 7.17. Going back to our running example: $V^\sharp = \langle \lfloor \mathbf{0} + \mathbf{1}.(\mathbf{0} + \mathbf{1}) \rfloor, \{\lfloor \mathbf{0} + \mathbf{1}.\mathbf{0} \rfloor, \lfloor \mathbf{1}.\mathbf{1} \rfloor\}, \{\lfloor \mathbf{0} + \mathbf{1}.\mathbf{0} \rfloor \leqslant \lfloor \mathbf{1}.\mathbf{1} \rfloor\} \rangle$. We have: $\uparrow \mathfrak{p} = (\lfloor \mathbf{0} + \mathbf{1}.\mathbf{0} \rfloor \mapsto \{\wr 0 \int, \wr 1, 0 \int\}, \lfloor \mathbf{1} \rfloor \mapsto \wr 1 \int)$. Hence, $\gamma_1(V^\sharp) = \{(\wr 0 \int \mapsto \alpha, \wr 1 \int \mapsto \beta) \mid \alpha \leqslant \beta\} \cup \{(\wr 1, 0 \int \mapsto \alpha, \wr 1 \int \mapsto \beta) \mid \alpha \leqslant \beta\} \cup \{(\wr 0 \int \mapsto \alpha, \wr 1, 0 \int \mapsto \gamma, \wr 1 \int \mapsto \beta) \mid \alpha \leqslant \beta \land \gamma \leqslant \beta\}$.

We now define the $\sqcup$ operator that relies on the **unify_join** operator of Algorithm 7.7. In the following definition of the join operator, we compute (once elements are unified) the common partitions and both outer-partitions and merge them to form the resulting subpartitioning.

Definition 7.37 (Union abstract operator)**.** Given $U^\sharp, V^\sharp \in \mathcal{C}^\sharp(\mathcal{R})$, if

$$(\langle s, \mathfrak{p}, R^\sharp \rangle, \langle s', \mathfrak{p}', R^{\sharp\prime} \rangle) = \textbf{unify\_join}(U^\sharp, V^\sharp)$$

let $\mathfrak{c}$ be $\mathfrak{p} \cup \mathfrak{p}'$, let u^o ($U^\sharp$ outer-partition) be $\{e \in \mathfrak{p} \mid e \subseteq s'^c\}$, let v^o ($V^\sharp$ outer-partition) be $\{e \in \mathfrak{p}' \mid e \subseteq s^c\}$, we then define:

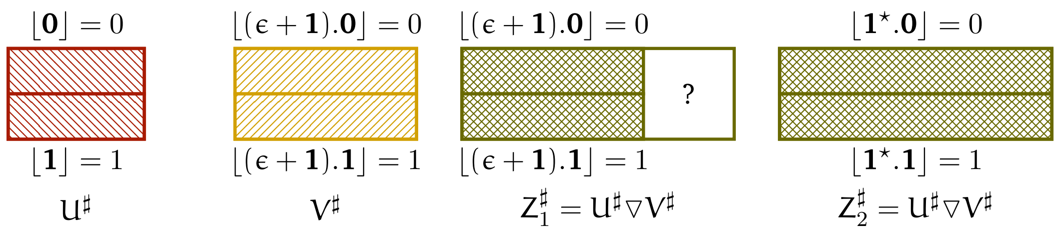
$$U^\sharp \sqcup_{\mathcal{C}^\sharp(\mathcal{R})} V^\sharp = \langle s \cup s', \mathfrak{c} \cup u^o \cup v^o, R^\sharp_{|\mathfrak{c} \cup u^o} \sqcup R^{\sharp\prime}_{|\mathfrak{c} \cup v^o} \rangle$$

Proposition 7.21 ($\sqcup$ is sound)**.** *We have:* $\forall U^\sharp, V^\sharp \in \mathcal{C}^\sharp(\mathcal{R}), \gamma_1(U^\sharp) \cup \gamma_1(V^\sharp) \subseteq \gamma_1(U^\sharp \sqcup_{\mathcal{C}^\sharp(\mathcal{R})} V^\sharp)$.

See proof on page 158.

Example 7.18. Consider the two following abstract elements (this is the particular case of our running example where all numerical values are equal): $V^\sharp = \langle \lfloor \mathbf{0} + \mathbf{1}.(\mathbf{0} + \mathbf{1}) \rfloor (= s), \{\lfloor \mathbf{0} + \mathbf{1}.\mathbf{0} \rfloor (= a), \lfloor \mathbf{1}.\mathbf{1} \rfloor (= b), \{a = b\}\} \rangle$, and $U^\sharp = \langle \lfloor \mathbf{0} + \mathbf{1} \rfloor (= s'), \{\lfloor \mathbf{0} \rfloor (= c), \lfloor \mathbf{1} \rfloor (= d)\}, \{c = d\} \rangle$. Intuitively $U^\sharp$ could encode the term $(x + x)$ and $V^\sharp$ the term $(x + (x + x))$. The unification of those two elements is: $V^\sharp_1 = \langle s, \{c, b, \lfloor \mathbf{1}.\mathbf{0} \rfloor (= e)\}, R^\sharp \rangle$ where $R^\sharp = \langle \{c = b, e = b\}, \{b\}, \{c, b, e\} \rangle$ and $U^\sharp_1 = U^\sharp$, moreover the common environment ($\mathfrak{c}$ in previous definition) is: $\{c\}$, $V^\sharp$ outer-partitioning is $\{e, f\}$, $U^\sharp$ outer-partitioning is $\{d\}$. Hence: the numerical component resulting of the join is: $\langle \{c = d\}, \{c, d\}, \{c, d\} \rangle \sqcup \langle \{c = b, e = b\}, \{b\}, \{c, b, e\} \rangle$ which is: $\langle \{c = b, e = b, c = d\}, \emptyset, \{c, d, e, b\} \rangle$. We see here that using a naive numerical join operator, instead of the one provided by the Clip abstraction using a **strenghtening** operator, we would not have been able to get such a precise result (the numerical join would have yielded $\top$).

unify_widen. $\mathcal{C}^\sharp(\mathcal{R})$ contains infinite increasing chains, therefore we need to provide a widening operator. As for the other operators, widening is computed on unified abstract elements. However, recall that in order not to increase the number of partitions, we defined **unify_widen** (in Algorithm 7.9, illustrated in Figure 7.8). This operator produces $U^\sharp$ and $V^\sharp$, over approximations of its inputs with the same number of partitions. Moreover it ensures that each partition of $U^\sharp$ intersects exactly one partition of $V^\sharp$. This can be obtained by iterative merging partitions that overlap in both arguments until the abstract elements have the exact same partitions. Therefore from the result of **unify_widen** we can extract a list of pairs (a, b) where a is a partition from $U^\sharp$, b is a partition from $V^\sharp$ and $a \cap b \neq \emptyset$. This defines a bijection from partitions of $U^\sharp$ onto partitions of $V^\sharp$.



$\lfloor \mathbf{0} \rfloor = 0$
$\lfloor (\epsilon + \mathbf{1}).\mathbf{0} \rfloor = 0$
$\lfloor (\epsilon + \mathbf{1}).\mathbf{0} \rfloor = 0$
$\lfloor \mathbf{1}^\star.\mathbf{0} \rfloor = 0$
?
$\lfloor \mathbf{1} \rfloor = 1$
$\lfloor (\epsilon + \mathbf{1}).\mathbf{1} \rfloor = 1$
$\lfloor (\epsilon + \mathbf{1}).\mathbf{1} \rfloor = 1$
$\lfloor \mathbf{1}^\star.\mathbf{1} \rfloor = 1$
$U^\sharp$
$V^\sharp$
$Z_1^\sharp = U^\sharp \triangledown V^\sharp$
$Z_2^\sharp = U^\sharp \triangledown V^\sharp$

Figure 7.7: Widening illustration

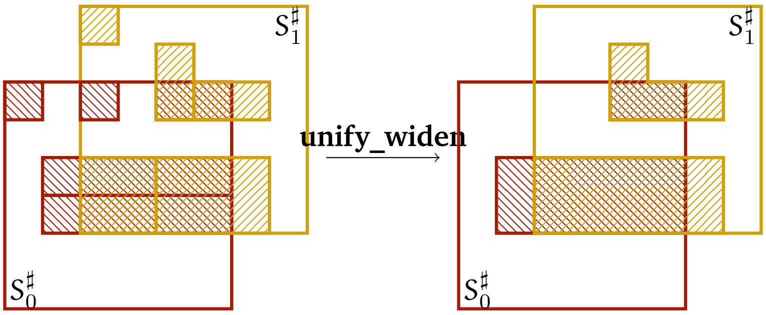


Figure 7.8: **unify_widen**

Extend. In order to ensure stabilization we first need to stabilize the supports on which abstract elements are defined. This is easily done using the automaton widening ($s_1 \triangledown s_2$ in Algorithm 7.10). Figure 7.7 illustrates the following simple example: $U^\sharp$ is an abstract element with support $\lfloor \mathbf{0} + \mathbf{1} \rfloor$, two partitions $u = \lfloor \mathbf{0} \rfloor$ and $u' = \lfloor \mathbf{1} \rfloor$, and numerical constraints $u' = 1$ and $u = 0$. $V^\sharp$ is an abstract element with support $\lfloor (\epsilon + \mathbf{1}).(\mathbf{0} + \mathbf{1}) \rfloor$, two partitions $v = \lfloor (\epsilon + \mathbf{1}).\mathbf{0} \rfloor$ and $v' = \lfloor (\epsilon + \mathbf{1}).\mathbf{1} \rfloor$ with the numerical constraints that $v = 0$ and $v' = 1$. Supports are unstable, therefore we start by widening them, which yields a new support: $\lfloor \mathbf{1}^\star.(\mathbf{0} + \mathbf{1}) \rfloor$. The unification of $U^\sharp$ and $V^\sharp$ leaves subpartitionings unchanged and yields the bijection $(u \mapsto v, u' \mapsto v')$. Given this information we now need to provide a new subpartitioning for the result of the widening. We see in this example that we could soundly use the subpartitioning from $V^\sharp$, this would produce the abstract element $Z_1^\sharp$ depicted in Figure 7.7. However due to the widening of the support, paths of the form $\wr 1, 1, 1, 0 \wr$ are in the support of the result but are left unconstrained as they are not in any of the partitions. Therefore we need to use the opportunity of the extension of the support to place constraints on the newly added paths. Indeed it is always sound to add constraints on paths that can not be found in any of the arguments, the idea is therefore to choose the constraints based on the "proximity" of the newly added paths with respect to already existing partitions. In order to do so we would like to force the extension of the existing partitions from $U^\sharp$ and $V^\sharp$ into the new support. Therefore we need to define a **extend** operator that produces a sound new partition, given: (1) a pair a, b of partitions (such as the one produced by **unify_widen**), (2) the support s_1 (resp. s_2) in which a (resp. b) lives and (3) a space to occupy r. The following criteria must be verified by the resulting partition p in order to be sound and to terminate: $p \cap s_1 = a$, $p \cap s_2 = b$ and $p \setminus (s_1 \cup s_2) \subseteq r$. A variety of **extend** operators could be defined.

Definition 7.38 (extend). Given a, b, s_1, s_2 and r regular expressions, we define:

$$\mathbf{extend}(a, b, s_1, s_2, r) = a \cup (b \cap (s_2 \setminus s_1)) \cup ((a \triangledown (a \cup b)) \cap r)$$

The idea is the following: we keep a (as it is always sound thanks to the definition of the **unify_widen** operator), we keep the part from b that satisfies the soundness condition, and we extend into the space left to occupy according to the automata widening of a and $a \cup b$. In our

Algorithm 7.10: widening operator

```
Input : $U^\sharp, V^\sharp$ two abstract elements
$(\langle s_1, \mathfrak{p}_1, R_1^\sharp\rangle, \langle s_2, \mathfrak{p}_2, R_2^\sharp\rangle) \leftarrow$ unify_widen$(U^\sharp, V^\sharp)$ ;
$s \leftarrow s_1 \nabla s_2$;
$r \leftarrow s \setminus (s_1 \cup s_2)$;
foreach $a \in \mathfrak{p}_1$ do
    $b \leftarrow$ the unique element from $\mathfrak{p}_2$ such that $b \cap a \neq \emptyset$;
    $p \leftarrow$ extend$(a, b, s_1, s_2, r)$;
    $\mathfrak{p} \leftarrow \{p\} \cup \mathfrak{p}$;
    $R_1^{\sharp\star} \leftarrow R_1^{\sharp\star}[a \mapsto p]$;
    $R_2^{\sharp\star} \leftarrow R_1^{\sharp\star}[b \mapsto p]$;
    $r \leftarrow r \setminus p$;
if $\mathfrak{p} = \mathfrak{p}_1$ then
    return $\langle s, \mathfrak{p}, R_1^{\sharp\star} \nabla R_2^{\sharp\star}\rangle$;
else
    return $\langle s, \mathfrak{p}, R_1^{\sharp\star} \sqcup R_2^{\sharp\star}\rangle$;
```

example, considering the pair (u, v), this would translate as: $a = \lfloor \mathbf{0} \rfloor$, $b \cap (s_2 \setminus s_1) = \lfloor \mathbf{1.0} \rfloor$ and $(a \nabla (a \cup b)) \cap r = \lfloor \mathbf{0} \rfloor \nabla \lfloor (\epsilon + 1).\mathbf{0} \rfloor \cap \lfloor \mathbf{1}^{\geqslant 2}(\mathbf{0} + \mathbf{1}) \rfloor = \lfloor \mathbf{1}^{\geqslant 2}.\mathbf{0} \rfloor$. We get the new partition: $\lfloor \mathbf{1}^\star.\mathbf{0} \rfloor$. Doing the same with the pair (v, v') yields $\lfloor \mathbf{1}^\star.\mathbf{1} \rfloor$. Finally we get the abstract element $Z_2^\sharp$ from Figure 7.7, which is more precise than $Z_1^\sharp$.

Definition 7.39 (Widening)**.** Algorithm 7.10 provides the definition of a widening operator using the **unify_widen** operator and parameterized by a **extend** function.

Widening stabilization. Our abstraction contains three components: (1) a support that describes the set of paths to numerical positions in the tree, (2) a subpartitioning of this support and (3) a numerical component giving constraints on partitions in the subpartitioning. We will show how the widening operator from Algorithm 7.10 stabilizes all three components.

- Regular expression widening is used on supports when widening is called. Therefore ensuring support stabilization.
- Once supports are stable (this means $s_2 \subseteq s_1$), we have $p = a$ for every pair (a, b) of partitions. Meaning that once shapes stabilize, the only modifications allowed on the subpartitionings are those made by the **unify_widen** operator. Moreover each partition resulting from the operator is the union of input partitions, when the shape stabilizes there is only a finite number of those, hence the subpartitioning will stabilize.
- Once subpartitionings are stable ($\mathfrak{p}_1 = \mathfrak{p}$ in Algorithm 7.10) numerical widening is applied on the numerical component in order to ensure stabilization.

Proposition 7.22 (Soundness and termination of the widening operator)**.** *The ∇ operator is sound: $\forall U_1^\sharp, U_2^\sharp, \gamma(U_1^\sharp) \subseteq \gamma(U_1^\sharp \nabla U_2^\sharp) \wedge \gamma(U_2^\sharp) \subseteq \gamma(U_1^\sharp \nabla U_2^\sharp)$. Moreover it ensures stabilization of infinite sequences: $\forall (U_n^\sharp)_{n \in \mathbb{N}}, (V_n^\sharp)_{n \in \mathbb{N}}, V_0^\sharp = U_0^\sharp \wedge \forall n \geqslant 0, V_{n+1}^\sharp = V_n^\sharp \nabla U_{n+1}^\sharp \Rightarrow \exists N \in \mathbb{N}, \forall k \geqslant N, V_k^\sharp = V_N^\sharp$.*

Example 7.19. Consider the following example on $\mathcal{R} = \{f(2)\}$ (Figure 7.9 depicts some of the values mentioned in this example): $U^\sharp = \langle \lfloor \mathbf{0} + \mathbf{1} \rfloor, \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1} \rfloor\}, \{\lfloor \mathbf{0} \rfloor \leqslant \lfloor \mathbf{1} \rfloor\}$ and $V^\sharp = \langle \lfloor \mathbf{0} + \mathbf{1} + \mathbf{1.0} + \mathbf{1.1} \rfloor, \{\lfloor \mathbf{0} + \mathbf{1.0} \rfloor, \lfloor \mathbf{1} + \mathbf{1.1} \rfloor\}, \{\lfloor \mathbf{0} + \mathbf{1.0} \rfloor \leqslant \lfloor \mathbf{1} + \mathbf{1.1} \rfloor\}\rangle$. With notations from Algorithm 7.10 and Algorithm 7.8 we have the connected components C_0 and C_1, depicted in Fig 7.9. Regular expression widening of s_1 with s_2 yields $s = \lfloor \mathbf{1}^\star.(\mathbf{1} + \mathbf{0}) \rfloor$.

- For the first loop iteration, considering component $C_0 = \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{0} + \mathbf{1.0} \rfloor\}$ we get: $r = s \setminus (s_1 \cup s_2) = \lfloor \mathbf{1.1.1}^\star.(\mathbf{0} + \mathbf{1}) \rfloor$, $\mathfrak{a} = \lfloor \mathbf{0} \rfloor$, $\mathfrak{b} = \lfloor \mathbf{0} + \mathbf{1.0} \rfloor$, $p = \lfloor \mathbf{0} \rfloor \cup \lfloor \mathbf{1.0} \rfloor \cup (\lfloor \mathbf{0} \rfloor \nabla_r \lfloor \mathbf{0} +$

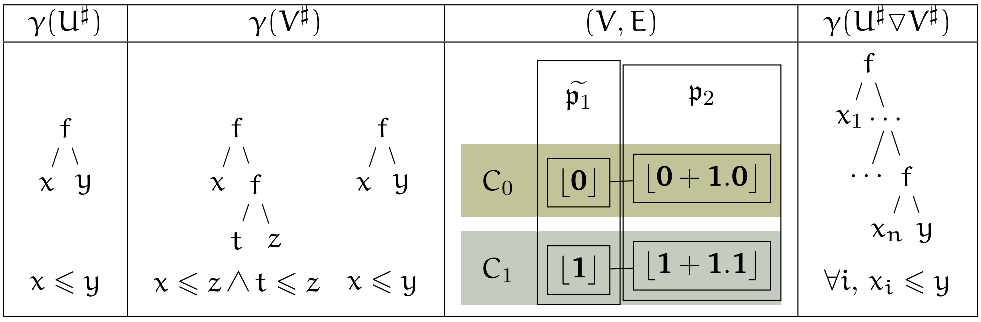

$\gamma(U^\sharp)$
$\gamma(V^\sharp)$
(V, E)
$\gamma(U^\sharp \triangledown V^\sharp)$
f
x y
$x \leqslant y$
f
x f
t z
$x \leqslant z \wedge t \leqslant z$
f
x y
$x \leqslant y$
$\widetilde{\mathfrak{p}_1}$
$\mathfrak{p}_2$
C_0
$\lfloor \mathbf{0} \rfloor$
$\lfloor \mathbf{0}+\mathbf{1}.\mathbf{0} \rfloor$
C_1
$\lfloor \mathbf{1} \rfloor$
$\lfloor \mathbf{1}+\mathbf{1}.\mathbf{1} \rfloor$
f
$x_1 \cdots$
$\cdots$ f
x_n y
$\forall i,\ x_i \leqslant y$

Figure 7.9: Values from Example 7.19

$\mathbf{1}.\mathbf{0}\rfloor) = \lfloor(\epsilon + \mathbf{1}).\mathbf{0}\rfloor \cup (\lfloor \mathbf{1}^\star.\mathbf{0}\rfloor \cap \lfloor \mathbf{1}^{\geqslant 2}.\mathbf{0}\rfloor) = \lfloor \mathbf{1}^\star.\mathbf{0}\rfloor$. Moreover $R_1^{\sharp\star} = \{\lfloor \mathbf{1}^\star.\mathbf{0}\rfloor \leqslant \lfloor \mathbf{1}\rfloor\}$ and $R_2^{\sharp\star} = \{\lfloor \mathbf{1}^\star.\mathbf{0}\rfloor \leqslant \lfloor \mathbf{1}+\mathbf{1}.\mathbf{1}\rfloor$.

- For the second loop iteration, considering component $C_1 = \{\lfloor \mathbf{1}\rfloor, \lfloor \mathbf{1}+\mathbf{1}.\mathbf{1}\rfloor\}$ we get: $r = \lfloor \mathbf{1}.\mathbf{1}.\mathbf{1}^\star.\mathbf{1}\rfloor$, $\mathfrak{a} = \lfloor \mathbf{1}\rfloor$, $\mathfrak{b} = \lfloor \mathbf{1}+\mathbf{1}.\mathbf{1}\rfloor$, $\mathfrak{p} = \lfloor \mathbf{1}^\star.\mathbf{1}\rfloor$. Moreover $R_1^{\sharp\star} = \{\lfloor \mathbf{1}^\star.\mathbf{0}\rfloor \leqslant \lfloor \mathbf{1}^\star.\mathbf{1}\rfloor$ and $R_2^{\sharp\star} = \{\lfloor \mathbf{1}^\star.\mathbf{0}\rfloor \leqslant \lfloor \mathbf{1}^\star.\mathbf{1}\rfloor$.

The loop computation therefore yields $\mathfrak{p} = \{\lfloor \mathbf{1}^\star.\mathbf{0}\rfloor, \lfloor \mathbf{1}^\star.\mathbf{1}\rfloor\}$, $R_1^\sharp = R_2^\sharp = \{\lfloor \mathbf{1}^\star.\mathbf{0}\rfloor \leqslant \lfloor \mathbf{1}^\star.\mathbf{1}\rfloor\}$. Hence: $U^\sharp \triangledown V^\sharp = \langle \lfloor \mathbf{1}^\star.(\mathbf{0}+\mathbf{1})\rfloor, \{\lfloor \mathbf{1}^\star.\mathbf{0}\rfloor, \lfloor \mathbf{1}^\star.\mathbf{1}\rfloor\}, \{\lfloor \mathbf{1}^\star.\mathbf{0}\rfloor \leqslant \lfloor \mathbf{1}^\star.\mathbf{1}\rfloor\}\rangle$

Example 7.20. Consider now a modified version of the previous example: $\mathcal{R} = \{f(2)\}$. Figure 7.10 depicts some of the values mentioned in this example:

$$U^\sharp = \langle \lfloor \mathbf{1}^{\leqslant 2}.(\mathbf{0}+\mathbf{1})\rfloor, \left\{ \begin{array}{l} \lfloor \mathbf{0}\rfloor, \lfloor \mathbf{1}\rfloor, \\ \lfloor \mathbf{1}.\mathbf{1}\rfloor, \lfloor \mathbf{1}.\mathbf{1}.\mathbf{1}\rfloor, \\ \lfloor \mathbf{1}.(\mathbf{0}+\mathbf{1}.\mathbf{0})\rfloor \end{array} \right\}, \left\{ \begin{array}{l} \lfloor \mathbf{0}\rfloor = \lfloor \mathbf{1}\rfloor = \lfloor \mathbf{1}.\mathbf{1}\rfloor, \\ \lfloor \mathbf{1}.\mathbf{1}\rfloor = \lfloor \mathbf{1}.\mathbf{1}.\mathbf{1}\rfloor \\ \lfloor \mathbf{1}.\mathbf{1}.\mathbf{1}\rfloor = \lfloor \mathbf{1}.(\mathbf{0}+\mathbf{1}.\mathbf{0})\rfloor \end{array} \right\}\rangle$$

$$V^\sharp = \langle \lfloor \mathbf{1}^{\leqslant 2}.(\mathbf{0}+\mathbf{1})\rfloor, \left\{ \begin{array}{l} \lfloor \mathbf{1}^{\leqslant 2}.\mathbf{0}\rfloor, \\ \lfloor (\mathbf{1}+\mathbf{1}.\mathbf{1}).\mathbf{1}\rfloor \end{array} \right\}, \{\lfloor \mathbf{1}^{\leqslant 2}.\mathbf{0}\rfloor \leqslant \lfloor (\mathbf{1}+\mathbf{1}.\mathbf{1}).\mathbf{1}\rfloor\}\rangle$$

We do not detail loop computations, which yield $\mathfrak{p} = \{\lfloor \mathbf{1}^{\leqslant 2}.\mathbf{0}\rfloor, \lfloor (\mathbf{1}+\mathbf{1}.\mathbf{1}).\mathbf{1}\rfloor\}$, $R_1^\sharp = \{\lfloor \mathbf{1}^{\leqslant 2}.\mathbf{0}\rfloor = \lfloor \mathbf{1}\rfloor = \lfloor (\mathbf{1}+\mathbf{1}.\mathbf{1}).\mathbf{1}\rfloor\}$ and $R_2^\sharp = \{\lfloor \mathbf{1}^{\leqslant 2}.\mathbf{0}\rfloor \leqslant \lfloor (\mathbf{1}+\mathbf{1}.\mathbf{1}).\mathbf{1}\rfloor\}$. Hence: $U^\sharp \triangledown V^\sharp = V^\sharp$. We see here that: as supports were not extended, neither were the partitions. The widening only tried to salvage numerical constraints while preventing a growth of the number of partitions.

Example 7.21 (Numerical example)**.** Consider the simple example where: $\mathcal{R} = \{f(2)\}$, $U^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1}\rfloor, \{\lfloor \mathbf{0}\rfloor, \lfloor \mathbf{1}\rfloor\}, \{\lfloor \mathbf{1}\rfloor = \lfloor \mathbf{0}\rfloor\}\rangle$ and $V^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1}\rfloor, \{\lfloor \mathbf{0}\rfloor, \lfloor \mathbf{1}\rfloor\}, \{\lfloor \mathbf{1}\rfloor \geqslant \lfloor \mathbf{0}\rfloor, \lfloor \mathbf{1}\rfloor \leqslant \lfloor \mathbf{0}\rfloor + 1\}\rangle$. $U^\sharp$ and $V^\sharp$ have the same shape, therefore widening will be performed on the numerical component of the abstraction, therefore: $U^\sharp \triangledown V^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1}\rfloor, \{\lfloor \mathbf{0}\rfloor, \lfloor \mathbf{1}\rfloor\}, \{\lfloor \mathbf{1}\rfloor \geqslant \lfloor \mathbf{0}\rfloor\}\rangle$

Reducing dimensionality and improving precision

As emphasized by the previous examples, definitions and illustrations, the numerical component of an abstract state is used as a container for constraints on regular expressions, every node in a regular expression must then satisfy all numerical constraints on the underlying regular expression. Therefore when two nodes of a tree satisfy the same constraints, they should be stored in the same partition so as to reduce the dimension of the numerical domain (thus improving efficiency). Moreover the widening operator provided in Figure 7.10 relies (for precision) on the fact that partitions are built by similarity of constraints, therefore partition merging, when it does not result in an over-approximation, also leads to a precision gain. The unification operator defined in Figure 7.7 tends to split partitions whereas the widening operator defined in Algorithm 7.10 tends to merge them. In order to reduce dimensionality (for example, after a

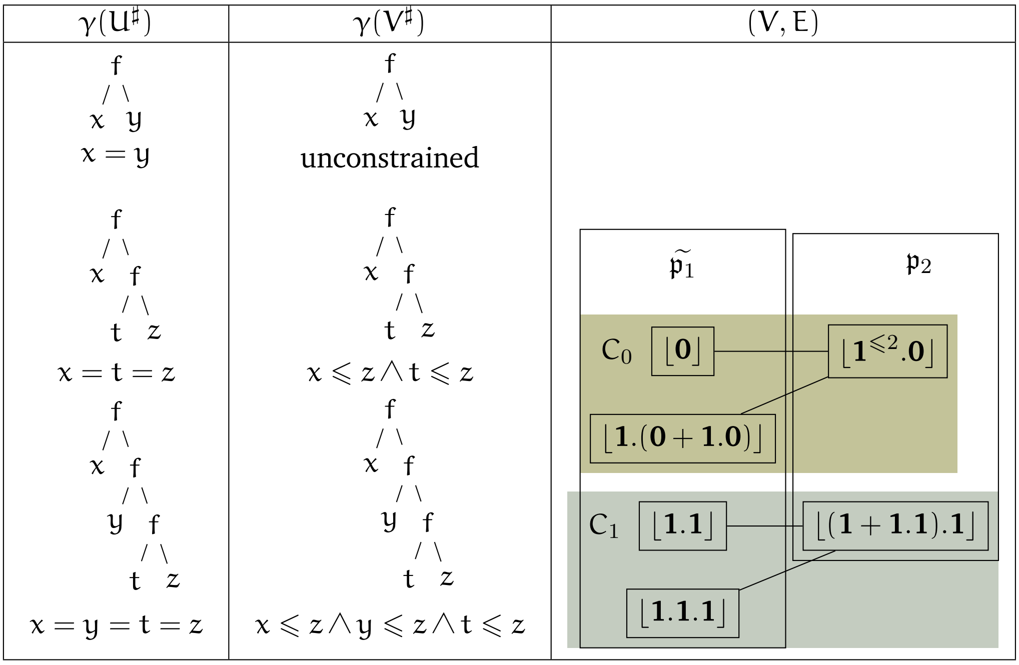

$\gamma(U^\sharp)$
$\gamma(V^\sharp)$
(V, E)
$x = y$
unconstrained
$x = t = z$
$x \leqslant z \wedge t \leqslant z$
$x = y = t = z$
$x \leqslant z \wedge y \leqslant z \wedge t \leqslant z$
$\widetilde{p_1}$
p_2
C_0
$\lfloor \mathbf{0} \rfloor$
$\lfloor \mathbf{1}^{\leqslant 2}.\mathbf{0} \rfloor$
$\lfloor \mathbf{1}.(\mathbf{0}+\mathbf{1}.\mathbf{0}) \rfloor$
C_1
$\lfloor \mathbf{1}.\mathbf{1} \rfloor$
$\lfloor (\mathbf{1}+\mathbf{1}.\mathbf{1}).\mathbf{1} \rfloor$
$\lfloor \mathbf{1}.\mathbf{1}.\mathbf{1} \rfloor$

Figure 7.10: Values from Example 7.20

join computation that greatly increased the number of partitions), we would like to define a **reduce** : $\mathcal{C}^\sharp(\mathcal{R}) \to \mathcal{C}^\sharp(\mathcal{R})$ operator, that folds variables with similar constraints into one. Please note that for all S, S' and x such that $S \cap S' \subseteq \{x\}$, $x \in S$ and for all $R^\sharp \in N_S$, we have that $R^\sharp \sqsubseteq_{N_S}$ **expand**(**fold**$(R^\sharp, x, S'), x, S')$. This means that when variables are folded into one, expanding them afterwards would yield a bigger abstract element. For example, consider the octagon $R^\sharp = \{x \geqslant 2, y \geqslant 2, x = y\}$ then **fold**$(R^\sharp, z, \{x, y\}) = \{z \geqslant 2\}(\triangleq R^{\sharp\prime})$ and **expand**$(R^{\sharp\prime}, z, \{x, y\}) = \{x \geqslant 2, y \geqslant 2\}$. However if we consider $R^\sharp = \{x \geqslant 2, y \geqslant 2\}$ then **fold**(**expand**$(R^\sharp, z, \{x, y\}), z, \{x, y\}) = R^\sharp$. Therefore if we assume given a score function **score**$(R^\sharp, x, S')$ ranging in $[0, 1]$ such that **score**$(R^\sharp, x, S') = 1 \Leftrightarrow R^\sharp =$ **expand**(**fold**$(R^\sharp, x, S'), x, S')$, we are able to define a generic **reduce** operator parameterized by a value α (see Algorithm 7.11). This **reduce** operator merges partitions until no more set of partitions has a high enough score according to the **score** function. Finding a good **score** function is a work in progress. As a first approximation we used the following trivial one: **score**$_0(R^\sharp, S) = 1$ when **expand**(**fold**$(R^\sharp, x, S), x, S) = R^\sharp$ and 0 otherwise. This **score**$_0$ guarantees there is no loss of precision, but can miss opportunities for simplification.

Algorithm 7.11: reduce operator

```
Input  : $X^\sharp$ an abstract element
Output: $Z^\sharp$ an over-approximation of $X^\sharp$ with fewer dimensions
$\langle s, p, R^\sharp \rangle \leftarrow X^\sharp$;
while $\exists \mathcal{S} \subseteq p$, score$(R^\sharp, \mathcal{S}) \geqslant \alpha$ do
    $\mathcal{S}' \leftarrow \text{argmax}_{\mathcal{S}' \subseteq p}$score$(R^\sharp, \mathcal{S}')$;
    $e \leftarrow \cup_{x \in \mathcal{S}'} x$;
    $\langle s, p, R^\sharp \rangle \leftarrow \langle s, (p \setminus S') \cup \{e\},$ fold$(R^\sharp, e, S')$;
end
return $\langle s, p, R^\sharp \rangle$;
```

Example 7.22. Consider the following example: $U^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1} \rfloor, \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1} \rfloor\}, \{\lfloor \mathbf{0} \rfloor = 0, \lfloor \mathbf{1} \rfloor = 0\} \rangle$. Relations on $\lfloor \mathbf{0} \rfloor$ and $\lfloor \mathbf{1} \rfloor$ can be expressed in one relation using the summarizing variable $\lfloor \mathbf{0}+\mathbf{1} \rfloor$. This yields: **reduce**$(U^\sharp) = \langle \lfloor \mathbf{0}+\mathbf{1} \rfloor, \{\lfloor \mathbf{0}+\mathbf{1} \rfloor\}, \{\lfloor \mathbf{0}+\mathbf{1} \rfloor = 0\} \rangle$. Note that **expand**$(\{\lfloor \mathbf{0}+\mathbf{1} \rfloor = 0\}, \lfloor \mathbf{0}+\mathbf{1} \rfloor, \{\lfloor \mathbf{1} \rfloor, \lfloor \mathbf{0} \rfloor\}) = \{\lfloor \mathbf{0} \rfloor = 0, \lfloor \mathbf{1} \rfloor = 0\}$. Therefore no information is lost.

7.6.2 Abstract semantics of operators

As for tree automata, abstract semantics of operators defined in Section 7.2 can be defined as simple transformations on regular automata. Indeed the `make_symbolic(s` $\in \mathcal{R}$`)` (resp. `get_son`) operator, amounts to adding (resp. removing) an integer letter to: (1) the partitions in the subpartitioning and (2) the support. `make_integer(e` $\in$ *expr*`)` amounts to building an abstract element with support $\lfloor \epsilon \rfloor$ and a subpartitioning containing only $\{\lfloor \epsilon \rfloor\}$, on which we put the constraint that it is equal to e. `is_symbol` needs only split the support and each partition, in the two language $L = \{\epsilon\}$ and $A_n^\star \setminus L$. Indeed in order to restrict to terms having only an integer as root, the support must be reduced to ϵ. The `get_sym_head` operator always yields the whole ranked alphabet (as this was abstracted away and will be refined by the automaton abstraction). Finally for `get_num_head`: (1) if the empty path $\wr\int$ is in the support we produce the set of integers satisfying the numerical constraints on the partition containing ϵ, and $\top$ in case no such partition could be found, and (2) otherwise we know that no numerical value is produced.

Algorithms 7.12 to 7.17 respectively provide the definition of transformers **make_symbolic**, **make_integer**, **get_son**, **is_symbol**, **get_sym_head** and **get_num_head**. These algorithms use the following notation: when r is a regular expression over $\Sigma^\star$ and $a \in \Sigma$, $\partial_a(r) = \{w \mid a.w \in r\}$ and $\int_a(r) = \{a.w \mid w \in r\}$.

As for the value abstraction of Section 7.5, these operators are defined only on values, and are then lifted to an environment abstraction. We will not redefine here this lifting as it follows the same construction as that of Section 7.5.3.

Algorithm 7.12: make_symbolic

```
Input : $s \in \mathcal{F}_n, \langle s_1, \mathfrak{p}_1, R_1^\sharp\rangle, \dots \langle s_n, \mathfrak{p}_n, R_n^\sharp\rangle$
$s \leftarrow \int_{\lfloor \mathbf{0} \rfloor} s_1 \cup \cdots \cup \int_{\lfloor \mathbf{n\text{-}1} \rfloor} s_n$;
$(\mathfrak{p}_i')_{i \leqslant n} \leftarrow (\bigcup_{r \in \mathfrak{p}_i} \int_{\lfloor \mathbf{i\text{-}1} \rfloor} r)$;
$(\phi_i)_{i \leqslant n} \leftarrow (\biguplus_{r \in \mathfrak{p}_i} (r \mapsto \int_{\lfloor \mathbf{i\text{-}1} \rfloor} r)$;
$(R_i^{\sharp\prime})_{i \leqslant n} \leftarrow R_i^\sharp[\phi_i]$;
return $\langle s, \cup_{i \leqslant n} \mathfrak{p}_i', \bigotimes_{i \leqslant n} R_i^{\sharp\prime}\rangle$
```

Algorithm 7.13: make_integer

```
Input : $v \in \mathbb{Z}$
return $\langle \lfloor \epsilon \rfloor, \{\lfloor \epsilon \rfloor\}, S^\sharp[\![\mathtt{Assume}(\lfloor \epsilon \rfloor = v)]\!] \circ S^\sharp[\![\mathtt{add}(\lfloor \epsilon \rfloor)]\!](\top)$;
```

Algorithm 7.14: get_son

```
Input : i an integer, $\langle s, \mathfrak{p}, R^\sharp\rangle$
$s' \leftarrow \partial_i(s)$;
$\mathfrak{p}' \leftarrow [\bigcup_{r \in \mathfrak{p}} \partial_i(r)]_\emptyset$;
$\phi \leftarrow \lambda_{|\mathfrak{p}} r.$ if $\partial_i(r) \neq \emptyset$ then $\partial_i(r)$ else undefined;
$R^{\sharp\prime} \leftarrow R^\sharp[\phi]$;
return $\langle s', \mathfrak{p}', R^{\sharp\prime}_{|\mathfrak{p}'}\rangle$;
```

Algorithm 7.15: is_symbol

```
Input : $\langle s, p, R^\sharp \rangle$
if $\epsilon \in s$ then
    return $\{\mathtt{true}, \mathtt{false}\}$;
else
    return $\{\mathtt{true}\}$;
```

Algorithm 7.16: get_sym_head

```
Input : $\langle s, p, R^\sharp \rangle$
return $\mathcal{R}$;
```

Algorithm 7.17: get_num_head

```
Input : $\langle s, p, R^\sharp \rangle$
if $\epsilon \in s$ then
    if $\exists a \in p,\ \epsilon \in a$ then
        return $a$
    else
        return $\top$
else
    return $\bot$
```

7.7 Building up the full abstraction

7.7.1 Product

The abstraction by tree automata defined in Section 7.5 and the abstraction by numerical constraints on tree paths defined in Section 7.6 provide non comparable information on the set of terms they abstract. Indeed the former describes precisely the shape of the term but can not express numerical constraints whereas the latter abstracts away most of the shape and focuses on numerical constraints. Therefore, to benefit from both kinds of information, we use a product between the two domains. Both abstractions in the product contain information on potential integer positions. Indeed the position of the □ symbol in the tree automaton abstraction and the support in the numerical constraints abstractions yield this information. Therefore we remove the support component from the product as the information can be retrieved from the tree abstraction. The definitions of the abstract operators in Section 7.6 require the support to be a regular language.

We show in the following subsection how to retrieve the support of a tree automaton with holes and that it is regular.

7.7.2 Removing redundant information

Given a Fta $(Q, \mathcal{R}, Q_f, \delta)$ over a ranked alphabet $\mathcal{R}$ with maximum arity n. We assume that every state in Q is reachable. Consider the following system over variables ν_p for $p \in Q$ with values in the set of languages over the alphabet A_n (. designates the classical concatenation operator lifted to languages) :

$$\{\nu_p = \bigcup_{(s,(q_1,\dots,q_m),q)\in\delta \mid q_i = p} \nu_q.\{i\} \cup \begin{cases} \{\epsilon\} & \text{if } p \in Q_f \\ \emptyset & \text{otherwise} \end{cases} \mid p \in Q\}$$

Every language $\{i\}$ for $i \in \mathbb{N}$ is regular and does not contain ϵ, moreover $\emptyset$ and $\{\epsilon\}$ are regular languages. Therefore by application of the results recalled in Section 7.3.1 we can compute the unique solution of this system, moreover every ν_p is regular. Variable ν_p is defined so that: $w \in \nu_p$ if and only if there exists a tree t recognized by the automaton such that $p \in \textsc{reach}(t_{|w})$. Therefore if $\Box \in \mathcal{R}$ we have that the regular language: $\cup_{(\Box,(),p)\in\delta}\nu_p$ represents exactly the potential positions of integers in trees accepted by the tree automaton. In the following, whenever $\mathcal{A}$ is a Dfta we denote as $\textbf{hole}(\mathcal{A})$ the regular expression encoding the potential position of holes in the trees recognized by $\mathcal{A}$.

7.7.3 Factoring away the numerical component

We have shown in Section 7.5.3 how the tree automaton value abstraction was lifted to an environment abstraction. Performing this similar lifting on the numerical constraints abstraction and building a product of both abstractions (while removing the support from the second abstraction, as we have shown it to be redundant) would yield an abstract tree environment of the form:

$$F^\sharp = \mathcal{T} \to \{\langle \mathcal{A}, \mathfrak{p}, R^\sharp \rangle \mid \mathcal{A} \in \textsc{Dfta}, \mathfrak{p} \in P(\textbf{hole}(\mathcal{A})), R^\sharp \in \mathfrak{M}^\sharp_\mathfrak{p})\}$$

The obvious drawback of this abstraction is the fact that it features one numerical environment for each term in the map. Moreover please note that in the complete abstraction, containing an abstract numerical environment and an abstract tree environment, the numerical variables from the program will be stored in a different abstract environment from the dimensions allocated for numerical constraints on tree leaves. Such an abstraction would not be able to represent numerical relations between program variables and leaves of trees or between leaves of two different trees. For these reasons, we factor away every abstract numerical environment and the abstract numerical environment dedicated to the handling of program variables in one abstract numerical environment. The resulting abstraction is therefore an abstract numerical environment $E^\sharp$, containing program variables and dimensions used for the representation of constraints on tree leaves and an abstract tree environment containing a tree automaton and a partitioning of its hole position denoted $F^\sharp$. Note that, the numerical environment might be asked to store dimensions on two identical regular expressions from two different trees, we will precede regular expression variables in the numerical domain by the name of the tree to which it refers.

Example 7.23. As an example, consider the following tree abstraction: $(t \mapsto (\mathcal{A}, \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1} \rfloor\}),$ $u \mapsto (\mathcal{A}, \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1} \rfloor\}))$, where $\mathcal{A}$ is the tree automaton recognizing precisely the tree set $\{f(\Box, \Box),$ $g(\Box, \Box)\}$, adjoined with the numerical environment: $\{t.\lfloor \mathbf{0} \rfloor = u.\lfloor \mathbf{1} \rfloor, t.\lfloor \mathbf{1} \rfloor = u.\lfloor \mathbf{0} \rfloor, i = u.\lfloor \mathbf{0} \rfloor\}$. This represents the set of concrete environments $\{((t \mapsto \mathfrak{s}(\alpha, \beta), u \mapsto \mathfrak{s}'(\beta, \alpha)), (i \mapsto \beta)) \mid (\alpha,$ $\beta) \in \mathbb{I}^2 \wedge (\mathfrak{s}, \mathfrak{s}') \in \{f, g\}^2\}$. This example emphasizes how the abstraction allows to express numerical relations between different tree leaves values and program variables.

7.7.4 Functional evaluation of tree expressions

When asked to perform the evaluation of an expression, the analyzer does not return a value, but a partial value (containing the tree automaton and the subpartitioning) and a numerical environment. Indeed consider the minimal example where we have the following numerical abstract environment $E^\sharp = \{0 \leqslant x \leqslant 10\}$, and the tree environment is $F^\sharp = \emptyset$. Assume moreover that we want to evaluate the expression: `make_integer(x)`. Instead of returning the value $\langle \mathcal{A}, \{\lfloor \epsilon \rfloor\}, \{0 \leqslant \lfloor \epsilon \rfloor \leqslant 10\} \rangle$ where $\mathcal{A}$ is a Dfta accepting exactly the language $\Box$, we return the partial value $\langle \mathcal{A}, \{\lfloor \epsilon \rfloor\} \rangle$ and the modified abstract numerical environment $\{0 \leqslant x \leqslant 10, x = \lfloor \epsilon \rfloor\}$. This allows the result of the evaluation of tree expressions to be relational with other dimensions in the numerical environment. However this transformation requires the abstract interpreter to return the abstract environment with the expected abstract value. This is the case for the Mopsa framework as was presented in Chapter 3. We do not provide here a full mathematical definition of this transformation as it mostly consists of variables manipulations and renamings.

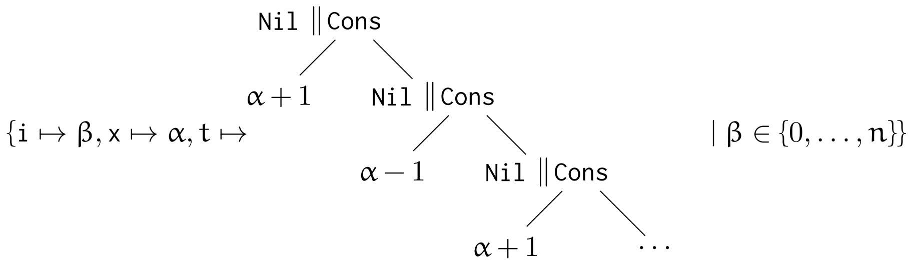


Figure 7.11: Concretization of OCaml example

Example 7.24. In this example we will only consider tree automata recognizing one tree, they will therefore be assimilated as the only tree they recognize. Consider the following abstract environment: $U^\sharp = (\mathsf{t} \mapsto (\mathsf{f}(\Box,\Box),\{\lfloor\mathbf{0}\rfloor,\lfloor\mathbf{1}\rfloor\})(\triangleq F^\sharp),\{\mathsf{t}.\lfloor\mathbf{0}\rfloor = \mathsf{t}.\lfloor\mathbf{1}\rfloor + 1\})$. Consider moreover the expression: $e \triangleq$ `make_symbolic("f", {t, make_integer(i)})`. The successive evaluations and their results are the following:

$$\begin{array}{l}
\mathbb{E}^\sharp[\![e]\!](U^\sharp)\\
\quad \mathbb{E}^\sharp[\![\mathtt{make\_integer(i)}]\!](U^\sharp)\\
\quad\quad \mathbb{E}^\sharp[\![\mathtt{i}]\!](U^\sharp)\\
\quad\quad \hookrightarrow \mathtt{i}\\
\quad \hookrightarrow (\Box,\{\lfloor\epsilon\rfloor\}),\{\mathsf{t}.\lfloor\mathbf{0}\rfloor = \mathsf{t}.\lfloor\mathbf{1}\rfloor + 1, \lfloor\epsilon\rfloor = \mathtt{i}\}(\triangleq R^\sharp)\\
\quad \mathbb{E}^\sharp[\![\mathtt{t}]\!]((F^\sharp, R^\sharp))\\
\quad \hookrightarrow (\mathsf{t} \mapsto (\mathsf{f}(\Box,\Box),\{\lfloor\mathbf{0}\rfloor,\lfloor\mathbf{1}\rfloor\}),\{\mathsf{t}.\lfloor\mathbf{0}\rfloor = \mathsf{t}.\lfloor\mathbf{1}\rfloor + 1, \lfloor\epsilon\rfloor = \mathtt{i}, \lfloor\mathbf{0}\rfloor = \mathsf{t}.\lfloor\mathbf{0}\rfloor, \lfloor\mathbf{1}\rfloor = \mathsf{t}.\lfloor\mathbf{1}\rfloor\}\\
\hookrightarrow (\mathsf{t} \mapsto (\mathsf{f}(\mathsf{f}(\Box,\Box),\Box),\{\lfloor\mathbf{0.0}\rfloor,\lfloor\mathbf{0.1}\rfloor,\lfloor\mathbf{1}\rfloor\}),\{\mathsf{t}.\lfloor\mathbf{0}\rfloor = \mathsf{t}.\lfloor\mathbf{1}\rfloor + 1, \lfloor\mathbf{1}\rfloor = \mathtt{i}, \lfloor\mathbf{0.0}\rfloor = \mathsf{t}.\lfloor\mathbf{0}\rfloor,\\
\quad\quad\quad \lfloor\mathbf{0.1}\rfloor = \mathsf{t}.\lfloor\mathbf{1}\rfloor\}
\end{array}$$

This emphasizes how the numerical domain is used to keep relations during the evaluations.

7.8 Implementation and example

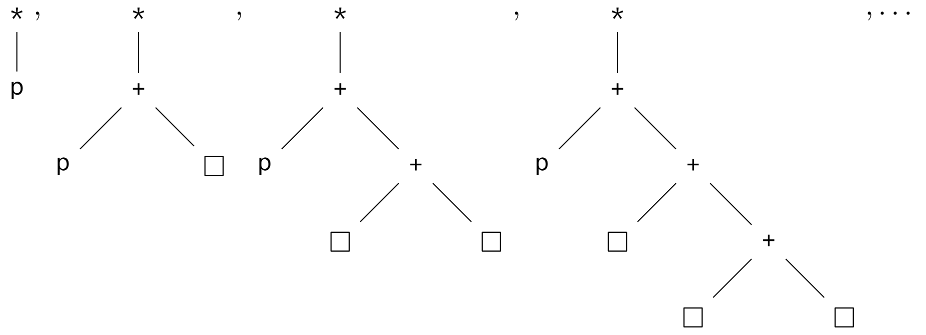
The analyzer presented in this chapter was implemented (in OCaml) in Mopsa, using the (tree) regular language operators mentioned in Section 7.3.

We recall that numerical variables of the form $\mathsf{t}.x$, where t is a numerical term variable, represent a variable allocated for tree t. For example: $\mathsf{t}.r$ where r is a regular expression is the variable allocated for partition r in tree t.

C introductory example. Let us consider the introductory example Program 7.4. The loop invariant inferred with our analysis is the following abstract element: $U^\sharp = (\mathsf{y} \mapsto (\mathcal{A},\{\lfloor\mathbf{0}.(\mathbf{0.0})^\star.\mathbf{1}\rfloor(\triangleq r)\}), R^\sharp)$, with $\mathcal{A} = \langle\{a,b,c,d\},\{*(1),+(2),\Box(0),(p,0)\},\{c\},\{*(d) \to c, +(c,a) \to d, \Box() \to a, p \to c\}\rangle$, and $R^\sharp$ satisfies the constraints: $\{i \geqslant 0, i \leqslant \mathsf{n}, \mathsf{y}.r = 4\}$. This describes precisely the set of terms of the form: p, $*(p+4)$, $*(*(p+4)+4)$, ... As mentioned in Section 7.7.4 evaluations of tree expressions yield pairs containing an expression and an abstract environment. Tree expressions are pairs $(\mathcal{A},\mathfrak{p})$, partitions in $\mathfrak{p}$ are bound by the adjoined environment. Let us now present the result of the evaluation of the `make_integer(4)` expression in the abstract environment $U^\sharp$. We get the expression $(\mathcal{A}',\{\lfloor\epsilon\rfloor\})$ (where $\mathcal{A}'$ recognizes only $\Box$) in the environment: $(\mathsf{y} \mapsto (\mathcal{A},\{r\}), R^{\sharp\prime})$ where $R^{\sharp\prime} = R^\sharp \cup \{\lfloor\epsilon\rfloor = 4\}$. This example emphasizes how the environment is used to give constraints on the adjoined expression. This yields the ability to transport numerical relations from the leafs of the expression up to the assigned variable t.

OCaml introductory example. Let us now consider the introductory example Program 7.5. The inferred loop invariant is the following ($r = \lfloor(\mathbf{1}.\mathbf{1})^\star.\mathbf{0}\rfloor$ and $r' = \lfloor(\mathbf{1}.\mathbf{1})^\star.\mathbf{1}.\mathbf{0}\rfloor$): $(\mathtt{t} \mapsto (\mathcal{A}, \{r, r'\}), R^\sharp)$ and $R^\sharp$ satisfies the constraints: $\{t.r' = \mathtt{x} - 1, t.r = t.r' + 2, i \geqslant 0, i \leqslant \mathtt{n}\}$ and $\mathcal{A} = (\{a, b, c, d\}, \{\mathtt{Cons}(2), \mathtt{Nil}(0), \square(0)\}, \{a\}, \{\mathtt{Cons}(c, a) \to d, \mathtt{Cons}(c, d) \to a, \mathtt{Nil} \to a, \square \to c\})$. The concretization is illustrated in Figure 7.11. Please note that at the end of the `while` loops the two numerical environments that need to be joined are not defined over the same set of variables (in the environments that have not gone through the loop, variables $\mathtt{t}.r'$ and $\mathtt{t}.r$ are not present). However thanks to the $\uplus$ operator, we do not have to loose the numerical relations between these variables and $\mathtt{x}$. Hence we are able to prove that the assertion holds.

String copy. In addition to the two previous examples taken from the introductory remarks, let us consider the `strcpy` function from Chapter 4 (Program 4.1). We want to discover the memory zones used by this function, which will enable us to improve the framing mechanism for the modular analysis. Consider the following abstract element $S^\sharp = (\mathtt{p} \mapsto \mathcal{A}, \mathtt{q} \mapsto \mathcal{A}), \{\lfloor \mathbf{0}.\mathbf{1}^+.(\epsilon + \mathbf{0})\rfloor = 1\}$ where $\mathcal{A}$ is the tree automaton accepting the following tree regular language:



The values of p and q in the environments represented by $S^\sharp$ represents an over-approximation of the memory regions that will indeed be read/write during the execution of `strcpy`. Please note moreover that this abstract state can be inferred by performing the analysis of a tree manipulating program, obtained by a hand-made transformation of the original `strcpy` program. It is our intention to develop in future works an automatic pre-analysis discovering $S^\sharp$ that does not require such a hand-made transformation.

7.9 Conclusion

We presented a relational abstract environment for sets of trees over a finite algebra, with numerically labeled leaves. We emphasized the potential applications of being able to describe such trees: description of reachable memory zones, tracking symbolic equalities between program variables, description of tree like structures. All domains presented here have been implemented as a library in the Mopsa framework.

The expressiveness of the abstraction was underlined on the examples from our introductory remarks. Even though these examples target a language manipulating trees, the tree abstraction should be used as a tool for the definition of higher level abstractions. The most direct of these applications being: (1) the generalization of symbolic propagation ([Min06b]) to handle merging points and loops; (2) the design of a pre-analyzer for the C language.

Chapter 8

Conclusion

8.1 Summary of the thesis

The problem of inferring and proving sound, precise, and reusable contracts for a statement is a hard one. Contracts need to represent relations between abstract states (input/output relations), therefore our goal was to define a structure that could be interpreted as such. We have shown that a finite list of pairs of abstract states can be used to describe such contracts but fails to express some relationality between input and output states. We improved such contracts to win back relationality, this yielded the modular analyzer of Chapter 5. Due to the lack of experimentation on real life C code projects, we can not that say we provided a fully functional scalable modular analyzer. However we feel that the combination of results from the literature (I/O pairs mixed with relational domains) as well as the idea of filter generalization by use of widenings are a good starting point for the design of a modular analyzer as underlined by our preliminary results, able to tackle an almost complete version of the C language. In order to emphasize the fact that our modular analysis was not purely numerical, we focused on handling the most common data structure from the C language: strings.

We initially intended to reuse an abstraction from the literature, however we found none that would work nicely with the Cell abstraction that we chose for handling all other C features. Some of the classical string abstractions from related works intertwined the handling of string manipulations with the handling of pointer manipulations in one abstract domain. In our process of designing a modular[1] analyzer in the Mopsa project we decided to keep the abstractions handling these two distinct features of C separated. Our goal was to provide a string abstraction that could be optionally added in a reduced product with the Cell domain, already handling the C memory representation. We reused ideas from existing abstractions for the definition of the string abstraction of Chapter 4, namely tracking the length and the allocated size of strings with a numerical variable. We used the opportunity of the definition of this domain to provide a mechanism allowing a dynamic translation from the Cell domain to the String domain.

The reusability of the contracts inferred by a modular analyzer is mostly dependant on the quality of the framing mechanism. This framing should not only remove, from the input context, values that are not used by the function, but also provide information as to which variables should be generalized for better reusability. We therefore decided to represent the set of reachable memory zones as a set of terms over the C pointer manipulating expression languages.

This led to the development of the tree abstraction of Chapter 7. Most of the work from the literature focused on sets of trees containing constants whereas we wanted to be able to store any numerical values in the leaves of our C pointer manipulating expressions. Moreover we felt that targeting simple patterns of pointer manipulations would be precise enough for our goal of designing a pre-analysis. For this reason we used tree regular languages to represent the set of potential shapes of our C expressions. Works from the literature provided a widening operator

[1] Modular is to be understood, here, in the sense that each abstract domains handles one feature of the language.

on regular languages that could be directly translated to tree regular languages. In the process of adding numerical values at the leaf positions of the set of trees, we had to modify the semantics of the underlying numerical domain, so that it would account for potentially absent variables.

As the problem had been encountered beforehand in Mopsa (during the analysis of Python programs where variables could be both numerical and non-numerical), we decided to propose a general solution to both problems by defining a lifting of the semantics of classical numerical abstract domains to an heterogeneous framework. As this problem arose at two different occasions and on two different subjects, and as people have been using a modified version of the interval abstract domain (the semantics of this modified version being unclear) to account for absent variables, we are confident that this relational lifting has use cases. Moreover we have shown that this lifting does not induce an overhead cost when there are no optional variables. Finally, as we felt that there was a gap in the precision/cost trade-off between the lifting we provided and partitioning, we designed a partitioning extension of the Clip abstraction defined in Chapter 6.

Finally I was part of the Mopsa project (presented in Chapter 3). Mopsa was doubly present in this thesis: firstly I participated in the development of this static analyzer, secondly Mopsa provided a framework for the development of all the abstract domains presented here. The variety of static analyses involved in this thesis and the diversity of features they required from the Mopsa framework provided tests of its modularity and of its expressiveness in terms of what analyses could be implemented.

8.2 Concluding example

The different chapters of this thesis provided results on various topics, some seemingly not related to the inference of contracts: strings, heterogeneous numerical environments, trees. In the previous section we provided a roadmap of the thesis, explaining how these different topics came into focus. In this section we give a small example showing how we intend, in future works, to bring together all these different parts. Program 8.1 defines a `strings` structure containing three string pointers, a `strcpy` function as was already shown in previous chapters, a `repeat` function taking a `strings` structure as argument and copying the second string in the first one. This program was handcrafted based on behaviors actually encountered in real life C code: the redefinition of `strcpy` as in Qmail, the usage of structures containing several strings as in the `web2c` type `key_entry`, and calls to `strcpy` on such strings, as in [AGH06]. Assume that we want to infer contracts for the `repeat` function.

- We start by performing a pre-analysis of the body of function `strcpy` using the tree abstraction from Chapter 7. This pre-analysis yields the information that `strcpy` only reads and writes in the memory zones `*p`, `*(p+1)`, `*(p+1+1)`, ..., `*q`, `*(q+1)`, `*(q+1+1)`, ... Even though the only numerical value found in this set of terms is 1, it is not here considered as a term constant but as a numerical value. For this reason, at line 8, when joining the two facts that `*p` and `*(p+1)` are both used by `strcpy`, we need to join two sets of numerical environments that do not share the same definition set. Indeed `*p` does not contain any numerical variable, whereas `*(p+1)` and `*(p+1+1)` contain a numerical variable with a relation setting it to 1. Joining these facts requires the use of the **strenghtening** operator of Chapter 6. Using this information we get that the `repeat` function only reads and writes on the memory zones `*s`, `*(s+8)`, `*(s.a)`, `*(s.a+1)`, `*(s.a+1+1)`, ..., `*(s.b)`, `*(s.b+1)`, `*(s.b+1+1)`, ...
- We then perform a top-down analysis of the program starting with the `main` function, this analysis is made using the string abstraction of Chapter 4. Values of (1) numerical variables `x` and `y`, (2) pointers `s`, `s.a`, `s.b` and `s.c` are all tracked by the Cell abstraction. Lengths of strings `a`, `b` and `c` are stored in the numerical domain by the string abstraction. At line 28, when encountering the first call to `repeat`, we evaluate the expressions discovered by the pre-analysis in the input environment to guide us as to which memory zones should be

```
typedef struct strings {
  char* a;
  char* b;
  char* c;
} strings;

void strcpy(char* p, char* q) {
  while (*q != '\0') {
    *p = *q;
    p ++;
    q ++;
  }
  *p = *q;
}

void repeat(strings s) {
  strcpy(s.a, s.b);
}

int  x = 23;
char a[10] = "first";
char b[10] = "second";
char c[10] = "third";
strings s = {a, b, c};

int main() {
  int y = 10;
  repeat(s);
}
```

Program 8.1: Complete example

kept and which should be universally quantified and removed from the environment for the modular analysis of `repeat`. Values of `x`, `y`, `s.c`, and `c` are forgotten. We then perform a modular analysis of the body of `repeat` and `strcpy` under input hypotheses. Using input hypotheses has the advantage that we perform an analysis assuming that `s.a` and `s.b` do not alias, however it has the drawback that the analysis is made under the assumption that the length of `b` is precisely 6. Subsequent calls to `repeat` and the use of widening operations on input states, as shown in Chapter 5, will remove this assumption and generalize the result. After generalization, the contract for function `repeat` will then be: for every inputs, where `s.a` and `s.b` do not alias and are valid pointers, `repeat` only modifies the memory zones pointed to by `s.a` and `s.b`, however the lengths of `s.a` and `s.b` at the end are equal to the length `s.b` in the input. The following provides a simplified illustration, of the form precondition ⇒ filter, of the generalized contract (unmentioned memory zones are not modified, primed variables denote input values):

$$\forall u \neq v,\ \begin{array}{ll} \text{pointers:} & \begin{array}{l} \texttt{s.a} \to u \\ \texttt{s.b} \to v \end{array} \\ \hline \text{offsets:} & \begin{array}{l} o(\texttt{s.a}) = 0 \\ o(\texttt{s.b}) = 0 \end{array} \\ \hline \text{length:} & \begin{array}{l} 0 \leqslant \text{len}(v) < \text{size}(v) \\ 0 < \text{size}(u) \end{array} \end{array} \Rightarrow \begin{array}{ll} \text{pointers:} & \begin{array}{l} \texttt{s.a} \to u \\ \texttt{s.b} \to v \end{array} \\ \hline \text{offsets:} & \begin{array}{l} o(\texttt{s.a}) = 0 \\ o(\texttt{s.b}) = 0 \end{array} \\ \hline \text{length:} & \begin{array}{l} \text{len}(v) < \text{size}(v) \\ \text{len}(v) < \text{size}(u) \\ \text{size}(u') = \text{size}(u) \\ \text{size}(v') = \text{size}(v) \\ \text{len}(v) = \text{len}(u) \\ \text{len}(v) = \text{len}(v') \end{array} \end{array}$$

8.3 Future works

As already emphasized by the concluding remarks from the above chapters, there are several points that are left unfinished, and some unstarted, at the end of this thesis.

Let us start with some future works related to the modular iterator, which was at the heart of this thesis.

Tests and benchmarks. The modular analysis framework has been tested on several test cases from the literature so as to measure its expressiveness and the quality of the inferred summary. Such results were used to guide our work on improving the quality of the framing and input generalization mechanisms, which are crucial in the development of a modular analysis that is not purely numerical. However these preliminary results can not be used as a validation of the quality of our interprocedural iterator. Indeed the primary goal of modular iterators is to increase the overall scalability of the analysis. Therefore tests and benchmarks on real life C projects are necessary for the validation of our method.

Heuristics on summarization points. It is (empirically) known from previous works [JM18] that a modular analysis based on the use of relational domains for the representation of relations is not beneficial when applied everywhere. Indeed the overhead cost induced by the relation computation kills the summarization gain. In a top-down analysis, there is a trade-off between function body inlining and summary discovery/usage. For this reason an important future work is to provide heuristics for the discovery of "good" program points where summarization should be applied. This has already been studied in [JM18], this work was not presented here but was done in parallel to this thesis. There remains to adapt these ideas to the context of the modular analysis of C presented in this thesis.

Framing. Chapter 7 is entirely devoted to the development of a tree domain, the initial purpose of which was to be used as a pre-analysis guiding the framing of the interprocedural iterator. Even though we finished the development of the abstract domain and tested its expressiveness on a tree toy language we have not yet implemented the pre-analyzer using this tree abstract domain and have not connected it to our modular analysis of C programs.

Proj-inclusion as an atomic operator for polyhedra. The proj-inclusion operator of Chapter 6 is at the heart of the definition of the abstract inclusion test and the abstract union operator of the Clip abstraction. In Chapter 6, it is implemented as the composition of several projections and one inclusion. We started working on improving this algorithmic definition in the special case of the polyhedra abstraction. We made the proj-inclusion operator atomic using results from parametric linear programming. This provides two major improvements. Firstly, it avoids computing the whole projected polyhedron when the inclusion does not hold: as soon as a constraint of $\mathcal{P}_{|\mathbf{def}(\mathcal{Q})}$ that does not include $\mathcal{Q}$ is found, the operator can return. Secondly, the whole proj-inclusion operator can be made incremental. It means that any subsequent test $\mathcal{Q} \mathrel{|\sqsubseteq} \mathsf{S}^\sharp[\![\texttt{Assume}(e)]\!](\mathcal{P})$ will build upon the result of $\mathcal{Q} \mathrel{|\sqsubseteq} \mathcal{P}$ to avoid redundant computations. We did not present these results in this thesis but underline as future works the fact that further testing needs to be done to evaluate the efficiency gain of this novel operator.

Fold-expandability. We have seen in Chapter 7 that tree leaves partitionings evolved dynamically during analysis. Indeed partitions are split by join operations and merged by widening operations. In order to have more control over the number of partitions, we defined a **reduce** operator that tries to fold variables together. As of now, variables are folded when they can be unfolded without loss of precision, this is tested in the implementation by actually folding, unfolding and testing the inclusion, which is quite costly. In the special case of polyhedra, we started to study the properties of dimensions that can be folded and expanded without precision loss. Preliminary results (not presented in this thesis) were not sufficient for the development of a better exact algorithm, however it is our hope that further works will provide one. Moreover, please note that it is always sound to fold any set of dimensions together, therefore we feel that a non-exact algorithm based on sampling points in polyhedra could be devised.

Folding and expanding operations are often used to represent potentially unbounded memory locations. It is usually the case that variables that should be folded together, are defined as such by the abstraction. For example in an array abstraction all array cells will be summarized together. We feel that the ability to automatically discover which dimensions should be folded together could:

- in some cases improve the precision of the analysis: if the cell of an array does not have the same value as the others, it is automatically kept separated in the numerical domain;
- in some cases improve the efficiency of the analysis by automatically reducing the dimensionality of the numerical abstraction.

Precise join detection. The efficiency of the $\wp$-Clip abstraction from Chapter 6 depends greatly upon the number of partitions in the abstraction. We have shown that the number of partition can be reduced by merging together pairs of elements of the partition. An element of the partition represents a set of environments, the definition sets of all these environment form a SVI (support variable interval). Knowing whether the union of two SVIs can be precisely represented as a SVI is an easy one. However, as for the partitioning abstraction it is hard to know whether the merging of two monomials from the partitioning induces an over-approximation on the set of environments for each definition sets (see e.g. [BHZ09, RHC18]). Further works on the exact/precise join detection problem would improve the quality of the $\wp$-Clip abstraction in the case where the underlying domain is the polyhedra abstraction.

Term index

Symbol index

Appendix A

Proofs

Proof of 6.1. Let P be in $\mathfrak{M}$ and $m \in \mathcal{P}^\sharp$. We want to prove that: $\alpha_{\mathcal{P}^\sharp}(P) \sqsubseteq_{\mathcal{P}^\sharp} m \Leftrightarrow P \subseteq \gamma_{\mathcal{P}^\sharp}(m)$.

- if $|\{\mathbf{def}(\rho) \mid \rho \in P\}| < \infty$, then $\alpha_{\mathcal{P}^\sharp}(P) = \biguplus_{\mathcal{W} \in \{\mathbf{def}(\rho) \mid \rho \in P\}} \mathcal{W} \mapsto \alpha^{\mathcal{W}}(\{\rho \in P \mid \mathbf{def}(\rho) = \mathcal{W}\})$.
  - if $m = \top$ then we have $\alpha_{\mathcal{P}^\sharp}(P) \sqsubseteq_{\mathcal{P}^\sharp} m$ and we have $P \subseteq \gamma_{\mathcal{P}^\sharp}(m)$
  - otherwise:
    - ⋆ Whenever $\alpha_{\mathcal{P}^\sharp}(P) \sqsubseteq_{\mathcal{P}^\sharp} m$, we have that $\forall \mathcal{W} \in \mathbf{def}(\alpha_{\mathcal{P}^\sharp}(P)), \mathcal{W} \in \mathbf{def}(m) \wedge \alpha_{\mathcal{P}^\sharp}(P)(\mathcal{W}) \sqsubseteq^{\mathcal{W}} m(\mathcal{W})$, therefore: let $\rho \in P$, let $\mathcal{W} = \mathbf{def}(\rho)$, by definition of $\alpha_{\mathcal{P}^\sharp}(P)$: $\alpha_{\mathcal{P}^\sharp}(P)(\mathcal{W}) = \alpha^{\mathcal{W}}(\{\rho \in P \mid \mathbf{def}(\rho) = \mathcal{W}\})$, hence $\rho \in \gamma^{\mathcal{W}}(\alpha_{\mathcal{P}^\sharp}(P)(\mathcal{W}))$. Moreover as $\alpha_{\mathcal{P}^\sharp}(P)(\mathcal{W}) \sqsubseteq^{\mathcal{W}} m(\mathcal{W})$, we have $\gamma^{\mathcal{W}}(\alpha_{\mathcal{P}^\sharp}(P)(\mathcal{W})) \subseteq \gamma^{\mathcal{W}}(m(\mathcal{W}))$, therefore $\rho \in \gamma^{\mathcal{W}}(m(\mathcal{W}))$. By definition of $\gamma_{\mathcal{P}^\sharp}$: $\gamma_{\mathcal{P}^\sharp}(m) = \bigcup_{\mathcal{W} \in \mathbf{def}(m)} \gamma^{\mathcal{W}}(m(\mathcal{W}))$ as $\mathcal{W} \in \mathbf{def}(m)$ we get that: $\rho \in \gamma_{\mathcal{P}^\sharp}(m)$
    - ⋆ Conversely whenever $P \subseteq \gamma_{\mathcal{P}^\sharp}(m)$, for $\alpha_{\mathcal{P}^\sharp}(P) \sqsubseteq_{\mathcal{P}^\sharp} m$ to hold (we can not have $m = \top$), we need to prove that: (1) $\alpha_{\mathcal{P}^\sharp}(P) \neq \top$ and that (2) $\forall \mathcal{W} \in \mathbf{def}(\alpha_{\mathcal{P}^\sharp}(P))$, $\mathcal{W} \in \mathbf{def}(m) \wedge \alpha_{\mathcal{P}^\sharp}(P)(\mathcal{W}) \sqsubseteq^{\mathcal{W}} m(\mathcal{W})$. By definition of $\mathcal{P}^\sharp$, $\mathbf{def}(m)$ is finite, therefore $\gamma_{\mathcal{P}^\sharp}(m) = \bigcup_{\mathcal{W} \in \mathbf{def}(m)} \gamma^{\mathcal{W}}(m(\mathcal{W}))$. As $P \subseteq \gamma_{\mathcal{P}^\sharp}(m)$, we have $|\{\mathbf{def}(\rho) \mid \rho \in P\}| \leqslant |\{\mathbf{def}(\rho) \mid \rho \in \gamma_{\mathcal{P}^\sharp}(m)\}| < \infty$, hence $\alpha_{\mathcal{P}^\sharp}(P) \neq \top$ thus proving (1). Let $\mathcal{W} \in \mathbf{def}(\alpha_{\mathcal{P}^\sharp}(P))$, let $P_{\mathcal{W}} \triangleq \{\rho \in P \mid \mathbf{def}(\rho) = \mathcal{W}\}$, by definition $P_{\mathcal{W}} \neq \emptyset$, hence $\emptyset \subset P_{\mathcal{W}} \subseteq P \subseteq \gamma_{\mathcal{P}^\sharp}(m)$ which implies that $\mathcal{W} \in \mathbf{def}(m)$, moreover $P_{\mathcal{W}} \subseteq \gamma^{\mathcal{W}}(m(\mathcal{W})) \Rightarrow \alpha^{\mathcal{W}}(P_{\mathcal{W}}) \sqsubseteq^{\mathcal{W}} m(\mathcal{W})$. As $\alpha_{\mathcal{P}^\sharp}(P)(\mathcal{W}) = \alpha^{\mathcal{W}}(P_{\mathcal{W}})$ we can conclude that (2) holds.
- otherwise we have : $\alpha_{\mathcal{P}^\sharp}(P) = \top$, hence $\alpha_{\mathcal{P}^\sharp}(P) \sqsubseteq_{\mathcal{P}^\sharp} m \Leftrightarrow \top \sqsubseteq_{\mathcal{P}^\sharp} m \Leftrightarrow m = \top$. Clearly $m = \top \Rightarrow \gamma_{\mathcal{P}^\sharp}(m) = \mathfrak{M}$. Moreover when $\gamma_{\mathcal{P}^\sharp}(m) = \mathfrak{M}$ if we assume that $m \neq \top$, then by definition $\gamma_{\mathcal{P}^\sharp}(m) = \bigcup_{\mathcal{W} \in \mathbf{def}(m)} \gamma^{\mathcal{W}}(m(\mathcal{W}))$. This union is finite (as $m \in \mathcal{P}^\sharp$), therefore as $\mathcal{V}$ is infinite, we have a contradiction, hence $m = \top$. This yields $m = \top \Leftrightarrow \gamma_{\mathcal{P}^\sharp}(m) = \mathfrak{M}$. Finally we clearly have that: $\gamma_{\mathcal{P}^\sharp}(m) = \mathfrak{M} \Rightarrow P \subseteq \gamma_{\mathcal{P}^\sharp}(m)$. Conversely, whenever $P \subseteq \gamma_{\mathcal{P}^\sharp}(m)$ we have that $\{\mathbf{def}(\rho) \mid \rho \in P\} \subseteq \{\mathbf{def}(\rho) \mid \rho \in \gamma_{\mathcal{P}^\sharp}(m)\}$, as $\{\mathbf{def}(\rho) \mid \rho \in P\}$ is not finite: $\{\mathbf{def}(\rho) \mid \rho \in \gamma_{\mathcal{P}^\sharp}(m)\}$ is not either, by definition of $\gamma_{\mathcal{P}^\sharp}(m)$ we must have that: $\gamma_{\mathcal{P}^\sharp}(m) = \top$. Therefore we have: $\alpha_{\mathcal{P}^\sharp}(P) \sqsubseteq_{\mathcal{P}^\sharp} m \Leftrightarrow P \subseteq \gamma_{\mathcal{P}^\sharp}(m)$.

□

Proof of 6.2. Let $u \in \wp_f(\mathcal{V})$ and $u' \in \wp_f(\mathcal{V})$, let $N^\sharp \in \mathcal{N}^u$ and $N^{\sharp\prime} \in \mathcal{N}^{u'}$, be such that $u \subseteq u'$ and $N^\sharp \mathrel{|\sqsubseteq} N^{\sharp\prime}$. By definition we have $N^\sharp \sqsubseteq^u N^{\sharp\prime}_{|u}$, by soundness of the $\sqsubseteq^u$ operator we have: $\gamma^u(N^\sharp) \subseteq \gamma^u(N^{\sharp\prime}_{|u})$, finally as the projection operator is exact we have: $\gamma^u(N^{\sharp\prime}_{|u}) = \gamma^{u'}(N^{\sharp\prime})_{|u}$, hence $\gamma^u(N^\sharp) \subseteq \gamma^{u'}(N^{\sharp\prime})_{|u}$. □

Proof of 6.3.

- Soundness: We want to prove that γ is monotonic for $\sqsubseteq$, which is that $\forall X^\sharp \sqsubseteq Y^\sharp$, $\gamma(X^\sharp) \subseteq \gamma(Y^\sharp)$. Let us assume that $\langle N_1^\sharp, l_1, u_1 \rangle \sqsubseteq \langle N_2^\sharp, l_2, u_2 \rangle$, let $\rho \in \gamma(\langle N_1^\sharp, l_1, u_1 \rangle)$. By definition of γ we have: there exists $l_1 \subseteq \mathcal{W} \subseteq u_1$, such that $\mathbf{def}(\rho) = \mathcal{W}$ and $\rho \in$

$(\gamma^{u_1}(N_1^\sharp))_{|\mathcal{W}}$, hence there exists $\tau : (u_1 \setminus \mathcal{W}) \nrightarrow \mathbb{I}$, such that $\rho \uplus \tau \in \gamma^{u_1}(N_1^\sharp)$. Moreover by definition of $\sqsubseteq$ we have: $l_2 \subseteq l_1 \wedge u_1 \subseteq u_2$, therefore: $l_2 \subseteq \mathcal{W} \subseteq u_2$ (a). Moreover $N_1^\sharp \mathrel{|\sqsubseteq} N_2^\sharp$, hence by soundness of this operator $\gamma^{u_1}(N_1^\sharp) \subseteq \gamma^{u_2}(N_2^\sharp)_{|u_1}$, therefore: $\rho \uplus \tau \in \gamma^{u_2}(N_2^\sharp)_{|u_1}$ and by projection: $\rho \in (\gamma^{u_2}(N_2^\sharp)_{|u_1})_{|\mathcal{W}} = \gamma^{u_2}(N_2^\sharp)_{|\mathcal{W}}$ (b). From (a) and (b) we get that: $\rho \in \gamma(\langle N_2^\sharp, l_2, u_2 \rangle)$. Thus proving that $\sqsubseteq$ is sound.

- Completeness: Assume that $\forall \mathcal{W} \in \wp_f(\mathcal{V}), \gamma^{\mathcal{W}}(U^\sharp) \subseteq \gamma^{\mathcal{W}}(V^\sharp) \Rightarrow U^\sharp \sqsubseteq^{\mathcal{W}} V^\sharp$ (i.e. the comparison operator of the underlying domain is complete). We want to prove that $\forall X^\sharp, Y^\sharp$, $\gamma(X^\sharp) \subseteq \gamma(Y^\sharp) \Rightarrow X^\sharp \sqsubseteq Y^\sharp$, let us assume that $X^\sharp = \langle N_1^\sharp, l_1, u_1 \rangle$ and $Y^\sharp = \langle N_2^\sharp, l_2, u_2 \rangle$.
  - If $(\gamma^{u_1}(N_1^\sharp))_{|l_1} = \emptyset$ then $\gamma^{u_1}(N_1^\sharp) = \emptyset$, hence $N_1^\sharp = \perp^{u_1}$, which is absurd by definition. Therefore, let $\rho \in (\gamma^{u_1}(N_1^\sharp))_{|l_1}$, we have $\rho \in \gamma(X^\sharp)$, hence $\rho \in \gamma(Y^\sharp)$, hence $\mathbf{def}(\rho) = l_1 \supseteq l_2$. *mutatis mutandis,* $u_1 \subseteq u_2$.
  - Moreover, $\gamma(X^\sharp) \subseteq \gamma(Y^\sharp)$, therefore we have: $\gamma(X^\sharp)_{|u_1} \subseteq \gamma(Y^\sharp)_{|u_1}$. As $\gamma(X^\sharp)_{|u_1} = \gamma^{u_1}(N_1^\sharp)$ and $\gamma(X^\sharp)_{|u_1} = \gamma^{u_2}(N_2^\sharp)_{|u_1}$, we have that $\gamma^{u_1}(N_1^\sharp) \subseteq \gamma^{u_2}(N_2^\sharp)_{|u_1}$. By soundness of the projection operator: $\gamma^{u_2}(N_2^\sharp)_{|u_1} \subseteq \gamma^{u_1}((N_2^\sharp)_{|u_1})$. Finally as $\gamma^{u_1}(N_1^\sharp) \subseteq \gamma^{u_1}((N_2^\sharp)_{|u_1})$, using the completeness of $\sqsubseteq^{u_1}$ we have: $N_1^\sharp \sqsubseteq^{u_1} (N_2^\sharp)_{|u_1}$.

  This last two points give us that $X^\sharp \sqsubseteq Y^\sharp$.

□

Proof of 6.4. We want to prove that: if $N_1^\sharp \in N_u$ and $N_2^\sharp \in N_\nu$, then $\gamma_0^u(N_1^\sharp) \subseteq (\gamma_0^u(N_1^\sharp \boxast N_2^\sharp))_{|u}$ and $\gamma_0^\nu(N_2^\sharp) \subseteq (\gamma_0^\nu(N_1^\sharp \boxast N_2^\sharp))_{|\nu}$.

- We start by proving what can be considered the soundness of the **strengthening** operator: whenever $\gamma^u(N_1^\sharp) \subseteq \gamma^{u \cup \nu}(N^{\sharp\prime})_{|u}$ then $\gamma^u(N_1^\sharp) \subseteq \gamma^{u \cup \nu}(\textbf{strenghtening}(C, N^{\sharp\prime}, N_1^\sharp, N_2^\sharp))_{|u}$. Therefore let us assume that $\gamma^u(N_1^\sharp) \subseteq \gamma^{u \cup \nu}(N^{\sharp\prime})_{|u}$. We prove the following invariant of the **foreach** loop of Algorithm 6.1: $\gamma^u \subseteq \gamma^{u \cup \nu}(\texttt{res})_{|u}$. This holds trivially initially as `res` is initialized to $N^{\sharp\prime}$. Whenever `res` is not modified by the `test` during some loop then we have that $N_1^\sharp \mathrel{|\sqsubseteq} \texttt{test}$ holds. By soundness of $|\sqsubseteq$ we have that $\gamma^u(N_1^\sharp) \subseteq \gamma^{u \cup \nu}(\texttt{test})_{|u}$. Which concludes the proof of the invariant. As Algorithm 6.1 returns `res`, the result is proven.
- Let us now show that $N^{\sharp\prime}$ form Definition 6.10 satisfies: $\gamma^u(N_1^\sharp) \subseteq \gamma^{u \cup \nu}(N^{\sharp\prime})_{|u}$. We have $N^{\sharp\prime} = (N_1^\sharp{}_{|\mathfrak{c}} \sqcup^{\mathfrak{c}} N_2^\sharp{}_{|\mathfrak{c}})_{|u \cup \nu}$, where $\mathfrak{c} = u \cap \nu$. Let $\rho \in \gamma^u(N_1^\sharp)$, such that $\rho = \tau \uplus \mu$ with $\tau \in \mathfrak{c} \to \mathbb{I}$ and $\mu \in (u \setminus \mathfrak{c}) \to \mathbb{I}$. By soundness of the projection we have $\tau \in \gamma^{\mathfrak{c}}(N_1^\sharp{}_{|\mathfrak{c}})$, by soundness of the join operator: $\tau \in \gamma^{\mathfrak{c}}(N_1^\sharp{}_{|\mathfrak{c}} \sqcup^{\mathfrak{c}} N_2^\sharp{}_{|\mathfrak{c}})$, finally by soundness of the abstract universal quantification: $\forall \upsilon \in ((u \cup \nu) \setminus \mathfrak{c}) \to \mathbb{I}, \tau \uplus \upsilon \in \gamma^{\mathfrak{c}}((N_1^\sharp{}_{|\mathfrak{c}} \sqcup^{\mathfrak{c}} N_2^\sharp{}_{|\mathfrak{c}})_{|u \cup \nu})$. Therefore, as $\mathbb{I} \neq \emptyset$, $\exists \eta \in (u \cup \nu)\setminus$, $\tau \uplus \mu \uplus \eta \in \gamma^{\mathfrak{c}}((N_1^\sharp{}_{|\mathfrak{c}} \sqcup^{\mathfrak{c}} N_2^\sharp{}_{|\mathfrak{c}})_{|u \cup \nu})$ and by projection on u we get $\rho = \tau \uplus \mu \in \gamma^{\mathfrak{c}}((N_1^\sharp{}_{|\mathfrak{c}} \sqcup^{\mathfrak{c}} N_2^\sharp{}_{|\mathfrak{c}})_{|u \cup \nu})_{|u}$. Hence we conclude that $\gamma^u(N_1^\sharp) \subseteq \gamma^{u \cup \nu}(N^{\sharp\prime})_{|u}$.

Using the last two points we have that $\gamma^u(N_1^\sharp) \subseteq \gamma^{u \cup \nu}(N_1^\sharp \boxast N_2^\sharp)_{|u}$, the other case is symmetric.

□

Proof of 6.5. We want to prove that $\sqcup$ soundly abstracts the union, that is $\forall U_1^\sharp, U_2^\sharp$, $\gamma(U_1^\sharp) \subseteq \gamma(U_1^\sharp \sqcup U_2^\sharp) \wedge \gamma(U_2^\sharp) \subseteq \gamma(U_1^\sharp \sqcup U_2^\sharp)$. By symmetry we only prove that $\forall U_1^\sharp, U_2^\sharp$, $\gamma(U_1^\sharp) \subseteq \gamma(U_1^\sharp \sqcup U_2^\sharp)$. Let $U_1^\sharp = \langle N_1^\sharp, l_1, u_1 \rangle$ and $U_2^\sharp = \langle N_2^\sharp, l_2, u_2 \rangle$, such that $U_1^\sharp \sqcup U_2^\sharp = \langle N_1^\sharp \boxast N_2^\sharp, l_1 \cap l_2, u_1 \cup u_2 \rangle$. Let $\rho \in \gamma(U_1^\sharp)$, let $A = \mathbf{def}(U_1^\sharp)$ we have (by definition of γ) that $l_1 \subseteq A \subseteq u_1$, therefore $l_1 \cap l_2 \subseteq A \subseteq u_1 \cup u_2$ (a). Moreover $\rho \in (\gamma^{u_1}(N_1^\sharp))_{|A}$ hence there exists $\tau : (u_1 \setminus A) \nrightarrow \mathbb{I}$ such that $\rho \uplus \tau \in \gamma^{u_1}(N_1^\sharp)$. By soundness of $\boxast$ we have: $\rho \uplus \tau \in \gamma^{u_1}(N_1^\sharp) \subseteq (\gamma^{u_1 \cup \nu_1}(N_1^\sharp \boxast N_2^\sharp))_{|u_1}$ and by projection: $\rho \in ((\gamma^{u_1 \cup \nu_1}(N_1^\sharp \boxast N_2^\sharp))_{|u_1})_{|A} = (\gamma^{u_1 \cup u_2}(N_1^\sharp \boxast N_2^\sharp))_{|A}$ (b). From (a) and (b) we get that $\rho \in \gamma(U_1^\sharp \sqcup U_2^\sharp)$. □

Proof of 6.6. The proposition holds trivially for the first two cases of the definition. The third case is similar to the definition of $\boxast$, therefore, using the soundness of the **strengthening** operator proved in Proposition 6.4, we know that we only need to prove that $N^{\sharp\prime}$ form Definition 6.12 satisfies: $\gamma^{u}(N^{\sharp}_{1}) \subseteq \gamma^{u\cup v}(N^{\sharp\prime})_{|u}$. We have $N^{\sharp\prime} = (N^{\sharp}_{1\,|\mathfrak{c}} \sqcup^{\mathfrak{c}} N^{\sharp}_{2\,|\mathfrak{c}})_{|u\cup v} \sqcap^{u\cup v} (N^{\sharp}_{1\,|u\setminus\mathfrak{c}})_{|u\cup v} \sqcap^{u\cup v} (N^{\sharp}_{2\,|v\setminus\mathfrak{c}})_{|u\cup v}$.

- Following the proof of Proposition 6.4, if $\rho \in \gamma^{u}(N^{\sharp}_{1})$, such that $\rho = \tau \uplus \mu$ with $\tau \in \mathfrak{c} \to \mathbb{I}$ and $\mu \in (u \setminus \mathfrak{c}) \to \mathbb{I}$ then $\forall \eta \in (u \cup v) \setminus u$, $\tau \uplus \mu \uplus \eta \in \gamma^{u\cup v}((N^{\sharp}_{1\,|\mathfrak{c}} \sqcup^{\mathfrak{c}} N^{\sharp}_{2\,|\mathfrak{c}})_{|u\cup v})$.
- Moreover as $N^{\sharp}_{2} \neq \bot^{v}$ we have that $\gamma^{v}(N^{\sharp}_{2}) \neq \emptyset$, let $\rho_2 \in \gamma^{v}(N^{\sharp}_{2})$, such that $\rho_2 = \tau_2 \uplus \mu_2$ with $\tau_2 \in \mathfrak{c} \to \mathbb{I}$ and $\mu_2 \in (v \setminus \mathfrak{c}) \to \mathbb{I}$ by soundness of the projection operator, we have that $\mu_2 = \rho_{2\,|v\setminus\mathfrak{c}} \in \gamma^{v\setminus\mathfrak{c}}(N^{\sharp}_{2\,|v\setminus\mathfrak{c}})$, by soundness of the universal quantification operator, we have that $\forall \theta_2 \in u \to \mathbb{I}$, $\mu_2 \uplus \theta_2 \in \gamma^{u\cup v}((N^{\sharp}_{2\,|v\setminus\mathfrak{c}})_{|u\cup v})$. In particular, $\mu_2 \uplus \tau \uplus \mu \in \gamma^{u\cup v}((N^{\sharp}_{2\,|v\setminus\mathfrak{c}})_{|u\cup v})$.
- Finally by soundness of projection we have: $\mu \in \gamma^{u\setminus\mathfrak{c}}(N^{\sharp}_{1\,|u\setminus\mathfrak{c}})$, by soundness of universal quantification: $\forall \theta_1 \in v \to \mathbb{I}, \mu \uplus \theta_2 \in \gamma^{u\cup v}((N^{\sharp}_{1\,|u\setminus\mathfrak{c}})_{|u\cup v})$. In particular: $\mu \uplus \tau \uplus \mu_2 \in \gamma^{u\cup v}((N^{\sharp}_{2\,|v\setminus\mathfrak{c}})_{|u\cup v})$

Therefore we have that $\mu \uplus \tau \uplus \mu_2 \in \gamma^{u\cup v}((N^{\sharp}_{1\,|\mathfrak{c}} \sqcup^{\mathfrak{c}} N^{\sharp}_{2\,|\mathfrak{c}})_{|u\cup v})$, that $\mu_2 \uplus \tau \uplus \mu \in \gamma^{u\cup v}((N^{\sharp}_{2\,|v\setminus\mathfrak{c}})_{|u\cup v})$ and also that $\mu \uplus \tau \uplus \mu_2 \in \gamma^{u\cup v}((N^{\sharp}_{2\,|v\setminus\mathfrak{c}})_{|u\cup v})$. By soundness of the $\sqcup^{u\cup v}$ operator, we get that $\mu \uplus \tau \uplus \mu_2 \in \gamma^{u\cup v}(N^{\sharp\prime})$, by projection we get that: $\rho = (\mu \uplus \tau \uplus \mu_2)_{|u} \in \gamma^{u\cup v}(N^{\sharp\prime})_{|u}$. □

Proof of 6.7. We have to prove that $\forall U^{\sharp}_{1}, U^{\sharp}_{2}, \gamma(U^{\sharp}_{1}) \cap \gamma(U^{\sharp}_{2}) \subseteq \gamma(U^{\sharp}_{1} \sqcap U^{\sharp}_{2})$. Let $\langle N^{\sharp}_{1}, l_1, u_1 \rangle$ be $U^{\sharp}_{1}$ and $\langle N^{\sharp}_{2}, l_2, u_2 \rangle$ be $U^{\sharp}_{2}$.

- If $l_1 \cup l_2 \not\subseteq u_1 \cap u_2$, let us assume that there exists $\rho \in \gamma(U^{\sharp}_{1}) \cap \gamma(U^{\sharp}_{2})$, we have $l_1 \subseteq \mathbf{def}(\rho) \subseteq u_1$ and $l_2 \subseteq \mathbf{def}(\rho) \subseteq u_2$, hence $l_1 \cup l_2 \subseteq \mathbf{def}(\rho) \subseteq u_1 \cap u_2$. This is absurd, hence $\gamma(U^{\sharp}_{1}) \cap \gamma(U^{\sharp}_{2}) = \emptyset$, as $U^{\sharp}_{1} \sqcap U^{\sharp}_{2} = \bot$, we indeed have $\gamma(U^{\sharp}_{1}) \cap \gamma(U^{\sharp}_{2}) \subseteq \gamma(U^{\sharp}_{1} \sqcap U^{\sharp}_{2})$
- If $l_1 \cup l_2 \not\subseteq u_1 \cap u_2$, let $\rho \in \gamma(U^{\sharp}_{1}) \cap \gamma(U^{\sharp}_{2})$, let $A = \mathbf{def}(\rho)$, we have $l_1 \cup l_2 \subseteq A \subseteq u_1 \cap u_2$, moreover we also have $\rho \in \gamma^{u_1}(N^{\sharp}_{1})_{|A}$ and $\rho \in \gamma^{u_2}(N^{\sharp}_{2})_{|A}$. Let us consider $\theta_1 \in (u_1 \setminus A) \to \mathbb{I}$ (resp. $\theta_2 \in (u_2 \setminus A) \to \mathbb{I}$) such that $\rho \uplus \theta_1 \in \gamma^{u_1}(N^{\sharp}_{1})$ (resp. $\rho \uplus \theta_2 \in \gamma^{u_2}(N^{\sharp}_{2})$). Moreover by soundness of the projection operator $(\rho \uplus \theta_1)_{|u_1\cap u_2} = \rho \uplus \theta_{1\,|(u_1\cap u_2)\setminus A} \in \gamma^{u_1\cap u_2}(N^{\sharp}_{1\,|u_1\cap u_2})$. Let us decompose $\rho = \tau \uplus \mu$, where $\tau \in (l_1 \cup l_2) \to \mathbb{I}$ and $\mu \in (A \setminus (l_1 \cup l_2)) \to \mathbb{I}$. By soundness of the projection operator, we have $\tau \in \gamma^{l_1\cup l_2}(N^{\sharp}_{2\,|l_1\cup l_2})$, and by soundness of the universal quantification we get that $\forall \theta \in ((u_1 \cap u_2) \setminus (l_1 \cap l_2)) \to \mathbb{I}, \tau \uplus \theta \in \gamma^{u_1\cap u_2}((N^{\sharp}_{2\,|l_1\cup l_2})_{|u_1\cap u_2})$. In particular, let us consider the case where $\theta = \mu \uplus \theta_{1\,|(u_1\cap u_2)\setminus A}$, we have $\mathbf{def}(\theta) = \mathbf{def}(\mu) \cup \mathbf{def}(\theta_{1\,|(u_1\cap u_2)\setminus A}) = (A \setminus (l_1 \cup l_2)) \cup ((u_1 \cap u_2) \setminus A) = (u_1 \cap u_2) \setminus (l_1 \cup l_2)$. Hence we have: $\tau \uplus \theta \in \gamma^{u_1\cap u_2}((N^{\sharp}_{2\,|l_1\cup l_2})_{|u_1\cap u_2})$, $\tau \uplus \theta = \tau \uplus \mu \uplus \theta_{1\,|(u_1\cap u_2)\setminus A} = \rho \uplus \theta_{1\,|(u_1\cap u_2)\setminus A}$. As $\rho \uplus \theta_{1\,|(u_1\cap u_2)\setminus A} \in \gamma^{u_1\cap u_2}((N^{\sharp}_{2\,|l_1\cup l_2})_{|u_1\cap u_2})$ and $\rho \uplus \theta_{1\,|(u_1\cap u_2)\setminus A} \in \gamma^{u_1\cap u_2}(N^{\sharp}_{1\,|u_1\cap u_2})$, we can conclude by soundness of the intersection operator that $\rho \uplus \theta_{1\,|(u_1\cap u_2)\setminus A} \in \gamma^{u_1\cap u_2}(N^{\sharp}_{1\,|u_1\cap u_2} \sqcap^{u_1\cap u_2} (N^{\sharp}_{2\,|l_1\cup l_2})_{|u_1\cap u_2})$. Finally by projection onto A we get that $\rho \in \gamma^{u_1\cap u_2}(N^{\sharp}_{1\,|u_1\cap u_2} \sqcap^{u_1\cap u_2} (N^{\sharp}_{2\,|l_1\cup l_2})_{|u_1\cap u_2})_{|A}$, together with the fact that $l_1 \cup l_2 \subseteq A \subseteq u_2 \cap u_2$ we get that $\rho \in \gamma(U^{\sharp}_{1} \sqcap U^{\sharp}_{2})$

□

Proof of 6.8. We want to prove that $\triangledown$ soundly abstracts the union and that it stabilizes infinite chains.

- We start by proving that $\triangledown$ soundly abstract the union. Which means that: $\forall U^{\sharp}_{1}, U^{\sharp}_{2}$, $\gamma(U^{\sharp}_{1}) \subseteq \gamma(U^{\sharp}_{1} \triangledown U^{\sharp}_{2}) \wedge \gamma(U^{\sharp}_{2}) \subseteq \gamma(U^{\sharp}_{1} \triangledown U^{\sharp}_{2})$. Let $U^{\sharp}_{1} = \langle N^{\sharp}_{1}, l_1, u_1 \rangle$ and $U^{\sharp}_{2} = \langle N^{\sharp}_{2}, l_2, u_2 \rangle$.

  - If $l_2 \subset l_1$ and $u_2 \subseteq u$, we then have $U_1^\sharp \triangledown U_2^\sharp = U_1^\sharp \sqcup U_2^\sharp$, soundness is then ensured by Proposition 6.5.
  - If $u_1 \subset u_2$ $U_1^\sharp \triangledown U_2^\sharp = \top$ which ensures soundness.
  - Finally if $l_1 \subseteq l_2 \wedge u_2 \subseteq u_1$: we have $U_1^\sharp \triangledown U_2^\sharp = \langle N_1^\sharp \triangledown^{u_1} N^\sharp_{2\,|u_1}, l_1, u_1 \rangle$. Let $\rho_1 \in \gamma(U_1^\sharp)$ and $\rho_2 \in \gamma(U_2^\sharp)$. We have: $l_1 \subseteq \mathbf{def}(\rho_1) = A_1 \subseteq u_1$ and $l_1 \subseteq l_2 \subseteq \mathbf{def}(\rho_2) = A_2 \subseteq u_2 \subseteq u_1$. Moreover $\rho_1 \in (\gamma^{u_1}(N_1^\sharp))_{|A_1}$ hence there exists $\tau_1 : (u_1 \setminus A_1) \to \mathbb{I}$ such that $\rho \uplus \tau_1 \in \gamma^{u_1}(N_1^\sharp)$ and as $\rho_2 \in (\gamma^{u_2}(N_2^\sharp))_{|A_2}$ there exists $\tau_2 : u_2 \setminus A_2 \to \mathbb{I}$ such that $\rho_2 \uplus \tau_2 \in \gamma^{u_2}(N_2^\sharp)$. Moreover as $u_2 \subseteq u_1$, by soundness of the universal quantification and as $\mathbb{I} \neq \emptyset$, there exists τ_2' such that $\rho_2 \uplus \tau_2 \uplus \tau_2' \in \gamma^{u_1}((N_2^\sharp)_{|u_1})$. By soundness of $\triangledown^{u_1}$ we get that: $\rho_1 \uplus \tau_1 \in \gamma^{u_1}(N_1^\sharp \triangledown^{u_1} (N_2^\sharp)_{|u_1})$ and $\rho_2 \uplus \tau_2 \uplus \tau_2' \in \gamma^{u_1}(N_1^\sharp \triangledown^{u_1} N^\sharp_{2\,|u_1})$. Finally by projection we get that: $\rho_1 \in (\gamma^{u_1}(N_1^\sharp \triangledown^{u_1} (N_2^\sharp)_{|u_1}))_{|A_1}$ and $\rho_2 \in (\gamma^{u_2}(N_1^\sharp \triangledown^{u_1} (N_2^\sharp)_{|u_1}))_{|A_2}$. Finally we have that: $\rho_1 \in \gamma(U_1^\sharp \triangledown U_2^\sharp)$ and $\rho_2 \in \gamma(U_1^\sharp \triangledown U_2^\sharp)$. Thus proving soundness.

- We now prove the infinite sequences are stabilized by the widening. Consider a sequence $(U_i^\sharp)_{i \geqslant 0} \in \mathfrak{M}$ and let $(V_i^\sharp)_{i \geqslant 0}$ be: $V_0^\sharp = U_0^\sharp$ and $V_{i+1}^\sharp = V_i^\sharp \triangledown U_{i+1}^\sharp$. If for some $i \geqslant 0$, $V_i^\sharp = \top$ we are given by soundness of $\triangledown$, that $\forall j \geqslant i, V_j^\sharp = \top$, and therefore the sequence is stabilized. Therefore we assume in the following that $\forall i \geqslant 0,\ V_i^\sharp \neq \top$. Given an abstract element $X^\sharp$ we denote $X^\sharp[N^\sharp]$, $X^\sharp[l]$, $X^\sharp[u]$ the elements such that: $X^\sharp = \langle X^\sharp[N^\sharp]$, $X^\sharp[l]$, $X^\sharp[u]\rangle$.
  - Consider the sequence $(V_i^\sharp[u])_{i \geqslant 0} \in \wp(\mathcal{V})^\star$. We have that: $\forall i, U_{i+1}^\sharp[u] \subseteq V_i^\sharp[u]$ otherwise $V_{i+1}^\sharp[u] = \top$. Hence $\forall i \geqslant 0, V_{i+1}^\sharp[u] = V_i^\sharp[u]$.
  - Consider then the sequence $(V_i^\sharp[l])_{i \geqslant 0} \in \wp(\mathcal{V})^\star$ we have: $\forall i \geqslant 0, V_{i+1}^\sharp[l] \subseteq V_i^\sharp[l]$. However $V_0^\sharp[l]$ is a finite set, therefore we can not have an infinitely decreasing sequence, hence there exists $j \geqslant 0$ such that $\forall k \geqslant j, V_k^\sharp[l] = V_j^\sharp[l]$.
  - Consider finally the sequence: $(Z_k^\sharp)_{k \geqslant 0} = (V_{k+j}^\sharp)_{k \geqslant 0}$, we have: $\forall k \geqslant 0, Z_k^\sharp[l] = Z_{k+1}^\sharp[l] \wedge Z_k^\sharp[u] = Z_{k+1}^\sharp[u]$, hence we have that: $\forall k \geqslant 0, U_{k+j}^\sharp[u] \subseteq Z_k^\sharp[u] \wedge Z_k^\sharp[l] \subseteq U_{k+j}^\sharp[l]$, hence there exists an $u \in \wp(V)$ such that $\forall k \geqslant 0, Z_{k+1}^\sharp[N^\sharp] = Z_k^\sharp[N^\sharp] \triangledown^u U_{k+j+1}^\sharp[N^\sharp]$ by the stabilization property of the $\triangledown^u$ operator we get that sequence $(Z_k^\sharp[N^\sharp])_{k \geqslant 0}$ stabilizes, hence $(V_k^\sharp[N^\sharp])_{k \geqslant 0}$ stabilizes.

  Hence we have stabilization of the sequence $(V_k^\sharp)_{k \geqslant 0}$.

□

Proof of 6.9. Let us prove that $\forall \texttt{stmt}, \mathbb{S}\llbracket \texttt{stmt} \rrbracket(\gamma(U^\sharp)) \subseteq \gamma(\mathbb{S}^\sharp\llbracket \texttt{stmt} \rrbracket(U^\sharp))$ holds. Let $U^\sharp = \langle N^\sharp, l, u\rangle \in \mathfrak{M}^\sharp$. We recall here the definition of the concrete operators:

$$\mathbb{S}\llbracket \texttt{Assume(c)} \rrbracket_{\mathbb{I}}(R \in \mathfrak{M}) = \bigcup_{\mathbf{fv}(c) \subseteq \mathcal{W} \subseteq \mathcal{V}} \mathbb{S}\llbracket \texttt{Assume(c)} \rrbracket_{\mathbb{I}}^{\mathcal{W}}(R_{\mathcal{W}})$$

$$\mathbb{S}\llbracket x \leftarrow e \rrbracket_{\mathbb{I}}(R \in \mathfrak{M}) = \bigcup_{(\{x\} \cup \mathbf{fv}(c)) \subseteq \mathcal{W} \subseteq \mathcal{V}} \mathbb{S}\llbracket x \leftarrow e \rrbracket_{\mathbb{I}}^{\mathcal{W}}(R_{\mathcal{W}})$$

$$\mathbb{S}\llbracket \texttt{remove}(v) \rrbracket_{\mathbb{I}}(R \in \mathfrak{M}) = \bigcup_{\mathcal{W} \subseteq \mathcal{V}} \{\rho_{|\mathcal{W} \setminus \{v\}} \mid \rho \in R_{\mathcal{W}}\}$$

$$\mathbb{S}\llbracket \texttt{add}(v) \rrbracket_{\mathbb{I}}(R \in \mathfrak{M}) = \bigcup_{\mathcal{W} \subseteq \mathcal{V} \setminus \{v\}} \{\rho[v \mapsto i] \mid \rho \in R_{\mathcal{W}}, i \in \mathbb{I}\} \cup \bigcup_{\{v\} \subseteq \mathcal{W} \subseteq \mathcal{V}} R_{\mathcal{W}}$$

- If $\texttt{stmt} = \texttt{remove}(v)$. Let $\rho \in \mathbb{S}\llbracket \texttt{remove}(v) \rrbracket(\gamma(U^\sharp)) = \bigcup_{\mathcal{W} \subseteq \mathcal{V}} \{\rho_{|\mathcal{W} \setminus \{v\}} \mid \rho \in \gamma(U^\sharp)_{\mathcal{W}}\}$.

  - Let $\{v\} \subseteq \mathcal{W} \subseteq \mathcal{V}$, such that $\rho \in \{\tau_{|\mathcal{W}\setminus\{v\}} \mid \tau \in \gamma(U^\sharp)_\mathcal{W}\}$, let $\tau \in \gamma(U^\sharp)_\mathcal{W}$ such that $\rho = \tau_{|\mathcal{W}\setminus\{v\}}$. We have that $\tau \in \gamma(U^\sharp)$ and $\mathbf{def}(\tau) = \mathcal{W}$, hence $\tau \in \gamma^u(N^\sharp)_{|\mathcal{W}}$ and $l \subseteq \mathcal{W} \subseteq u$, there exists $\mu \in (u \setminus \mathcal{W}) \to \mathbb{I}$ such that $\tau \uplus \mu \in \gamma^u(N^\sharp)$, and by soundness of the projection operator we have that $(\tau \uplus \mu)_{|u\setminus\{v\}} = \tau_{|u\setminus\{v\}} \uplus \mu = \rho \uplus \mu \in \gamma^{u\setminus\{v\}}(N^\sharp_{|u\setminus\{v\}})$, and finally by projecting onto $\mathcal{W} \setminus \{v\}$ we have $\rho \uplus \mu_{|\mathcal{W}\setminus\{v\}} = \rho \in \gamma^{u\setminus\{v\}}(N^\sharp_{|u\setminus\{v\}})_{|\mathcal{W}\setminus\{v\}}$. Moreover we have $\mathbf{def}(\rho) = \mathcal{W} \setminus \{v\}$, which is such that $l \setminus \{v\} \subseteq \mathcal{W} \setminus \{v\} \subseteq u \setminus \{v\}$. Finally $\rho \in \gamma(\langle N^\sharp_{|u\setminus\{v\}}, l \setminus \{v\}, u \setminus \{v\}\rangle) = \gamma(\mathbb{S}^\sharp\llbracket \mathsf{remove}(v) \rrbracket(U^\sharp))$.
  - Let $v \notin \mathcal{W} \subseteq \mathcal{V}$ such that $\rho \in \{\tau_{|\mathcal{W}\setminus\{v\}} \mid \tau \in \gamma(U^\sharp)_\mathcal{W}\}$, let $\tau \in \gamma(U^\sharp)_\mathcal{W}$ such that $\rho = \tau_{|\mathcal{W}\setminus\{v\}} = \tau$. We have $\mathbf{def}(\rho) = \mathbf{def}(\tau) = \mathcal{W}$. As $\tau \in \gamma(U^\sharp)_\mathcal{W}$, we have $l \subseteq \mathcal{W} \subseteq u$, hence $l \setminus \{v\} \subseteq \mathcal{W} \setminus \{v\} \subseteq u \setminus \{v\}$, therefore $l \setminus \{v\} \subseteq \mathbf{def}(\rho) \setminus \{v\} \subseteq u \setminus \{v\}$. Moreover as $\tau \in \gamma(U^\sharp)_\mathcal{W}$ there exists $\mu \in u \setminus \mathcal{W} \to \mathbb{I}$ such that $\tau \uplus \mu \in \gamma^u(N^\sharp)$. By soundness of the projection operator: $(\tau \uplus \mu)_{|u\setminus\{v\}} \in \gamma^{u\setminus\{v\}}(\mathbb{S}^\sharp\llbracket \mathsf{remove}(v) \rrbracket(N^\sharp))$. Finally as $(\tau \uplus \mu)_{|\setminus\{v\}} = \tau \uplus \mu_{|u\setminus\{v\}}$, we have $\tau \in \gamma^{u\setminus\{v\}}(\mathbb{S}^\sharp\llbracket \mathsf{remove}(v) \rrbracket(N^\sharp))_{|\mathcal{W}}$ and $\tau \in \gamma(\mathbb{S}^\sharp\llbracket \mathsf{remove}(v) \rrbracket(U^\sharp))$.

- If `stmt` $=$ $\mathsf{add}(v)$. Let $\rho \in \mathbb{S}\llbracket \mathsf{add}(v) \rrbracket(\gamma(U^\sharp)) = \bigcup_{\mathcal{W}\subseteq\mathcal{V}\setminus\{v\}}\{\rho[v \mapsto i] \mid \rho \in \gamma(U^\sharp)_\mathcal{W}, i \in \mathbb{I}\} \cup \bigcup_{\{v\}\subseteq\mathcal{W}\subseteq\mathcal{V}} \gamma(U^\sharp)_\mathcal{W}$.
  - If $v \in l$ then $\forall \mathcal{W}, \mathcal{W} \subseteq \mathcal{V} \setminus \{v\} = \emptyset \Rightarrow \gamma(U^\sharp)_\mathcal{W} = \emptyset$. Indeed all maps from $\gamma(U^\sharp)$ are defined on v. Therefore there exists $\{v\} \subseteq \mathcal{W} \subseteq \mathcal{V}$ such that $\rho \in \gamma(U^\sharp)_\mathcal{W}$, hence $\rho \in \gamma(U^\sharp)$. As $\mathbb{S}^\sharp\llbracket \mathsf{add}(v) \rrbracket(U^\sharp) = U^\sharp$ we have $\rho \in \gamma(\mathbb{S}^\sharp\llbracket \mathsf{add}(v) \rrbracket(U^\sharp))$.
  - Otherwise if $\exists \{v\} \subseteq \mathcal{W} \subseteq \mathcal{V}, \rho \in \gamma(U^\sharp)_\mathcal{W}$, we have $\mathbf{def}(\rho) = \mathcal{W}$, $\rho \in \gamma^u(U^\sharp)_{|\mathcal{W}}$ and $l \subseteq \mathcal{W} \subseteq u$. There exists $\tau \in u \setminus \mathcal{W}$ such that $\rho \uplus \tau \in \gamma^u(U^\sharp)$. We have, by soundness of the projection operator, $(\rho \uplus \mu)_{|u\setminus\{v\}} = \rho_{|u\setminus\{v\}} \uplus \mu \in \gamma^{u\setminus\{v\}}(N^\sharp_{|u\setminus\{v\}})$, by soundness of the universal quantification we have that $\forall i \in \mathbb{I}, (v \to i) \uplus \rho_{|u\setminus\{v\}} \uplus \mu \in \gamma^u((N^\sharp_{|u\setminus\{v\}})_{|u\cup\{v\}})$. In particular by choosing $i = \rho(v)$ we have: $(v \to \rho(v)) \uplus \rho_{|u\setminus\{v\}} \uplus \mu = \rho \uplus \mu \in \gamma^u((N^\sharp_{|u\setminus\{v\}})_{|u\cup\{v\}})$. By projection onto $\mathcal{W}$ we get that $(\rho \uplus \mu)_{|\mathcal{W}} = \rho \in (\gamma^u((N^\sharp_{|u\setminus\{v\}})_{|u\cup\{v\}}))_{|\mathcal{W}}$. Moreover as $l \cup \{v\} \subseteq \mathcal{W} \subseteq u \cup \{v\}$. Finally we have $\rho \in \gamma(\langle (N^\sharp_{|u\setminus\{v\}})_{|u\cup\{v\}}, l \cup \{v\}, u \cup \{v\}\rangle) = \gamma(\mathbb{S}^\sharp\llbracket \mathsf{add}(v) \rrbracket(U^\sharp))$.
  - Otherwise if $\exists \mathcal{W} \in \mathcal{V} \setminus \{v\}, \rho \in \{\tau[v \mapsto i] \mid \tau \in \gamma(U^\sharp)_\mathcal{W}, i \in \mathbb{I}\}$. Let $i_0 \in \mathbb{I}$ and $\tau \in \gamma(U)_\mathcal{W}$ such that $\rho = \tau[v \mapsto i_0]$. As $\tau \in \gamma(U)_\mathcal{W}$ we have that $\mathbf{def}(\tau) = \mathcal{W}$ and $\tau \in \gamma^u(N^\sharp)_{|\mathcal{W}}$, there exists $\mu \in (u \setminus \mathcal{W}) \to \mathbb{I}$ such that $\tau \uplus \mu \in \gamma^u(N^\sharp)$.
    * If $v \in u$: By soundness of the projection operator we have $(\tau \uplus \mu)_{|u\setminus\{v\}} = \tau \uplus \mu_{|u\setminus\{v\}} \in \gamma^{u\setminus\{v\}}(N^\sharp_{|u\setminus\{v\}})$. By soundness of the universal quantification we have $\forall i \in \mathbb{I}, (v \mapsto i) \uplus \tau \uplus \mu_{|u\setminus\{v\}} \in \gamma^u((N^\sharp_{|u\setminus\{v\}})_{|u\cup\{v\}})$. As $\mathbb{I} \neq \emptyset$, in particular we have $(v \mapsto i_0) \uplus \tau \uplus \mu_{|u\setminus\{v\}} \in \gamma^u((N^\sharp_{|u\setminus\{v\}})_{|u\cup\{v\}})$. Finally by projecting onto $\mathcal{W} \cup \{v\}$ we get: $((v \mapsto i_0) \uplus \tau \uplus \mu_{|u\setminus\{v\}})_{|\mathcal{W}\cup\{v\}} = \tau \uplus (v \mapsto i_0) = \tau[v \mapsto i_0] = \rho \in \gamma^u((N^\sharp{}_{|u\setminus\{v\}})_{|u\cup\{v\}})_{|\mathcal{W}\cup\{v\}}$. Moreover as $\mathbf{def}(\rho) = \mathcal{W} \cup \{v\}$ is such that $\cup \{v\} \subseteq \mathcal{W} \cup \{v\} \subseteq u \cup \{v\}$ we have that $\rho \in \gamma(\langle (N^\sharp_{|u\setminus\{v\}})_{|u\cup\{v\}}, l \cup \{v\}, u \cup \{v\}\rangle) = \gamma(\mathbb{S}^\sharp\llbracket \mathsf{add}(v) \rrbracket(U^\sharp))$
    * Finally if $v \notin u$, we have $U^\sharp_{|u\setminus\{v\}} = U^\sharp$. Moreover by soundness of the universal quantification operator we have $\forall i \in \mathbb{I}, \tau \uplus \mu \uplus (v \mapsto i) \in (\gamma^u(N^\sharp_{|u\cup\{v\}}))$. In particular $\tau \uplus \mu \uplus (v \mapsto i_0) \in \gamma^u(N^\sharp_{|u\cup\{v\}})$, hence by projection on $\{\mathcal{W} \cup \{v\}\}$ we get $(\tau \uplus \mu \uplus (v \mapsto i_0))_{|\mathcal{W}\cup\{v\}} = \tau \uplus (v \mapsto i_0) = \rho \in (\gamma^u(N^\sharp_{|u\cup\{v\}}))_{|\mathcal{W}\cup\{v\}}$. Moreover as $l \cup \{v\} \subseteq \mathcal{W} \cup \{v\} \subseteq u \cup \{v\}$, we finally have that $\rho \in \gamma(\langle N^\sharp_{|u\cup\{v\}}, l \cup \{v\}, u \cup \{v\}\rangle) = \gamma(\mathbb{S}^\sharp\llbracket \mathsf{add}(v) \rrbracket(U^\sharp))$.

- If `stmt` $=$ $\mathsf{Assume}(c)$ for some constraint $c \in$ *expr* . Let $\rho \in \mathbb{S}\llbracket \mathsf{Assume}(c) \rrbracket(\gamma(U^\sharp)) =$

$\bigcup_{\mathbf{fv}(c)\subseteq\mathcal{W}\subseteq\mathcal{V}} \mathbb{S}[\![\texttt{Assume(c)}]\!]^{\mathcal{W}}_{\mathbb{I}}(\gamma(U^\sharp)_{\mathcal{W}})$. Let $\mathbf{fv}(c) \subseteq \mathcal{W} \subseteq \mathcal{V}$ be the set of variables such that $\rho \in \mathbb{S}[\![\texttt{Assume(c)}]\!]^{\mathcal{W}}_{\mathbb{I}}(\gamma(U^\sharp)_{\mathcal{W}})$. By definition of $\mathbb{S}[\![\texttt{Assume(c)}]\!]^{\mathcal{W}}_{\mathbb{I}}$ we have $\mathbf{def}(\rho) = \mathcal{W}$, $\rho \in \gamma(U^\sharp)$ and $\mathbb{E}[\![c]\!]^{\mathcal{W}}_{\mathbb{I}}(\rho) = \texttt{true}$. As $\rho \in \gamma^{u}(N^\sharp)_{|\mathcal{W}}$, there exists $\tau \in u \setminus \mathcal{W} \to \mathbb{I}$ such that $\rho \uplus \tau \in \gamma^{u}(N^\sharp)$. As $\mathbf{def}(\tau) \cap \mathbf{fv}(c) = 0$, we have $\mathbb{E}[\![c]\!]^{u}_{\mathbb{I}}(\rho \uplus \tau) = \texttt{true}$. Therefore by definition of $\mathbb{S}^\sharp[\![\texttt{Assume(c)}]\!]^{u}$ we have: $\rho \uplus \tau \in \mathbb{S}^\sharp[\![\texttt{Assume(c)}]\!]^{u}(\gamma^{u}(N^\sharp))$. Therefore by soundness of the underlying $\mathbb{S}^\sharp[\![\texttt{Assume(c)}]\!]^{u}$ operator we have that: $\rho \uplus \tau \in \gamma^{u}(\mathbb{S}^\sharp[\![\texttt{Assume(c)}]\!]^{u}(N^\sharp))$. Once again, by projection over $\mathcal{W}$, we get $\rho \uplus \tau_{|\mathcal{W}} = \rho \in \gamma^{u}(\mathbb{S}^\sharp[\![\texttt{Assume(c)}]\!]^{u}(N^\sharp))_{|\mathcal{W}}$, moreover we have $\mathbf{fv}(c) \subseteq \mathcal{W}$ and as $\rho \in \gamma(U^\sharp)$, we have $l \subseteq \mathbf{def}(\rho) = \mathcal{W} \subseteq u$. Finally we have $l \cup \mathbf{fv}(c) \subseteq \mathcal{W} \subseteq u$. This gives us that $\rho \in \gamma(\langle \mathbb{S}^\sharp[\![\texttt{Assume(c)}]\!]^{u}(N^\sharp), l \cup \mathbf{fv}(c), u\rangle) = \gamma(\mathbb{S}^\sharp[\![\texttt{Assume(c)}]\!](U^\sharp))$.

- If $\texttt{stmt} = x \leftarrow e$ for some constraint variable $x \in \mathcal{V}$ and some expression $e \in$ *expr*. Let $\rho \in \mathbb{S}[\![x \leftarrow e]\!](\gamma(U^\sharp)) = \bigcup_{\{x\}\cup\mathbf{fv}(e)\subseteq\mathcal{W}\subseteq\mathcal{V}} \mathbb{S}[\![x \leftarrow e]\!]^{\mathcal{W}}_{\mathbb{I}}(\gamma(U^\sharp)_{\mathcal{W}})$. Let $\{x\} \cup \mathbf{fv}(c) \subseteq \mathcal{W} \subseteq \mathcal{V}$ be the set of variables such that $\rho \in \mathbb{S}[\![x \leftarrow e]\!]^{\mathcal{W}}_{\mathbb{I}}(\gamma(U^\sharp)_{\mathcal{W}})$. By definition of $\mathbb{S}[\![x \leftarrow e]\!]^{\mathcal{W}}_{\mathbb{I}}$ we have $\mathbf{def}(\rho) = \mathcal{W}$, and there exists μ, $i \in \mathbb{I}$ such that $i \in \mathbb{E}[\![c]\!]^{\mathcal{W}}_{\mathbb{I}}(\mu)$ and $\rho = \mu[x \mapsto i]$ and $\mu \in \gamma(U^\sharp)$. From $\mu \in \gamma(U^\sharp)$ we can conclude that $\mathbf{def}(\mu) = \mathcal{W}$, $l \subseteq \mathcal{W} \subseteq u$, $\mu \in \gamma^{u}(N^\sharp)_{|\mathcal{W}}$, let $\tau \in u \setminus \mathcal{W} \to \mathbb{I}$ be such that $\mu \uplus \tau \in \gamma^{u}(N^\sharp)$. As $\mathbf{def}(\tau) \cap \mathbf{fv}(e) = \emptyset$ we have $\mathbb{E}[\![c]\!]^{\mathcal{W}}_{\mathbb{I}}(\mu \uplus \tau) = \mathbb{E}[\![c]\!]^{\mathcal{W}}_{\mathbb{I}}(\mu)$. By definition of $\mathbb{S}[\![x \leftarrow e]\!]^{u}_{\mathbb{I}}$ we have $(\mu \uplus \tau)[x \mapsto \mathbb{E}[\![c]\!]^{\mathcal{W}}_{\mathbb{I}}(\mu \uplus \tau)] = \rho \uplus \tau \in \mathbb{S}[\![x \leftarrow e]\!]^{u}_{\mathbb{I}}(\gamma^{u}(N^\sharp))$. By soundness of $\mathbb{S}^\sharp[\![x \leftarrow e]\!]^{u}_{\mathbb{I}}$, we get that $\rho \uplus \tau \in \gamma^{u}(\mathbb{S}^\sharp[\![x \leftarrow e]\!]^{u}_{\mathbb{I}}(N^\sharp))$, by projection we have $\rho \in \gamma^{u}(\mathbb{S}^\sharp[\![x \leftarrow e]\!]^{u}_{\mathbb{I}}(N^\sharp))_{|\mathcal{W}}$. Finally we conclude that $\rho \in \gamma(\langle \mathbb{S}^\sharp[\![x \leftarrow e]\!]^{u}_{\mathbb{I}}(N^\sharp), l \cup \{x\} \cup \mathbf{fv}(e), u\rangle) = \gamma(\mathbb{S}^\sharp[\![x \leftarrow e]\!](U^\sharp))$

□

Proof of 6.10. Let us assume that $I \setminus J = \biguplus_{k\in\mathcal{K}} I_k$ and every I_k is a SVI. Let us denote $l' = l \uplus \{x_1, \dots, x_n\}$, $u = u' \uplus \{y_1, \dots, y_p\}$. Consider the two following sets: $\mathcal{L} = \{l' \cup \{y_j\} \mid j \in \{1, \dots, p\}\}$ and $\mathcal{U} = \{u' \setminus \{x_i\} \mid i \in \{1, \dots, n\}\}$. Let a and b be any two distinct elements from $\mathcal{L} \cup \mathcal{U}$, we will show a and b can not be in the same I_k.

- if $a \in \mathcal{L}$ and $b \in \mathcal{L}$, then there exists j and j' such that $j \neq j'$ and $a = l' \cup \{y_j\}$ and $b = l' \cup \{y_{j'}\}$, as $y_j \notin l'$ and $y_{j'} \notin l'$ by definition, we have that $a \cap b = l'$. Therefore if there exists k_0 such that $a \in I_{k_0}$ and $b \in I_{k_0}$ then $l' \in I_{k_0}$ (indeed SVIs are closed under intersection) which is absurd as $l' \in J$.
- if $a \in \mathcal{U}$ and $b \in \mathcal{U}$, then there exists i and i' such that $i \neq i'$ and $a = u' \setminus \{x_i\}$ and $b = u' \setminus \{x_{i'}\}$, as $x_i \in u'$ and $x_{i'} \in u'$ by definition, we have that $a \cup b = u'$. Therefore if there exists k_0 such that $a \in I_{k_0}$ and $b \in I_{k_0}$ then $u' \in I_{k_0}$ (indeed SVIs are closed under union) which is absurd as $u' \in J$.
- if $a \in \mathcal{L}$ and $b \in \mathcal{U}$ then there exists i and j such that $a = l' \cup \{y_j\}$ and $b = u' \setminus \{x_i\}$. As $y_j \notin u'$ and $y_j \neq x_i$, we have $a \cap b \subseteq l'$. Moreover as $x_i \in l'$ we have $a \cup b \supseteq u'$. Therefore if there exists k_0 such that $a \in I_{k_0}$ and $b \in I_{k_0}$ then $a \cap b \in I_{k_0}$ and $a \cup b \in I_{k_0}$ therefore $[a \cap b; a \cup b] \subseteq I_{k_0}$ and $l' \in I_{k_0}$, which is absurd as $u' \in J$.
- the final case is symmetric

Finally we have shown that no two elements from $\mathcal{L} \cup \mathcal{U}$ can be in the same I_k. As $|\mathcal{L} \cup \mathcal{U}| = p + n = |u \setminus u'| + |l' \setminus l|$, we have $|\mathcal{K}| \geqslant |u \setminus u'| + |l' \setminus l|$. □

Proof of 6.11. We have to show: $I \setminus J = \bigcup_{1\leqslant i\leqslant n} U_i \cup \bigcup_{1\leqslant j\leqslant p} U_i$ and that every two SVIs in the union are disjoint.

- Let us prove that $I \setminus J \subseteq \bigcup_{1\leqslant i\leqslant n} U_i \cup \bigcup_{1\leqslant j\leqslant p} U_i$. Let $a \in [l; u] = I$ such that $a \notin [l'; u'] = J$.
  - if $l' \subseteq a$, recall that $l' = l \uplus \{x_1, \dots, x_n\}$
    - if $\forall j \in \{1, p\}, y_j \notin a$ then, as $a \subseteq u$ and $u = u' \uplus \{y_1, \dots, y_p\}$ then $a \subseteq u'$, this is absurd.

        - ⋆ otherwise, $\exists j \in \{1,p\}, y_j \in a$. Let $j_0 = \min\{j \in \{1,p\} \mid y_j \in a\}$, we have $\forall j \leqslant j_0 - 1, y_j \notin a$. Therefore we have $l' \cup y_{j_0} \subseteq a$ and $a \subseteq u \setminus \bigcup_{1 \leqslant k \leqslant j_0-1}\{y_k\}$. Hence $a \in L_{j_0}$ and $a \in \bigcup_{1 \leqslant i \leqslant n} U_i \cup \bigcup_{1 \leqslant j \leqslant p} U_i$
    - otherwise $l' \nsubseteq a$, hence as $l \subseteq a$ there exists $i \in \{1,n\}$ such that $x_i \notin a$. Let $i_0 = \min\{i \in \{1,n\} \mid x_i \notin a\}$. We have $\forall k \leqslant i_0 - 1, x_k \in a$. Hence $l \cup \bigcup_{1 \leqslant k \leqslant i_0-1} \subseteq a$. Moreover $a \subseteq u$ and $x_{i_0} \notin a$, hence $a \in u \setminus \{x_{i_0}\}$. Finally $a \in U_{i_0}$ and $a \in \bigcup_{1 \leqslant i \leqslant n} U_i \cup \bigcup_{1 \leqslant j \leqslant p} U_i$
- Let us prove that $I \setminus J \supseteq \bigcup_{1 \leqslant i \leqslant n} U_i \cup \bigcup_{1 \leqslant j \leqslant p} U_i$. For all $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant p$ we have: $l \subseteq l \cup \bigcup_{1 \leqslant k \leqslant i-1}\{x_k\} \subseteq u$, $l \subseteq u \setminus \{x_i\} \subseteq u$, $l \subseteq l' \cup \{y_j\} \subseteq u$, $l \subseteq u \setminus \bigcup_{1 \leqslant k \leqslant j-1}\{y_k\} \subseteq u$. Hence for all $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant p$ we have $l \subseteq L_j \subseteq u$ and $l \subseteq U_i \subseteq u$, hence $I \supseteq \bigcup_{1 \leqslant i \leqslant n} U_i \cup \bigcup_{1 \leqslant j \leqslant p} U_i$. Therefore we only have to prove that $J \cap (\bigcup_{1 \leqslant i \leqslant n} U_i \cup \bigcup_{1 \leqslant j \leqslant p} U_i) = \emptyset$.
    - Let $1 \leqslant i \leqslant n$, we have $J \cap U_i = [l'; u'] \cap [l \cup \bigcup_{1 \leqslant k \leqslant i-1}\{x_k\}; u \setminus \{x_i\}] = [l' \cup l \cup \bigcup_{1 \leqslant k \leqslant i-1}\{x_k\}; u' \cap (u \setminus \{x_i\})] = [l'; u' \setminus \{x_i\}]$, as $x_i \in l'$ and $x_i \notin u' \setminus \{x_i\}$, we have $[l'; u' \setminus \{x_i\}] = \emptyset = J \cap U_i$.
    - Let $1 \leqslant j \leqslant p$, we have $J \cap L_j = [l'; u'] \cap [l' \cup \{y_j\}; u \setminus \bigcup_{1 \leqslant k \leqslant j-1}\{y_k\}] = [l' \cup l' \cup \{y_j\}; u' \cap (u \setminus \bigcup_{1 \leqslant k \leqslant j-1}\{y_k\})] = [l' \cup \{y_j\}; u']$, as $y_j \in l' \cup \{y_j\}$ and $y_j \notin u'$, we have $[l' \cup \{y_j\}; u'] = \emptyset = J \cap L_j$.

  Therefore we have that $J \cap (\bigcup_{1 \leqslant i \leqslant n} U_i \cup \bigcup_{1 \leqslant j \leqslant p} U_i) = \emptyset$. Finally $I \setminus J \supseteq \bigcup_{1 \leqslant i \leqslant n} U_i \cup \bigcup_{1 \leqslant j \leqslant p} U_i$
- Let us show that $\forall i, i',\ i \neq i' \Rightarrow U_i \cap U_{i'} = \emptyset, \forall j, j',\ j \neq j' \Rightarrow L_j \cap L_{j'} = \emptyset, \forall i, j, U_i \cap L_j = \emptyset$
    - Let $i, i' \in \{1,n\}$, assume *w.l.o.g.* that $i < i'$, $U_i \cap U_i' = [l \cup \bigcup_{1 \leqslant k \leqslant i-1}\{x_k\}; u \setminus \{x_i\}] \cap [l \cup \bigcup_{1 \leqslant k \leqslant i'-1}\{x_k\}; u \setminus \{x_i'\}] = [l \cup \bigcup_{1 \leqslant k \leqslant i'-1}\{x_k\}; u \setminus \{x_i, x_{i'}\}]$. We have $x_i \in \bigcup_{1 \leqslant k \leqslant i'-1}\{x_k\}$ and $x_i \notin u \setminus \{x_i, x_{i'}\}$, hence $[l \cup \bigcup_{1 \leqslant k \leqslant i'-1}\{x_k\}; u \setminus \{x_i, x_{i'}\}] = \emptyset = U_i \cap U_i'$.
    - Let $j, j' \in \{1,p\}$, assume *w.l.o.g.* that $j < j'$, $L_j \cap L_j' = [l' \cup \{y_j\}; u \setminus \bigcup_{1 \leqslant k \leqslant j-1}\{y_k\}] \cap [l' \cup \{y_j'\}; u \setminus \bigcup_{1 \leqslant k \leqslant j'-1}\{y_k\}] = [l' \cup \{y_j, y_j'\}; u \setminus \bigcup_{1 \leqslant k \leqslant j'-1}\{y_k\}]$. We have $y_j \in l' \cup \{y_j, y_{j'}\}$ and $y_j \notin u \setminus \bigcup_{1 \leqslant k \leqslant j'-1}\{y_k\}$, hence $[l' \cup \{y_j, y_j'\}; u \setminus \bigcup_{1 \leqslant k \leqslant j'-1}\{y_k\}] = \emptyset = L_j \cap L_j'$.
    - Let $i \in \{1,n\}$ and $j \in \{1,p\}$, $U_i \cap L_j = [l \cup \bigcup_{1 \leqslant k \leqslant i-1}\{x_k\}; u \setminus \{x_i\}] \cap [l' \cup \{y_j\}; u \setminus \bigcup_{1 \leqslant k \leqslant j-1}\{y_k\}] = [l' \cup \{y_j\}; (u \setminus \{x_i\}) \setminus \bigcup_{1 \leqslant k \leqslant j-1}\{y_k\}]$. We have $x_i \in l' \cup \{y_j\}$ and $x_i \notin (u \setminus \{x_i\}) \setminus \bigcup_{1 \leqslant k \leqslant j-1}\{y_k\}$, therefore $[l' \cup \{y_j\}; (u \setminus \{x_i\}) \setminus \bigcup_{1 \leqslant k \leqslant j-1}\{y_k\}] = \emptyset = U_i \cap L_j$

  Therefore SVIs in $\{U_1, \dots, U_n, L_1, \dots, L_p\}$ are pair-wise disjoint.

□

Proof of 6.11. Let us prove that **itv_incl**$(I, Is) = \texttt{true} \Leftrightarrow I \subseteq \bigcup_{J \in Is} J$. The loop invariant of Algorithm 6.2 is the following: $\textsc{Inv}_m \triangleq \bigcup_{L \in \texttt{rem}} L = I \setminus \bigcup_{1 \leqslant k \leqslant m} I_k$.

- Before loop iterations we have: $\texttt{rem} = \{I\}$. Therefore $\bigcup_{L \in \texttt{rem}} L = I = I \setminus \emptyset$. Hence the invariant holds before the loop iterations.
- Assume that $\textsc{Inv}_n$ holds at the end of the m-th loop. At the end of the $m+1$-st loop iteration we have: $\texttt{rem} = \bigcup_{L \in \texttt{rem}'}$ **itv_diff**(L, I_{m+1}), ($\texttt{rem}'$ denotes the value of $\texttt{rem}$ at the beginning of the m-th loop iteration). Hence $\bigcup_{L \in \texttt{rem}} L = \bigcup_{L \in \texttt{rem}'} \bigcup_{K \in \textbf{itv\_diff}(L, I_{m+1})} K$. Moreover $\bigcup_{K \in \textbf{itv\_diff}(L, I_{m+1})} K = L \setminus I_{m+1}$, hence $\texttt{rem} = \bigcup_{L \in \texttt{rem}'} (L \setminus I_{m+1}) = (\bigcup_{L \in \texttt{rem}'} L) \setminus I_{m+1} = I \setminus \bigcup_{1 \leqslant k \leqslant m+1} I_k$.

Given the invariant, we conclude that **itv_incl**$(I, Is) = \texttt{true} \Leftrightarrow \bigcup_{L \in \texttt{rem}} L = \emptyset \Leftrightarrow I \setminus \bigcup_{1 \leqslant k \leqslant n} I_k = \emptyset \Leftrightarrow I \setminus \bigcup_{J \in Is} J = \emptyset \Leftrightarrow I \subseteq \bigcup_{J \in Is} J$ □

Proof of 6.12. Let us assume that $(\mathfrak{S}_1^{\sharp}{}', \mathfrak{S}_2^{\sharp}{}') =$ **unify_leq**$(\mathfrak{S}_1^{\sharp}, \mathfrak{S}_2^{\sharp}) \wedge (\bigcup_{I \in \textbf{itvs}(\mathfrak{S}_1^{\sharp})} I \subseteq \bigcup_{J \in \textbf{itvs}(\mathfrak{S}_2^{\sharp})} J)$.

- $\gamma_{\wp}(\mathfrak{S}_2^{\sharp}) = \gamma_{\wp}(\mathfrak{S}_2^{\sharp}{}')$ holds trivially as $\mathfrak{S}_2^{\sharp}$ is not modified by **unify_leq**.

- Let $\langle N^\sharp, l, u\rangle \in \mathfrak{S}_1^\sharp$, we will show that $\gamma(\langle N^\sharp, l, u\rangle) \subseteq \gamma_\wp(\mathfrak{S}_1^{\sharp\,\prime})$. Let $\rho \in \gamma(\langle N^\sharp, l, u\rangle)$ and $\mathcal{W} = \mathbf{def}(\rho)$, we have $l \subseteq \mathcal{W} \subseteq u$. As $(\bigcup_{\langle\_, l, u\rangle \in \mathfrak{S}_1^\sharp} [l; u] \subseteq \bigcup_{\langle\_, l', u'\rangle \in \mathfrak{S}_2^\sharp} [l'; u'])$, there exists $\langle\_, l', u'\rangle \in \mathfrak{S}_2^\sharp$ such that $l' \subseteq \mathcal{W} \subseteq u'$. Hence $\mathcal{W} \in [l; u] \cap [l'; u'] \neq \emptyset$, let $[a; b] = [l; u] \cap [l'; u']$, we have $a \subseteq \mathcal{W} \subseteq b$. As $\rho \in \gamma(\langle N^\sharp, l, u\rangle)$, we have $\rho \in (\gamma^u(N^\sharp))_{|\mathcal{W}} = ((\gamma^u(N^\sharp))_{|b})_{|\mathcal{W}} \subseteq (\gamma^b(N^\sharp_{|b}))_{|\mathcal{W}}$. Hence $\rho \in \langle N^\sharp_{|b}, a, b\rangle \in \mathtt{res}$, as $\gamma_\wp(\mathtt{res}) = \bigcup_{U^\sharp \in \mathtt{res}} \gamma(U^\sharp)$, we have $\rho \in \gamma_\wp(\mathtt{res}) = \gamma_\wp(\mathfrak{S}_1^{\sharp\,\prime})$. Therefore we have $\gamma(\langle N^\sharp, l, u\rangle) \subseteq \gamma_\wp(\mathfrak{S}_1^{\sharp\,\prime})$ and as this holds for every $\langle N^\sharp, l, u\rangle \in \mathfrak{S}_1^\sharp$ we can conclude that $\gamma_\wp(\mathfrak{S}_1^\sharp) \subseteq \gamma_\wp(\mathfrak{S}_1^{\sharp\,\prime})$.
- Finally by definition we that that $\forall\langle\_, a, b\rangle \in \mathfrak{S}_1^{\sharp\,\prime}, \exists\langle\_, l, u\rangle \in \mathfrak{S}_1^\sharp, \langle\_, l', u'\rangle \in \mathfrak{S}_2^\sharp, [a; b] = [l; u] \cap [l'; u']$, hence $[a; b] \subseteq [l'; u']$, moreover $\mathfrak{S}_2^{\sharp\,\prime} = \mathfrak{S}_2^\sharp$, which gives us that $\forall\langle\_, a, b\rangle \in \mathfrak{S}_1^{\sharp\,\prime}, \exists\langle\_, l', u'\rangle \in \mathfrak{S}_2^{\sharp\,\prime}, [l; u] \subseteq [l'; u']$.

□

Proof of 6.13. Let $\mathfrak{S}_1^\sharp, \mathfrak{S}_2^\sharp$ be in $\mathfrak{M}_\wp^\sharp$ such that $\mathfrak{S}_1^\sharp \sqsubseteq_\wp \mathfrak{S}_2^\sharp$, we will show that $\gamma(\mathfrak{S}_1^\sharp) \subseteq \gamma(\mathfrak{S}_2^\sharp)$. Let $(\mathfrak{S}_1^{\sharp\,\prime}, \mathfrak{S}_2^{\sharp\,\prime}) = \mathbf{unify\_leq}(\mathfrak{S}_1^\sharp, \mathfrak{S}_2^\sharp)$, as $\forall I \in \mathbf{itvs}(\mathfrak{S}_1^\sharp), \mathbf{itv\_incl}(I, \mathbf{itvs}(\mathfrak{S}_2^\sharp))$, we have $\bigcup_{I \in \mathbf{itvs}(\mathfrak{S}_1^\sharp)} I \subseteq \bigcup_{J \in \mathbf{itvs}(\mathfrak{S}_2^\sharp)} J$. Therefore from Proposition 6.12 we have:

- $\gamma_\wp(\mathfrak{S}_1^\sharp) \subseteq \gamma_\wp(\mathfrak{S}_1^{\sharp\,\prime})$
- $\gamma_\wp(\mathfrak{S}_2^\sharp) = \gamma_\wp(\mathfrak{S}_2^{\sharp\,\prime})$

There remains only to show that $\gamma_\wp(\mathfrak{S}_1^{\sharp\,\prime}) \subseteq \gamma_\wp(\mathfrak{S}_2^{\sharp\,\prime})$. This comes trivially from the soundness of $\sqsubseteq$, indeed we have $\forall U_1^\sharp \in \mathfrak{S}_1^{\sharp\,\prime}, \exists U_2^\sharp \in \mathfrak{S}_2^{\sharp\,\prime}, U_1^\sharp \sqsubseteq U_2^\sharp$, hence $\forall U_1^\sharp \in \mathfrak{S}_1^{\sharp\,\prime}, \exists U_2^\sharp \in \mathfrak{S}_2^{\sharp\,\prime}, \gamma(U_1^\sharp) \subseteq \gamma(U_2^\sharp)$. Please note that we did not need the fact that $\forall\langle\_, l, u\rangle \in \mathfrak{S}_1^{\sharp\,\prime}, \exists\langle\_, l', u'\rangle \in \mathfrak{S}_2^{\sharp\,\prime}, [l; u] \subseteq [l'; u']$ to prove the soundness of the operator. Indeed this fact only improves the precision of the operator. □

Proof of 6.14. For symmetry reasons, it suffices to prove that $\gamma_\wp(\mathfrak{S}_1^\sharp) \subseteq \gamma_\wp(\mathfrak{S}_1^{\sharp\,\prime})$ and that $\forall[l_1; u_1] \in \mathbf{itvs}(\mathfrak{S}_1^{\sharp\,\prime}), \forall[l_2, u_2] \in \mathbf{itvs}(\mathfrak{S}_2^{\sharp\,\prime}), [l_1; u_1] \cap [l_2; u_2] \neq \emptyset \Rightarrow l_1 = l_2 \wedge u_1 = u_2$

- Let us show that $\gamma_\wp(\mathfrak{S}_1^\sharp) \subseteq \gamma_\wp(\mathfrak{S}_1^{\sharp\,\prime})$, let $\rho \in \gamma_\wp(\mathfrak{S}_1^\sharp)$, there exists $\langle N^\sharp, l, u\rangle$ such that $\langle N^\sharp, l, u\rangle \in \mathfrak{S}_1^\sharp$ and $\rho \in \gamma(\langle N^\sharp, l, u\rangle)$. Let us consider the loop iteration at line 5 in Algorithm 6.4 when $\langle N^\sharp, l, u\rangle$ is chosen in $\mathfrak{S}_1^\sharp$. We consider the two following cases:
  - there exists $\langle N^{\sharp\prime}, l', u'\rangle \in \mathfrak{S}_2^\sharp$ such that $\mathbf{def}(\rho) \in [l'; u']$. In such a case $[a; b] \triangleq [l; u] \cap [l', u'] \neq \emptyset$, we then have that: $\langle N^\sharp_{|b}, a, b\rangle \in \mathtt{res}_{1,c} \subseteq \mathtt{res}_{1,c} \cup \mathtt{res}_1 = \mathfrak{S}_1^{\sharp\,\prime}$. As $\mathbf{def}(\rho) \in [l'; u']$ and $\mathbf{def}(\rho) \in [l; u]$, we have $\mathbf{def}(\rho) \in [a; b]$. Moreover $\rho \in \gamma^u(N^\sharp)_{|\mathbf{def}(\rho)} = (\gamma^u(N^\sharp)_{|b})_{|\mathbf{def}(\rho)} \subseteq \gamma^b(N^\sharp_{|b})_{|\mathbf{def}(\rho)}$, hence $\rho \in \gamma(\langle N^\sharp_{|b}, a, b\rangle)$ and $\rho \in \gamma_\wp(\mathfrak{S}_1^{\sharp\,\prime})$.
  - there exists no $\langle N^{\sharp\prime}, l', u'\rangle \in \mathfrak{S}_2^\sharp$ such that $\mathbf{def}(\rho) \in [l'; u']$. We can now prove by a trivial induction on loop 7 that $\exists[x; y] \in \mathtt{rem}, \mathbf{def}(\rho) \in [x; y]$, this comes from the fact that **itv_diff** indeed computes the difference between two SVIs. At the end of the loop there exists $[a; b]$ in $\mathtt{rem}$, such that $\mathbf{def}(\rho) \in [a; b]$, we can show as before that $\rho \in \gamma(\langle N^\sharp_{|b}, a, b\rangle)$ and as $\langle N^\sharp_{|b}, a, b\rangle \in \mathtt{res}_1 \subseteq \mathtt{res}_{1,c} \cup \mathtt{res}_1 = \mathfrak{S}_1^\sharp$, we have that $\rho \in \gamma_\wp(\mathfrak{S}_1^{\sharp\,\prime})$.
- Let us now show that $\forall[l_1; u_1] \in \mathbf{itvs}(\mathfrak{S}_1^{\sharp\,\prime}), \forall[l_2, u_2] \in \mathbf{itvs}(\mathfrak{S}_2^{\sharp\,\prime}), [l_1; u_1] \cap [l_2; u_2] \neq \emptyset \Rightarrow l_1 = l_2 \wedge u_1 = u_2$. Let $\langle N_1^\sharp, l_1, u_1\rangle \in \mathfrak{S}_1^{\sharp\,\prime}$ and $\langle N_2^\sharp, l_2, u_2\rangle \in \mathfrak{S}_2^{\sharp\,\prime}$ such that $[l_1; u_1] \cap [l_2; u_2] \neq \emptyset$. Consider the following cases (with notations from Algorithm 6.4):
  - $\langle N_1^\sharp, l_1, u_1\rangle \in \mathtt{res}_{1,c}$ and $\langle N_2^\sharp, l_2, u_2\rangle \in \mathtt{res}_{2,c}$, we thus have $[l_1; u_1] = [x_1; y_1] \cap [x_2; y_2]$ and $[l_2; u_2] = [x_1'; y_1'] \cap [x_2'; y_2']$ with $[x_1; y_1], [x_1'; y_1'] \in \mathfrak{S}_1^{\sharp\,\prime}$ and $[x_2; y_2], [x_2'; y_2'] \in \mathfrak{S}_2^{\sharp\,\prime}$. As $[l_1; u_1] \cap [l_2; u_2] \neq \emptyset$, we have: $([x_1; y_1] \cap [x_2; y_2]) \cap ([x_1'; y_1'] \cap [x_2'; y_2']) \neq \emptyset$, hence

$([x_1; y_1] \cap [x'_1; y'_1]) \neq \emptyset$, as $\mathfrak{S}^\sharp_1$ is a well-formed abstract element we have: $x_1 = x'_1$ and $y_1 = y'_1$, symmetrically $x_2 = x'_2$ and $y_2 = y'_2$, finally we get: $l_1 = l_2$ and $u_1 = u_2$.

- $\langle N^\sharp_1, l_1, u_1\rangle \in \mathtt{res}_{1,c}$ and $\langle N^\sharp_2, l_2, u_2\rangle \in \mathtt{res}_2$. We can trivially prove the following invariant on the loop of line 17: $\forall [a; b] \in \mathtt{rem},\ [a, b] \cap \bigcup I \in \mathbf{itvs}(\widetilde{\mathfrak{S}^\sharp_1})I = \emptyset$, where $\widetilde{\mathfrak{S}^\sharp_1}$ is the subset of $\mathfrak{S}^\sharp_1$ already visited during the considered loop iteration. Once again this holds because **diff_itv** indeed computes a difference. Using this invariant we know that at the end of the 17-loop iteration we have: $\forall [a; b] \in \mathtt{rem},\ [a, b] \cap \bigcup I \in \mathbf{itvs}(\mathfrak{S}^\sharp_1)I$. Therefore $[l_2; u_2] \cap \bigcup I \in \mathbf{itvs}(\mathfrak{S}^\sharp_1)I = \emptyset$. As there exists (by construction) $[l; u] \in \mathfrak{S}^\sharp_1$ such that $[l_1; u_1] \subseteq [l; u]$, we have $\emptyset = [l_2; u_2] \cap [l; u] \supseteq [l_2; u_2] \cap [l_1; u_1] \neq \emptyset$ which is absurd
- $\langle N^\sharp_1, l_1, u_1\rangle \in \mathtt{res}_1$ and $\langle N^\sharp_2, l_2, u_2\rangle \in \mathtt{res}_{2_c}$. This case is symmetric to the previous one
- $\langle N^\sharp_1, l_1, u_1\rangle \in \mathtt{res}_1$ and $\langle N^\sharp_2, l_2, u_2\rangle \in \mathtt{res}_2$. As in the previous case we have: $[l_1; u_1] \cap \bigcup I \in \mathbf{itvs}(\mathfrak{S}^\sharp_2)I = \emptyset$ and $[l_2; u_2] \cap \bigcup I \in \mathbf{itvs}(\mathfrak{S}^\sharp_1)I = \emptyset$ as there exists $[l; u] \in \mathfrak{S}^\sharp_1$ such that $[l_1; u_1] \subseteq [l; u]$, we have $\emptyset = [l_2; u_2] \cap [l; u] \supseteq [l_2; u_2] \cap [l_1; u_1] \neq \emptyset$ which is absurd

This proves the assertion.

□

Proof of 6.15.

- We start by proving the soundness of $\sqcup_\wp$. Let $\mathfrak{S}^\sharp_1$ and $\mathfrak{S}^\sharp_2$ be in $\mathfrak{M}^\sharp_\wp$. Let $\rho \in \gamma_\wp(\mathfrak{S}^\sharp_1) \cup \gamma_\wp(\mathfrak{S}^\sharp_2)$, *w.l.o.g.* we can assume that $\rho \in \gamma_\wp(\mathfrak{S}^\sharp_1)$. With notations from Definition 6.24 we have: $(\mathfrak{S}^{\sharp\,\prime}_1, \mathfrak{S}^{\sharp\,\prime}_2) = \mathbf{unify}(\mathfrak{S}^\sharp_1, \mathfrak{S}^\sharp_2)$. From the soundness of the **unify** operator (see Proposition 6.14) we deduce that $\rho \in \gamma_\wp(\mathfrak{S}^{\sharp\,\prime}_1)$. This implies that there exists $\langle N^\sharp_1, l, u\rangle \in \mathfrak{S}^{\sharp\,\prime}_1$ such that $\rho \in \gamma(\langle N^\sharp_1, l, u\rangle)$. Therefore $[l; u] \in \mathbf{itvs}(\mathfrak{S}^{\sharp\,\prime}_1)$, hence:
  - either there exists $\langle N^\sharp_2, l, u\rangle \in \mathfrak{S}^{\sharp\,\prime}_2$, in such a case, by soundness of the $\sqcup$ operator (see Proposition 6.5) we have: $\rho \in \gamma(\langle N^\sharp_1, l, u\rangle \sqcup \langle N^\sharp_2, l, u\rangle)$ and as $\langle N^\sharp_1, l, u\rangle \sqcup \langle N^\sharp_2, l, u\rangle \in \mathfrak{S}^\sharp_1 \sqcup_\wp \mathfrak{S}^\sharp_2$ we can conclude that $\rho \in \gamma_\wp(\mathfrak{S}^\sharp_1 \sqcup_\wp \mathfrak{S}^\sharp_2)$
  - or there exists no $\langle N^\sharp_2, l, u\rangle \in \mathfrak{S}^{\sharp\,\prime}_2$. In this case, we have that $\langle N^\sharp_1, l, u\rangle \in \mathfrak{S}^\sharp_1 \sqcup_\wp \mathfrak{S}^\sharp_2$, hence $\rho \in \gamma_\wp(\mathfrak{S}^\sharp_1 \sqcup_\wp \mathfrak{S}^\sharp_2)$.

  We conclude that $\gamma_\wp(\mathfrak{S}^\sharp_1) \cup \gamma_\wp(\mathfrak{S}^\sharp_2) \subseteq \gamma_\wp(\mathfrak{S}^\sharp_1 \sqcup_\wp \mathfrak{S}^\sharp_2)$
- Let us now prove the soundness of $\sqcap_\wp$. Let $\mathfrak{S}^\sharp_1$ and $\mathfrak{S}^\sharp_2$ be in $\mathfrak{M}^\sharp_\wp$. Let $\rho \in \gamma_\wp(\mathfrak{S}^\sharp_1) \cup \gamma_\wp(\mathfrak{S}^\sharp_2)$. With notations from Definition 6.24 we have: $(\mathfrak{S}^{\sharp\,\prime}_1, \mathfrak{S}^{\sharp\,\prime}_2) = \mathbf{unify}(\mathfrak{S}^\sharp_1, \mathfrak{S}^\sharp_2)$. From the soundness of the **unify** operator (see Proposition 6.14) we deduce that $\rho \in \gamma_\wp(\mathfrak{S}^{\sharp\,\prime}_1)$ and $\rho \in \gamma_\wp(\mathfrak{S}^{\sharp\,\prime}_2)$. This implies that there exists $\langle N^\sharp_1, l_1, u_1\rangle \in \mathfrak{S}^{\sharp\,\prime}_1$ such that $\rho \in \gamma(\langle N^\sharp_1, l_1, u_1\rangle)$ and there exists $\langle N^\sharp_2, l_2, u_2\rangle \in \mathfrak{S}^{\sharp\,\prime}_2$ such that $\rho \in \gamma(\langle N^\sharp_2, l_2, u_2\rangle)$. As we have $\mathbf{def}(\rho) \in [l_1; u_1]$ and $\mathbf{def}(\rho) \in [l_2; u_2]$ we know that $[l_1; u_1] \cap [l_2; u_2] \neq \emptyset$. From Proposition 6.14 we can conclude that $l_1 = l_2$ and $u_1 = u_2$. This implies that $[l; u] \in \mathbf{itvs}(\mathfrak{S}^{\sharp\,\prime}_1 \cap \mathbf{itvs}(\mathfrak{S}^{\sharp\,\prime}_2)$. Finally we have that $\langle N^\sharp_1, l, u\rangle \sqcap \langle N^\sharp_2, l, u\rangle \in \mathfrak{S}^\sharp_1 \sqcap_\wp \mathfrak{S}^\sharp_2$, by soundness of $\sqcap$ (see Proposition 6.7) we know that: $\rho \in \gamma(\langle N^\sharp_1, l, u\rangle \sqcap \langle N^\sharp_2, l, u\rangle)$, hence $\rho \in \gamma_\wp(\mathfrak{S}^\sharp_1 \sqcap_\wp \mathfrak{S}^\sharp_2)$

□

Proof of 6.16. The soundness of the **merge** operator relies solely on the fact that $\gamma(U^\sharp_1) \cup \gamma(U^\sharp_2) \subseteq \gamma(U^\sharp_1 \sqcup U^\sharp_2)$ (which is given by soundness of $\sqcup$, see Proposition [?]), indeed $\mathbf{merge}(\mathfrak{S}^\sharp)$ is obtained from $\mathfrak{S}^\sharp$ by iteratively applying this transformation. □

Proof of 6.17.

- We will not provide the details of the soundness proof. Indeed $\mathtt{merg}_1$ (resp. $\mathtt{merg}_2$) was built from $\mathfrak{S}^\sharp_1$ (resp. $\mathfrak{S}^\sharp_2$) by successive **merge** operation, which we proved sound in Proposition 6.16. There are then two cases to consider: (1) Every SVI of a monomial of $\mathfrak{S}^\sharp_2$ is contained in at least one SVI of a monomial of $\mathfrak{S}^\sharp_1$. In this case the result is the union of the pair-wise widening of every shared SVIs, and monomials from $\mathfrak{S}^\sharp_1$ which do not intersect a monomial from $\mathfrak{S}^\sharp_2$ (there are no monomials in $\mathfrak{S}^\sharp_2$ that do intersect a monomial from $\mathfrak{S}^\sharp_1$). Therefore the soundness of this case is given by the soundness of the underlying $\triangledown$ operator, which is given by Proposition 6.8. (2) In the other case, the result of the widening computation is $\top_\wp$, which is always sound.

- Let us now prove the stabilizing property of the $\triangledown_\wp$ operator. Let us consider a sequence $(\mathfrak{S}^\sharp_n)_{n\in\mathbb{N}}$, we want to show that the sequence $(\mathfrak{S}^\sharp_n{}')_{n\in\mathbb{N}}$, defined by $\mathfrak{S}^{\sharp}_0{}' = \mathfrak{S}^\sharp_0$ and $\mathfrak{S}^{\sharp}_{n+1}{}' = \mathfrak{S}^\sharp_n{}' \triangledown_\wp \mathfrak{S}^\sharp_n$, converges.
  - By soundness we know that: if there exists some $k \in \mathbb{N}$ such that $\mathfrak{S}^{\sharp}_k{}' = \top_\wp$, then for every $p \geqslant k$, $\mathfrak{S}^{\sharp}_k{}' = \top_\wp$, which ensures convergence. Therefore we assume for the rest of the proof that for all $k \in \mathbb{N}$ we have $\mathfrak{S}^{\sharp}_k{}' \neq \top_\wp$.
  - Given two elements $\mathfrak{S}^\sharp$ and $\mathfrak{S}^{\sharp\,\prime}$ such that $\mathfrak{S}^\sharp \triangledown_\wp \mathfrak{S}^{\sharp\,\prime} \neq \top_\wp$ we have $|\mathfrak{S}^\sharp \triangledown_\wp \mathfrak{S}^{\sharp\,\prime}| < |\mathfrak{S}^\sharp|$. Indeed with notations from Algorithm 6.7: $|\mathfrak{S}^\sharp \triangledown_\wp \mathfrak{S}^{\sharp\,\prime}| = |\mathtt{res}| = |\mathtt{merg}_1|$. As $|\mathtt{merg}_1|$ is obtained from $\mathfrak{S}^\sharp$ by successive merge operations, we deduce that $|\mathfrak{S}^\sharp \triangledown_\wp \mathfrak{S}^{\sharp\,\prime}| = |\mathtt{merg}|_1 \leqslant |\mathfrak{S}^\sharp|$. From this we conclude that $(|\mathfrak{S}^\sharp_n|)_{n\in\mathbb{N}}$ converges, let N be such that $\forall k \geqslant N, |\mathfrak{S}^\sharp_k| = |\mathfrak{S}^\sharp_N|$. Moreover we trivially have that if $|\mathfrak{S}^\sharp| = |\mathfrak{S}^\sharp \triangledown_\wp \mathfrak{S}^{\sharp\,\prime}|$ then $\mathbf{itv}(\mathfrak{S}^\sharp) = \mathbf{itv}(\mathfrak{S}^\sharp \triangledown_\wp \mathfrak{S}^{\sharp\,\prime})$. Indeed SVIs from $\mathfrak{S}^\sharp \triangledown_\wp \mathfrak{S}^{\sharp\,\prime}$ are obtained by merging SVIs from $\mathfrak{S}^\sharp$, as there are the same number of SVIs, none have been merged and $\mathbf{itv}(\mathfrak{S}^\sharp) = \mathbf{itv}(\mathfrak{S}^\sharp \triangledown_\wp \mathfrak{S}^{\sharp\,\prime})$. Therefore we have $\forall k \geqslant N, \mathbf{itvs}(\mathfrak{S}^\sharp_k) = \mathbf{itvs}(\mathfrak{S}^\sharp_N)$. Finally using notations introduced in Proposition **??**, we can show that $\forall [a;b] \in \mathbf{itvs}(\mathfrak{S}^\sharp_N), \forall k \geqslant N, \mathfrak{S}^{\sharp}_{k+1}{}'[a;b] = \mathfrak{S}^{\sharp}_k{}'[a;b] \vee \exists \mathfrak{S}^\sharp, \mathfrak{S}^{\sharp}_{k+1}{}'[a;b] = \mathfrak{S}^{\sharp}_k{}'[a;b] \triangledown \mathfrak{S}^\sharp$. We conclude by the stabilizing property of $\triangledown$ (given by Proposition 6.8).

□

Proof of 6.18. Let us show that Algorithm 6.8 enjoys the following loop invariant: $\bigcup_{U^\sharp\in S}\gamma(U^\sharp) \subseteq \bigcup_{U^\sharp\in \mathtt{res}}\gamma(U^\sharp)$.

- This holds trivially initially as $\mathtt{res} = S$
- Let us assume that at the beginning of some loop we have: $\bigcup_{U^\sharp\in S}\gamma(U^\sharp) \subseteq \bigcup_{U^\sharp\in \mathtt{res}}\gamma(U^\sharp)$. We note $\overline{\mathtt{res}}$ the value of $\mathtt{res}$ at the end of the loop. Let us consider $\rho \in \bigcup_{U^\sharp\in S}\gamma(U^\sharp)$. By induction hypothesis, there exists $U^\sharp \in \mathtt{res}$ such that $\rho \in \gamma(U^\sharp)$.
  - If $U^\sharp \neq \langle N^\sharp, l, u\rangle$ and $U^\sharp \neq \langle N^{\sharp\prime}, l', u'\rangle$ then $U^\sharp \in \overline{\mathtt{res}}$. Hence the invariant trivially holds.
  - Otherwise we can assume *w.l.o.g.* that $U^\sharp = \langle N^\sharp, l, u\rangle$.
    - If $\mathbf{def}(\rho) \in [a;b]$ then by definition of γ we have that $\rho \in \gamma^u(N^\sharp)_{|\mathbf{def}(\rho)}$. As we have already shown before, as $\mathbf{def}(\rho) \subseteq b$, we have $\rho \in \gamma^b(N^\sharp_{|b})_{|\mathbf{def}(\rho)}$, and $\rho \in \gamma^b(N^\sharp_{|b} \sqcup^b N^{\sharp\prime}_{|b})_{|\mathbf{def}(\rho)}$. Hence $\rho \in \gamma(\langle N^\sharp_{|b} \sqcup^b N^{\sharp\prime}_{|b}, a, b\rangle)$ as $\langle N^\sharp_{|b} \sqcup^b N^{\sharp\prime}_{|b}, a, b\rangle \in \overline{\mathtt{res}}$ we can conclude that the invariant holds.
    - If $\mathbf{def}(\rho) \notin [a;b]$, then there exists $[c;d] \in \mathbf{itv\_diff}([l;u],[a;b])$ such that $\mathbf{def}(\rho) \in [c;d]$. Hence there exists $[c;d] \in \mathtt{diff}_1$ such that $\mathbf{def}(\rho) \in [c;d]$. As we have already shown before, as $\mathbf{def}(\rho) \subseteq d$, we have $\rho \in \gamma^d(N^\sharp_{|d})_{|\mathbf{def}(\rho)}$. Hence we have $\rho \in \gamma(\langle N^\sharp_{|d}, c, d\rangle)$. As $\langle N^\sharp_{|d}, c, d\rangle \in \mathtt{res}$, we can conclude that the invariant holds.

We have proven that the loop invariant holds. At the end of the loop we have $\bigcup_{U^\sharp\in S}\gamma(U^\sharp) \subseteq \bigcup_{U^\sharp\in \mathtt{res}}\gamma(U^\sharp)$. Moreover the loop guard ensures that no two SVIs in $\mathtt{res}$ intersects, therefore $\mathtt{res} \in \mathfrak{M}^\sharp_\wp$ and $\bigcup_{U^\sharp\in \mathtt{res}}\gamma(U^\sharp) = \gamma_\wp(\mathtt{res})$. Finally $\bigcup_{U^\sharp\in S}\gamma(U^\sharp) \subseteq \gamma_\wp(\mathbf{wf}_p(S))$. □

Proof of 6.19. We only provide proof for $\mathbb{S}^\sharp[\![\mathtt{Assume(c)}]\!]_\wp$ as both definitions are identical. We prove that $\forall\mathfrak{S}^\sharp, \mathbb{S}[\![\mathtt{Assume(c)}]\!](\gamma_\wp(\mathfrak{S}^\sharp)) \subseteq \gamma_\wp(\mathbb{S}^\sharp[\![\mathtt{Assume(c)}]\!]_\wp(\mathfrak{S}^\sharp))$.

$$\begin{aligned}
\mathbb{S}[\![\mathtt{Assume(c)}]\!](\gamma_\wp(\mathfrak{S}^\sharp)) &= \mathbb{S}[\![\mathtt{Assume(c)}]\!]\Big(\bigcup_{U^\sharp\in\mathfrak{S}^\sharp} \gamma(U^\sharp)\Big) && \text{By definition}\\
&= \bigcup_{U^\sharp\in\mathfrak{S}^\sharp} \mathbb{S}[\![\mathtt{Assume(c)}]\!](\gamma(U^\sharp)) && \mathbb{S}[\![\mathtt{Assume(c)}]\!] \text{ is a join-morphism}\\
&\subseteq \bigcup_{U^\sharp\in\mathfrak{S}^\sharp} \gamma(\mathbb{S}^\sharp[\![\mathtt{Assume(c)}]\!](U^\sharp)) && \text{By soundness of } \mathbb{S}^\sharp[\![\mathtt{Assume(c)}]\!]\\
&\subseteq \bigcup_{V^\sharp\in\{\mathbb{S}^\sharp[\![\mathtt{Assume(c)}]\!](U^\sharp)\mid U^\sharp\in\mathfrak{S}^\sharp\}} \gamma(V^\sharp) && \text{Variable change}\\
&\subseteq \gamma_\wp(\mathbf{wf}_p(\{\mathbb{S}^\sharp[\![\mathtt{Assume(c)}]\!](U^\sharp) \mid U^\sharp \in \mathfrak{S}^\sharp\}) && \text{By Proposition 6.18}\\
&\subseteq \mathbb{S}^\sharp[\![\mathtt{Assume(c)}]\!]_\wp(\mathfrak{S}^\sharp) && \text{By definition}
\end{aligned}$$

□

Proof of 7.17. We want to prove that tree automata widening is sound and stabilizes infinite sequences. Consider two tree regular languages $U^\sharp$ and $V^\sharp$, which unique minimal deterministic automaton are $\mathcal{A}$ and $\mathcal{B}$. We have $U^\sharp \triangledown V^\sharp = \mathcal{L}([A \cup B]_w) \supseteq \mathcal{L}(A \cup B) = \mathcal{L}(A) \cup \mathcal{L}(B) = U^\sharp \cup V^\sharp$ which proves soundness. Let us now consider a sequence $(U^\sharp_i)_{i\geqslant 0}$ and $(V^\sharp_i)_{i\geqslant 0}$ defined by: $V^\sharp_0 = U^\sharp_0$ and $V^\sharp_{i+1} = V^\sharp_i \triangledown U^\sharp_i$. We have that $\forall i \geqslant 0, \mathcal{B}_i$ has at most w states. The set of tree automatas over $\mathcal{R}$ with at most w states is finite, moreover by soundness we have $\mathcal{B}_i \subseteq \mathcal{B}_{i+1}$, hence sequence $(\mathcal{B}_i)_{i\geqslant 0}$ stabilizes and sequence $(V^\sharp_i)_{i\geqslant 0}$ stabilizes. □

Proof of 7.19. Let us show that $\forall\mathcal{T} \in \wp(T_{\mathbb{I}}(\mathcal{R})), \forall\Gamma \in \mathcal{C}(\mathcal{R}), \mathcal{T} \subseteq \gamma_{\mathcal{C}(\mathcal{R})}(\Gamma) \Leftrightarrow \alpha_{\mathcal{C}(\mathcal{R})}(\mathcal{T}) \subseteq \Gamma$. Let $\mathcal{T} \in \wp(T_{\mathbb{I}}(\mathcal{R}))$ and $\Gamma \in \mathcal{C}(\mathcal{R})$, we have

$$\begin{aligned}
\mathcal{T} \subseteq \gamma_{\mathcal{C}(\mathcal{R})}(\Gamma) &\Leftrightarrow \mathcal{T} \subseteq \{t \in T_{\mathbb{I}}(\mathcal{R}) \mid t_{\mathbb{I}} \in \Gamma\}\\
&\Leftrightarrow \forall t \in \mathcal{T}, t_{\mathbb{I}} \in \Gamma\\
&\Leftrightarrow \alpha_{\mathcal{C}(\mathcal{R})}(\mathcal{T}) \subseteq \Gamma
\end{aligned}$$

□

Proof of 7.20. We assume given the soundness of the unification operator. Hence: $\forall U^\sharp, V^\sharp, (U^\sharp_1, V^\sharp_1) = \textbf{unify\_subset}(U^\sharp, V^\sharp) \Rightarrow U^\sharp \sqsubseteq U^\sharp_1 \wedge V^\sharp = V^\sharp_1$ (N1). Moreover we assume that $(U^\sharp_1, V^\sharp_1) = \textbf{unify\_subset}(U^\sharp, V^\sharp)$ all partitions in $U^\sharp$ intersects a partition in $V^\sharp$ (N2). Let us now prove that γ_1 is monotonic in $\sqsubseteq_{\mathcal{C}^\sharp}$, meaning that: $\forall U^\sharp, V^\sharp, U^\sharp \sqsubseteq_{\mathcal{C}^\sharp} V^\sharp \Rightarrow \gamma_1(U^\sharp) \subseteq \gamma_1(V^\sharp)$. Thanks to (N1) we only have to prove that: if $\langle s, \mathfrak{p}, N^\sharp\rangle$ and $\langle s', \mathfrak{p}', N^{\sharp\prime}\rangle$ are such that: $s \subseteq s' \wedge \forall b \in \mathfrak{p}', (b \subseteq s^c \vee \exists! a \in \mathfrak{p}, b \cap s = a) \wedge N^\sharp \sqsubseteq N^{\sharp\prime}[\phi]$, where ϕ is the renaming from $\mathfrak{p}'$ into $\mathfrak{p}$ that renames b in a when such an a exists (N3) then: $\gamma_1(\langle s, \mathfrak{p}, R^\sharp\rangle) \subseteq \gamma_1(\langle s', \mathfrak{p}', R^{\sharp\prime}\rangle)$. Therefore let us assume that $\rho \in \gamma_1(\langle s, \mathfrak{p}, R^\sharp\rangle)$ and assume every conditions (N3). By definition of γ_1 we get that $\rho \in \gamma[\uparrow \mathfrak{p}](R^\sharp)$ and $\mathbf{def}(\rho) \subseteq s$. We therefore have (by definition of $\gamma[\uparrow \mathfrak{p}](R^\sharp)$) that: $\downarrow_{\uparrow\mathfrak{p}} (\rho) \subseteq \gamma(R^\sharp)$.

- We have $\mathbf{def}(\rho) \subseteq s \subseteq s'$
- Let us now show that $\rho \in \gamma[\uparrow \mathfrak{p}'](R^{\sharp\prime})$. We only need to prove that $\downarrow_{\uparrow\mathfrak{p}'} (\rho) \subseteq \gamma(R^{\sharp\prime})$. Therefore let $g \in\downarrow_{\uparrow\mathfrak{p}'} (\rho)$. We have $g \in \mathrm{Reg}_n \nrightarrow \mathbb{I}$, moreover by definition of $\downarrow_{\uparrow\mathfrak{p}'} (\rho)$, we have $v' \in \mathbf{def}(g) \Leftrightarrow\uparrow \mathfrak{p}'(v') \cap \mathbf{def}(\rho) \neq \emptyset \Leftrightarrow v' \in \mathfrak{p}' \wedge v' \cap \mathbf{def}(\rho) \neq \emptyset$. Moreover as $\mathbf{def}(\rho) \subseteq s \subseteq s'$ we have that g is undefined on partitions contained in s^c. Therefore g is only defined on partitions where ϕ is defined. Hence let us consider $g[\phi]$ such that

$g[\phi](a) = g(b)$ when $\phi(b) = a$ and undefined otherwise. We have $\mathbf{def}(g[\phi]) = \{a \mid \exists b,\ \phi(b) = a \wedge g$ is defined on $b\} = \{\phi(b) \mid b \in \mathbf{def}(g)\} = \{\phi(b) \mid b \in \mathfrak{p}' \wedge b \cap \mathbf{def}(\rho) \neq \emptyset\} \subseteq \mathfrak{p}$ (N4). Let us show that $g[\phi] \in \downarrow_{\uparrow \mathfrak{p}} (\rho)$. Let $\nu' \in \mathrm{Reg}_n$:

- If $\uparrow \mathfrak{p}(\nu') \cap \mathbf{def}(\rho) = \emptyset$ then let us assume that $g[\phi]$ is defined on ν', then (from (N4)) $\nu' = \phi(b)$ and $\uparrow \mathfrak{p}(\nu') = \nu'$ (as $\nu' \in \mathfrak{p}'$) with $b \in \mathfrak{p}' \wedge b \cap \mathbf{def}(\rho) \neq \emptyset$, moreover $b \cap s = \nu'$ and $\mathbf{def}(\rho) \subseteq s$, therefore we have: $\nu' \cap \mathbf{def}(\rho) \neq \emptyset$ and thus $\uparrow \mathfrak{p}'(\nu') \cap \mathbf{def}(\rho) \neq \emptyset$ which is absurd, hence $g[\phi]$ is undefined on ν'.
- If $\uparrow \mathfrak{p}(\nu') \cap \mathbf{def}(\rho) \neq \emptyset$, we have $\nu' \in \mathfrak{p}$. Thanks to (N2) we also have that: $\exists b \in \mathfrak{p}', \nu' \cap b \neq \emptyset$, we have necessarily that $\phi(b) = \nu'$ and $b \cap s = \nu'$, hence $b \cap \mathbf{def}(\rho) \neq \emptyset$ and g is defined on b. Thus there exists $\nu \in \mathcal{W}(\mathcal{R})$ such that $\nu \in (\uparrow \mathfrak{p}')(b) \cap \mathbf{def}(\rho)$, $g(b) = \rho(\nu)$ (this comes from $g \in \downarrow_{\uparrow \mathfrak{p}'} (\rho)$). We have $\uparrow \mathfrak{p}'(b) = b$ and $b \cap \mathbf{def}(\rho) = \nu' \cap \mathbf{def}(\rho)$ (as $\mathbf{def}(\rho) \subseteq s$), hence $\nu \in \nu'$ and $\nu \in \uparrow \mathfrak{p}(\nu') \cap \mathbf{def}(\rho)$. Moreover $g[\phi](\nu') = g(b) = \rho(\nu')$. Therefore there exists $\nu' \in \uparrow \mathfrak{p}(\nu') \cap \mathbf{def}(\rho)$ such that $g[\phi](\nu') = \rho(\nu')$.

From the two previous point we get that $g[\phi] \in \downarrow_{\uparrow \mathfrak{p}} (\rho)$, hence $g[\phi] \in \gamma(R^\sharp)$, by soundness of the underlying domain and by (N3) we have: $g[\phi] \in \gamma(R^{\sharp\prime}[\phi])$, hence $g \in \gamma(R^{\sharp\prime})$. As this holds for every $g \in \downarrow_{\uparrow \mathfrak{p}'} (\rho)$, we have that: $\downarrow_{\uparrow \mathfrak{p}'} (\rho) \subseteq \gamma(R^{\sharp\prime})$.

Finally from the last two points we get that $\rho \in \gamma_1(s', \mathfrak{p}', R^{\sharp\prime})$. Hence $\gamma_1(\langle s, \mathfrak{p}, R^\sharp \rangle) \subseteq \gamma_1(\langle s', \mathfrak{p}', R^{\sharp\prime} \rangle)$ □

Proof of 7.21. Here again we assume the soundness of the **unify_join** operator: $\forall U^\sharp, V^\sharp$, if $U_1^\sharp, V_1^\sharp =$ **unify_join**$(U^\sharp, V^\sharp)$, then $\gamma(U^\sharp) \subseteq \gamma(U_1^\sharp)$ and $\gamma(V^\sharp) \subseteq \gamma(V_1^\sharp)$. Therefore in order to prove the soundness of the $\sqcup_{\mathcal{C}^\sharp(\mathcal{R})}$ operator we only have to prove that: let $(\langle s, \mathfrak{p}, R^\sharp \rangle, \langle s', \mathfrak{p}', R^{\sharp\prime} \rangle) =$ **unify_join**$(U^\sharp, V^\sharp)$, let $\mathfrak{c} = \mathfrak{p} \cup \mathfrak{p}'$, let $u^o = \{e \in \mathfrak{p} \mid e \subseteq s'^c\}$, let $\nu^o = \{e \in \mathfrak{p}' \mid e \subseteq s^c\}$ then if $\rho \in \gamma(\langle s, \mathfrak{p}, R^\sharp \rangle)$ then $\rho \in \gamma(\langle s \cup s', \mathfrak{c} \cup u^o \cup \nu^o, R^\sharp_{|\mathfrak{c} \cup u^o} \sqcup R^{\sharp\prime}_{|\mathfrak{c} \cup \nu^o} \rangle)$. Indeed the operator definition is symmetric therefore we only prove the result for $U^\sharp$. Let $\rho \in \gamma_1(\langle s, \mathfrak{p}, R^\sharp \rangle)$, we have by definition that $\mathbf{def}(\rho) \subseteq s$ hence $\mathbf{def}(\rho) \subseteq s \cup s'$. Moreover we have $\downarrow_{\uparrow \mathfrak{p}} (\rho) \subseteq \gamma(R^\sharp)$. Let us prove that we have: $\downarrow_{\uparrow (\mathfrak{c} \cup u^o \cup \nu^o)} (\rho) \subseteq \gamma(R^\sharp_{|\mathfrak{c} \cup u^o} \sqcup R^{\sharp\prime}_{|\mathfrak{c} \cup \nu^o})$. By soundness of $\sqcup$ we only need to prove that: $\downarrow_{\uparrow (\mathfrak{c} \cup u^o \cup \nu^o)} (\rho) \subseteq \gamma(R^\sharp_{|\mathfrak{c} \cup u^o})$. Therefore let $g \in \downarrow_{\uparrow (\mathfrak{c} \cup u^o \cup \nu^o)} (\rho)$. We have $\mathbf{def}(g) = \{\nu' \mid \nu' \in \mathfrak{c} \cup u^o \cup \nu^o \wedge \nu' \cap \mathbf{def}(\rho) \neq \emptyset\}$. Moreover if $\nu' \in \nu^o$ then $\nu' \subseteq s^c$, and as $\mathbf{def}(\rho) \subseteq s$ we have that $\nu' \cap \mathbf{def}(\rho) = \emptyset$. Hence $\mathbf{def}(g) = \{\nu' \mid \nu' \in \mathfrak{c} \cup u^o \wedge \nu' \cap \mathbf{def}(\rho) \neq \emptyset\}$, moreover $\forall \nu', \nu' \in \mathbf{def}(g), \exists \nu, \nu \in \nu' \cap \mathbf{def}(f) \wedge g(\nu') = \rho(\nu)$ (N1). Let us now define $h : \mathrm{Reg}_n \nrightarrow \mathbb{I}$ in the following way: if $\nu' \in \mathbf{def}(g)$ then $h(\nu') = g(\nu')$, if there exists some $\nu \in \uparrow \mathfrak{p}(\nu') \cap \mathbf{def}(\rho)$ then $h(\nu') = f(\nu)$ otherwise h is undefined. Note that there can be several such h, we choose one of them. Let us show that $h \in \downarrow_{\uparrow \mathfrak{p}} (\rho)$: let $\nu' \in \mathrm{Reg}_n$,

- if $\uparrow \mathfrak{p}(\nu') \cap \mathbf{def}(\rho) = \emptyset$ then if h is defined, by definition it is either because:
  - $\uparrow \mathfrak{p}(\nu') \cap \mathbf{def}(\rho) \neq \emptyset$, which is absurd,
  - or because g is defined on ν'. Hence we have that $\nu' \in \mathfrak{c} \cup u^o$ and $\nu' \cap \mathbf{def}(\rho) \neq \emptyset$, which is also absurd.

  Hence h is undefined.
- if $\uparrow \mathfrak{p}(\nu') \cap \mathbf{def}(\rho) \neq \emptyset$ then:
  - if $\nu' \in \mathbf{def}(g)$, then $\uparrow \mathfrak{p}(\nu') = \nu'$ as $\mathbf{def}(g) \subseteq \mathfrak{c} \cup u^o \subseteq \mathfrak{p}$ we have (due to (N1)). Moreover there exists some $\nu \in \nu' \cap \mathbf{def}(f)$ such that $g(\nu') = \rho(\nu)$, hence $h(\nu') = \rho(\nu)$
  - otherwise by construction of h there exists some $\nu \in \uparrow \mathfrak{p}(\nu') \cap \mathbf{def}(\rho)$ such that $h(\nu') = \rho(\nu)$.

Finally we have $h \in \downarrow_{\uparrow \mathfrak{p}} (\rho)$ and therefore $h \in \gamma(R^\sharp)$. Moreover if $\nu' \in \mathbf{def}(h) \cap (\mathfrak{c} \cup u^o)$ we have $\nu' \in \mathbf{def}(g)$ or $\uparrow \mathfrak{p}(\nu') \cap \mathbf{def}(\rho) \neq \emptyset$. In the latter case as $\mathfrak{p} = \mathfrak{c} \cup u^i \cup u^o$, we have $\uparrow \mathfrak{p}(\nu) \cap \mathbf{def}(\rho) = \uparrow (c \cup u^o \cup \nu^o)(\nu) \cap \mathbf{def}(\rho) \neq \emptyset$, and therefore $\nu' \in \mathbf{def}(g)$. Hence $h_{|\mathfrak{c} \cup u^o} = g$, and as $h \in \gamma(R^\sharp)$ we have: $g \in \gamma(R^\sharp_{|\mathfrak{c} \cup u^o})$. We thus conclude that $\downarrow_{\uparrow (\mathfrak{c} \cup u^o \cup \nu^o)} (\rho) \subseteq \gamma(R^\sharp_{|\mathfrak{c} \cup u^o})$ hence

$\downarrow_{\uparrow(\mathfrak{c} \cup u^o \cup v^o)}(\rho) \subseteq \gamma(R^{\sharp}_{|\mathfrak{c} \cup u^o} \sqcup R^{\sharp\prime}_{|\mathfrak{c} \cup v^o})$. Thus giving us that $\rho \in \gamma(\langle s \cup s', \mathfrak{c} \cup u^o \cup v^o, R^{\sharp}_{|\mathfrak{c} \cup u^o} \sqcup R^{\sharp\prime}_{|\mathfrak{c} \cup v^o}\rangle)$ □

Bibliography

[AGH06] Xavier Allamigeon, Wenceslas Godard, and Charles Hymans. Static analysis of string manipulations in critical embedded C programs. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2006.

[All08] Xavier Allamigeon. Non-disjunctive numerical domain for array predicate abstraction. In *ESOP*, volume 4960, pages 163–177. Springer, 2008.

[BBY17] S, Blazy, D. Bühler, and B. Yakobowski. Structuring abstract interpreters through state and value abstractions. In *Verification, Model Checking, and Abstract Interpretation*, pages 112–130. Springer, 2017.

[BCC$^+$10] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace*, number 2010-3385, pages 1–38. AIAA, Apr. 2010.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[BH19] Rémy Boutonnet and Nicolas Halbwachs. Disjunctive relational abstract interpretation for interprocedural program analysis. In *VMCAI*, volume 11388 of *Lecture Notes in Computer Science*, pages 136–159. Springer, 2019.

[BHRV06] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *Proc. of SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 52–70. Springer, 2006.

[BHV04] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Abstract regular model checking. In Rajeev Alur and Doron A. Peled, editors, *Proc. of CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2004.

[BHZ08] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[BHZ09] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Exact join detection for convex polyhedra and other numerical abstractions. *CoRR*, abs/0904.1783, 2009.

[BNSV14] G. Brat, J. A. Navas, N. Shi, and A. Venet. Ikos: A framework for static analysis based on abstract interpretation. In *Software Engineering and Formal Methods*, pages 271–277. Springer, 2014.

[Bot18] Vincent Botbol. *Analyse statique de programmes concurrents avec variables numériques*. PhD thesis, Sorbonne Université, 2018.

[Bou92a] François Bourdoncle. Abstract interpretation by dynamic partitioning. *J. Funct. Program.*, 2(4):407–423, 1992.

[Bou92b] François Bourdoncle. *Sémantiques des Langages Impératifs d'Ordre Supérieur et Interprétation Abstraite*. PhD thesis, École polytechnique, 1992.

[BR06] Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 221–239. Springer, 2006.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[CC77a] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, pages 237–277. North-Holland, 1977.

[CC77b] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of 4th ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. POPL*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[CC94] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 95–112, Toulouse, France, 16–19 May 1994. IEEE Computer Society Press, Los Alamitos, California.

[CC02] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Proc. of CC ETAPS*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2002.

[CCF$^+$06a] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. In *Proc. of ASIAN'06*, volume 4435 of *LNCS*, pages 272–300. Springer, Dec. 2006.

[CCF$^+$06b] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer, 2006.

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 105–118. ACM, 2011.

[CDD$^+$15a] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NFM*, pages 3–11. Springer, 2015.

[CDD$^+$15b] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015.

[CDG$^+$07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007. release October, 12th 2007.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of POPL*, pages 84–96. ACM Press, 1978.

[CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279. ACM, 2000.

[Che68] N. V. Chernikova. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Comput. Math. Math. Phys.*, 1968.

[CKK$^+$12] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27:573–609, 2012.

[Cou97] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electr. Notes Theor. Comput. Sci.*, 6:77–102, 1997.

[Cou03] Patrick Cousot. Verification by abstract interpretation. In Nachum Dershowitz, editor, *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 243–268. Springer, 2003.

[DPV11] Kamil Dudka, Petr Peringer, and Tomás Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 372–378, 2011.

[DRS01] Nurit Dor, Michael Rodeh, and Shmuel Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In Patrick Cousot, editor, *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2001.

[Fer01] Jérôme Feret. Abstract interpretation-based static analysis of mobile ambients. In *Proc. of SAS*, number 2126 in LNCS. Springer-Verlag, 2001. © Springer-Verlag.

[Fer04] Jérôme Feret. Static analysis of digital filters. In *European Symposium on Programming (ESOP'04)*, number 2986 in LNCS. Springer-Verlag, 2004. © Springer-Verlag.

[FL10] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, volume 6528 of *Lecture Notes in Computer Science*, pages 10–30. Springer, 2010.

[FMP13] Alexis Fouilhé, David Monniaux, and Michaël Périn. Efficient Generation of Correctness Certificates for the Abstract Domain of Polyhedra. In *SAS*, volume 7935 of *LNCS*. Springer, 2013.

[FOM18] A. Fromherz, A. Ouadjaout, and A. Miné. Static value analysis of Python programs by abstract interpretation. In *Proc. of NFM'18*, LNCS, pages 185–202. Springer, Apr. 2018.

[Gal08] Tristan Le Gall. *Abstract lattices for the verification of systèmes with stacks and queues.* PhD thesis, University of Rennes 1, France, 2008.

[GDD$^+$04] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In *Proc. of TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 512–529. Springer, 2004.

[GGLM12] Thomas Genet, Tristan Le Gall, Axel Legay, and Valérie Murat. Tree regular model checking for lattice-based automata. *CoRR*, abs/1203.1495, 2012.

[GJJ06] Tristan Le Gall, Bertrand Jeannet, and Thierry Jéron. Verification of communication protocols using abstract interpretation of FIFO queues. In *Proc. of AMAST*, volume 4019 of *Lecture Notes in Computer Science*, pages 204–219. Springer, 2006.

[Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics - IJCM*, 30:165–190, 01 1989.

[Gra91] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT, Vol.1*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192. Springer, 1991.

[Gra97] Philippe Granger. Static analyses of congruence properties on rational numbers (extended abstract). In *SAS*, volume 1302 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 1997.

[HIRV07] Peter Habermehl, Radu Iosif, Adam Rogalewicz, and Tomás Vojnar. Proving termination of tree manipulating programs. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *Proc. of ATVA*, volume 4762 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2007.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[JGR05] Bertrand Jeannet, Denis Gopan, and Thomas W. Reps. A relational abstraction for functions. In *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 186–202. Springer, 2005.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[JM18] Matthieu Journault and Antoine Miné. Inferring functional properties of matrix manipulating programs by abstract interpretation. *Formal Methods in System Design*, 53(2):221–258, 2018.

[JMO18] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. Modular static analysis of string manipulations in C programs. In *SAS*, volume 11002 of *Lecture Notes in Computer Science*, pages 243–262. Springer, 2018.

[JMO19] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. An abstract domain for trees with numeric relations. In *ESOP*, volume 11423 of *Lecture Notes in Computer Science*, pages 724–751. Springer, 2019.

[Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[KS18] Julian Kranz and Axel Simon. Modular analysis of executables using on-demand heyting completion. In *VMCAI*, volume 10747 of *Lecture Notes in Computer Science*, pages 291–312. Springer, 2018.

[KWN$^+$10] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Astrée: Proving the absence of runtime errors. In *Proc. of ERTS2 2010*, May 2010.

[LA04] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of CGO'04*, Mar 2004.

[Le 97] Gérard Le Lann. An analysis of the Ariane 5 flight 501 failure-a system engineering perspective. In *Workshop on Engineering of Computer-Based Systems (ECBS'97)*, pages 339–246, 1997.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[LR15] Jiangchao Liu and Xavier Rival. Abstraction of optional numerical values. In *APLAS*, volume 9458 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2015.

[LT93] Nancy G. Leveson and Clark Savage Turner. Investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[LV92] Hervé Le Verge. A Note on Chernikova's algorithm. Research Report RR-1662, INRIA, 1992.

[Mas92] François Masdupuy. Array abstractions using semantic analysis of trapezoid congruences. In *ICS*, pages 226–235. ACM, 1992.

[Mas93] François Masdupuy. Semantic analysis of interval congruences. In *Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 142–155. Springer, 1993.

[Mas01] Isabella Mastroeni. Numerical power analysis. In *PADO*, volume 2053 of *Lecture Notes in Computer Science*, pages 117–137. Springer, 2001.

[Mau99] Laurent Mauborgne. *Representation of Sets of Trees for Abstract Interpretation*. PhD thesis, Ecole polytechnique, 1999.

[Min01a] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2001.

[Min01b] Antoine Miné. The octagon abstract domain. In *Proc. of WCRE*, page 310. IEEE Computer Society, 2001.

[Min06a] Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In Mary Jane Irwin and Koen De Bosschere, editors, *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06), Ottawa, Ontario, Canada, June 14-16, 2006*, pages 54–63. ACM, 2006.

[Min06b] Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *Proc. of VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 348–363. Springer, 2006.

[Min13] Antoine Miné. *Static analysis by abstract interpretation of concurrent programs. (Analyse statique par interprétation abstraite de programmes concurrents)*. 2013.

[MMP17] Alexandre Maréchal, David Monniaux, and Michaël Périn. Scalable minimizing-operators on polyhedra via parametric linear programming. In *SAS*, volume 10422 of *LNCS*, 2017.

[MOJ18] A. Miné, A. Ouadjaout, and M. Journault. Design of a Modular Platform for Static Analysis. In *Proc. of (TAPAS)*, Lecture Notes in Computer Science (LNCS), page 4, 28 Aug. 2018.

[Mon99] David Monniaux. Abstracting cryptographic protocols with tree automata. In *Proc. of SAS*, number 1694 in Lecture Notes in Computer Science, pages 149–163. Springer Verlag, 1999.

[MRTT53] T.S. Motzkin, H. Raiffa, G.L. Thompson, and R.M. Thrall. The double description method. *Contributions to the Theory of Games*, 2:51–74, 1953.

[MS07] Markus Müller-Olm and Helmut Seidl. Analysis of modular arithmetic. *ACM Trans. Program. Lang. Syst.*, 29(5):29, 2007.

[NKH04] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 165–180. Springer, 2004.

[OHL$^+$12] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. *SIGPLAN Not.*, 47(6):229–238, June 2012.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of 17th IEEE (LICS 2002*, pages 55–74. IEEE Computer Society, 2002.

[RHC18] Andrew Ruef, Kesha Hietala, and Arlen Cox. Volume-based merge heuristics for disjunctive numeric domains. In *SAS*, volume 11002 of *Lecture Notes in Computer Science*, pages 383–401. Springer, 2018.

[RHS95] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61. ACM Press, 1995.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[RM07] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *Proc. of TOPLAS*, 29(5):26, 2007.

[Sak09] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

[Sim08] Axel Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer, 2008.

[SJ11] Pascal Sotin and Bertrand Jeannet. Precise interprocedural analysis in the presence of pointers to the stack. In Gilles Barthe, editor, *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pages 459–479. Springer, 2011.

[SK02] Axel Simon and Andy King. Analyzing string buffers in C. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2002.

[Ske92] R. Skeel. Roundoff error and the patriot missile. In *SIAM News*, volume 25. SIAM, 1992.

[SP78] M Sharir and A Pnueli. *Two approaches to interprocedural data flow analysis*. New York Univ. Comput. Sci. Dept., New York, NY, 1978.

[Spo05] F. Spoto. Julia: A generic static analyser for the Java bytecode. In *Proc. of FTfJP'2005*, page 17, July 2005.

[SR17] Tushar Sharma and Thomas W. Reps. A new abstraction framework for affine transformers. In Francesco Ranzato, editor, *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, volume 10422 of *Lecture Notes in Computer Science*, pages 342–363. Springer, 2017.

[Suz19] Thibault Suzanne. *Vérification par interprétation abstraite en mémoire faiblement cohérente*. PhD thesis, École normale supérieure, 2019.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[Tur37] Alan Turing. *On computable numbers, with an application to the Entscheidungsproblem*. 1937.

[WFBA00] David A. Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2000, San Diego, California, USA*. The Internet Society, 2000.

[WK03] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003.

[ZMNY14] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. Hybrid top-down and bottom-up interprocedural analysis. In *PLDI*, pages 249–258. ACM, 2014.